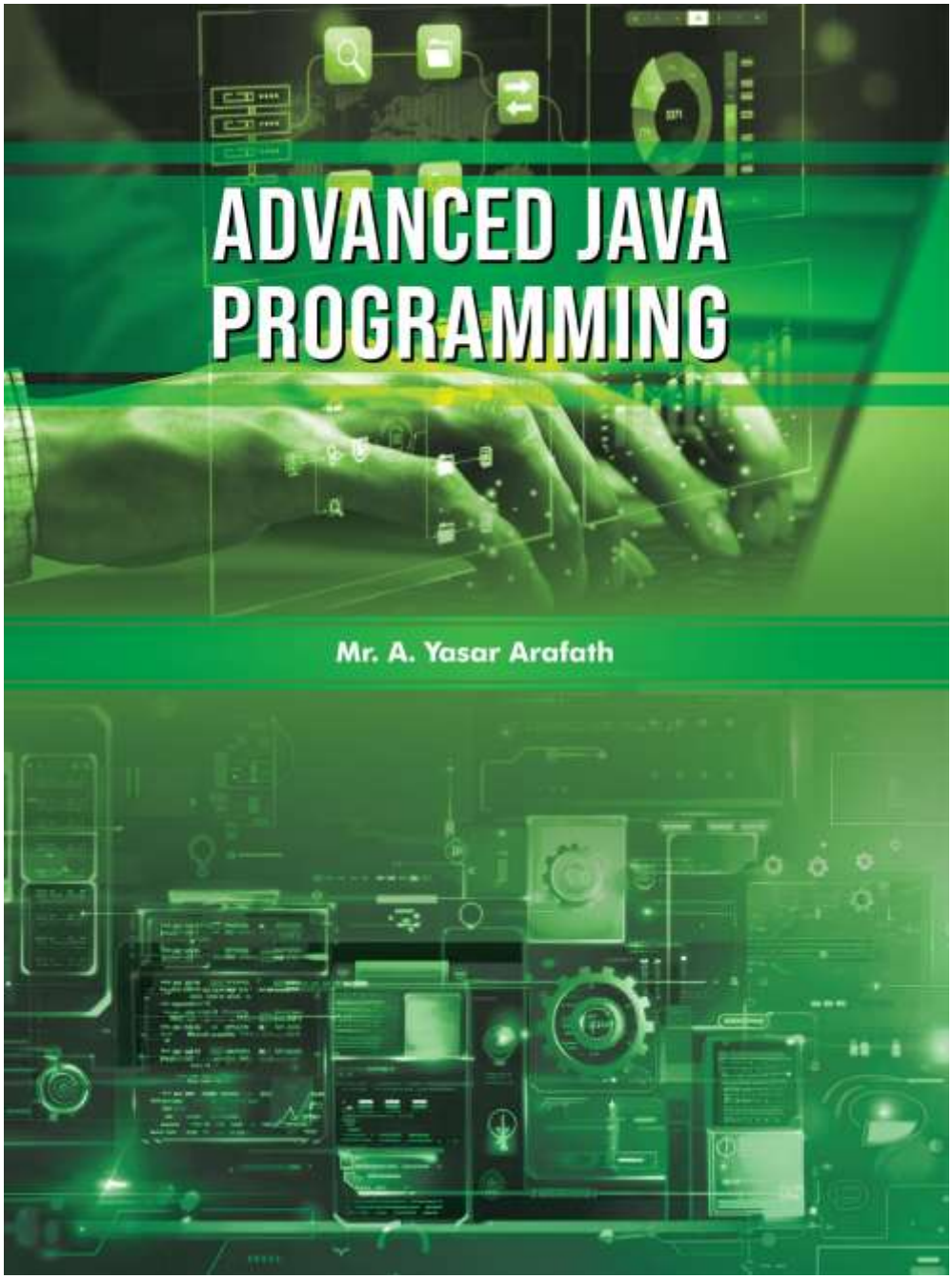

ADVANCED JAVA
PROGRAMMING
Mr. A. Yasar Arafath

Advanced Java Programming

Mr. A. Yasar Arafath MCA.,(PhD)
Assistant Professor
Department of Computer Applications
PET Engineering College
Vallioor – 627117

Edition Details (I,II,III): I

ISBN: **978-93-6786-382-4**

Month & Year: **October, 2025**

Pages: **447**

Price: **1000/-**`

About the Author

Mr. A. Yasar Arafath is a dedicated academician, researcher, and author with a profound expertise in the cutting-edge domains of Computer Science. Holding both a Bachelor of Computer Applications (BCA) and a Master of Computer Applications (MCA) with first-class distinctions, he has built a robust foundation in theoretical and applied computing. Currently serving as an **Assistant Professor** in the **Department of Computer Applications** at **PET Engineering College, Vallioor**, Mr. Arafath has amassed significant teaching experience, specializing in a wide array of subjects including Advanced Data Structures & Algorithms, Data Structures and Algorithms, Machine Learning, Digital Image Processing, Problem Solving and Programming in C, Basics of Computer Networks, Introduction to Computer Organization and Operating Systems, Object Oriented Software Engineering, Advanced Database Technology, DevOps and Microservices, Database Management System. His pedagogical excellence is demonstrated by the consistently outstanding results produced by his students. His forthcoming book titled, **"Advanced Java Programming,"** marks a significant milestone as his first single-authored work. This book distills his extensive practical and teaching experience into a comprehensive guide, designed to take readers from fundamental concepts to sophisticated, enterprise-level application development. It reflects his deep-seated commitment to creating impactful educational resources that bridge the gap between academic theory and industry practice. As an active researcher, his intellectual contributions are substantial. He has authored and published influential research papers in reputable international journals on topics such as **"Smart Soil Detection with Pseudo RGB Color Matching"** and **"Image Enhancement using Reduce Haze Algorithm."** His innovative spirit is further evidenced by a patented invention in the field of **deep learning for image-to-cartoon generation**, filed with the Government of India. His passion for knowledge dissemination is also reflected in his previous co-authored books on **"Advanced Data Structures and Algorithms"** and **"Artificial Intelligence and Machine Learning."** Committed to lifelong learning, he is an avid participant in the academic community, holding memberships in prestigious organizations like the **International Association of Engineers (IAENG)** and **International Organization for Academic and Scientific Development (IOASD).** He continuously enhances his expertise through numerous Faculty Development Programs, workshops, and certifications from leading institutions, focusing on Artificial Intelligence, Data Science and emerging technologies. Mr. Arafath currently pursuing his **Ph.D. as a Part-Time Research Scholar at Anna University, Chennai**, Mr. A. Yasar Arafath's research emphasizes **Artificial Intelligence applications in Precision Agriculture**. His doctoral work integrates **Deep Learning, IoT-based sensing, and data analytics** to enhance sustainable farming practices and optimize crop productivity. Mr. A. Yasar Arafath stands as a dynamic and knowledgeable voice in computer science education and research, whose written work is deeply informed by both practical industry knowledge and a commitment to academic excellence.

Preface

The book **"Advanced Java Programming"** is a comprehensive and detailed guide designed to help readers gain a deep understanding of advanced Java concepts, tools, and frameworks. Java, being one of the most powerful, platform-independent, and versatile programming languages, plays a vital role in enterprise application development, mobile computing, and web-based systems. This book is structured to take learners beyond the fundamentals and into professional-level Java programming, where efficiency, scalability, and security are essential.

It begins by revisiting the core concepts of object-oriented programming and gradually progresses into more complex topics such as **JDBC, Servlets, JSP, JavaFX, Multithreading, Exception Handling, RMI, and Networking**. Readers will also explore advanced frameworks like **Hibernate and Spring**, which are essential for developing robust, database-driven, and modular enterprise applications. Each chapter combines theoretical explanations with practical examples, coding exercises, and case studies to strengthen problem-solving and analytical skills.

The book focuses on building applications that are not only functional but also optimized for performance and maintainability. Emphasis is placed on design patterns, modular programming, and industry-standard practices that help readers adopt a clean, reusable, and scalable coding style. It provides detailed insights into **database connectivity**, **web-based application development**, and **framework integration**, helping learners connect Java concepts with real-world implementations. The content also highlights the importance of **security**, **data handling**, and **efficient resource management**, which are crucial in today's software industry. Readers will gain exposure to advanced concepts of **thread management, synchronization, and exception control**, enabling them to design applications capable of handling concurrent operations seamlessly.

The book not only strengthens technical proficiency but also cultivates logical thinking and creativity in problem-solving. Each topic has been presented in a clear, concise, and sequential manner to make learning smooth and engaging. By integrating both theoretical and practical approaches, the text ensures that readers can confidently apply what they learn in real-world projects. It is an invaluable resource for **students, educators, and professionals** aspiring to excel in Java programming.

The ultimate goal of this book is to empower readers with the ability to design, develop, and deploy **robust, efficient, and industry-ready Java applications**, aligning with current technological trends and software development standards. Through continuous learning and hands-on experience, this book inspires readers to master the art and science of advanced Java programming.

Mr. A. Yasar Arafath

Acknowledgement

The completion of this book on **Advanced Java Programming** has been an immensely fulfilling and intellectually enriching journey, made possible through the constant guidance, encouragement, and support of many individuals and institutions. We extend our deepest gratitude to everyone who has contributed, directly or indirectly, to the successful realization of this work.

First and foremost, we express our heartfelt appreciation to our **teachers, mentors, and colleagues**, whose valuable guidance, technical insights, and constructive suggestions have played a pivotal role in shaping the structure and content of this book. Their continuous encouragement and expertise have greatly enhanced the depth, clarity, and practical relevance of the topics presented.

We are sincerely thankful to our **families** for their unwavering love, patience, and moral support throughout the process of writing this book. Their understanding, motivation, and constant encouragement have been our source of strength and inspiration during every phase of this endeavor.

Our profound gratitude also extends to the **academic and professional communities in the field of computer science**, whose remarkable research, innovations, and open-source contributions have greatly influenced the development of Java and its advanced technologies. Their work has been a continuous source of learning and inspiration in compiling this comprehensive resource.

We would also like to acknowledge the immense role of **modern development tools, IDEs, and online programming resources**, which have aided us in coding, testing, and demonstrating practical Java implementations effectively. These tools have not only simplified complex programming concepts but also showcased the real-world applications of advanced Java technologies.

Above all, we offer our sincere thanks to **Almighty God** for His divine blessings, wisdom, and guidance, which have sustained us through challenges and enabled the successful completion of this work.

We hope this book serves as a valuable reference for **students, educators, and software professionals**, helping them deepen their understanding, enhance their programming skills, and explore the limitless possibilities of **Advanced Java Programming** in today's dynamic technological world.

Mr. A. Yasar Arafath

Table of Contents

CHAPTER – 1

CORE JAVA FUNDAMENTALS

1.1 Java Virtual Machine

The Java Virtual Machine (JVM) is a software-based engine that provides a runtime environment for executing Java programs. It translates Java bytecode into machine-specific code, enabling the "write once, run anywhere" (WORA) feature that allows Java applications to run seamlessly across different platforms and operating systems.

Acting as an intermediary between Java applications and the underlying hardware, the JVM manages essential functions such as memory allocation, garbage collection, and security.

As the core component of the Java Runtime Environment (JRE), the JVM ensures that Java programs can execute on any system without modification. It serves as an interpreter that converts Java bytecode into instructions understandable by the hardware.

The process works as follows:

Java source files (.java) are compiled by the Java compiler (javac) into bytecode (.class files). The JVM then loads this bytecode, verifies and links it, and finally executes it.

During execution, the JVM may interpret the bytecode or use Just-In-Time (JIT) compilation to translate frequently executed ("hot") code into native machine code, improving performance. Meanwhile, the garbage collector automatically reclaims memory from objects that are no longer in use, maintaining efficient memory management.

Architecture of JVM

The Java Virtual Machine (JVM) architecture consists of several key components that work together to execute Java programs efficiently and securely. It provides a platform-independent environment where Java bytecode is converted into machine-specific instructions.

The JVM architecture mainly includes the following components:

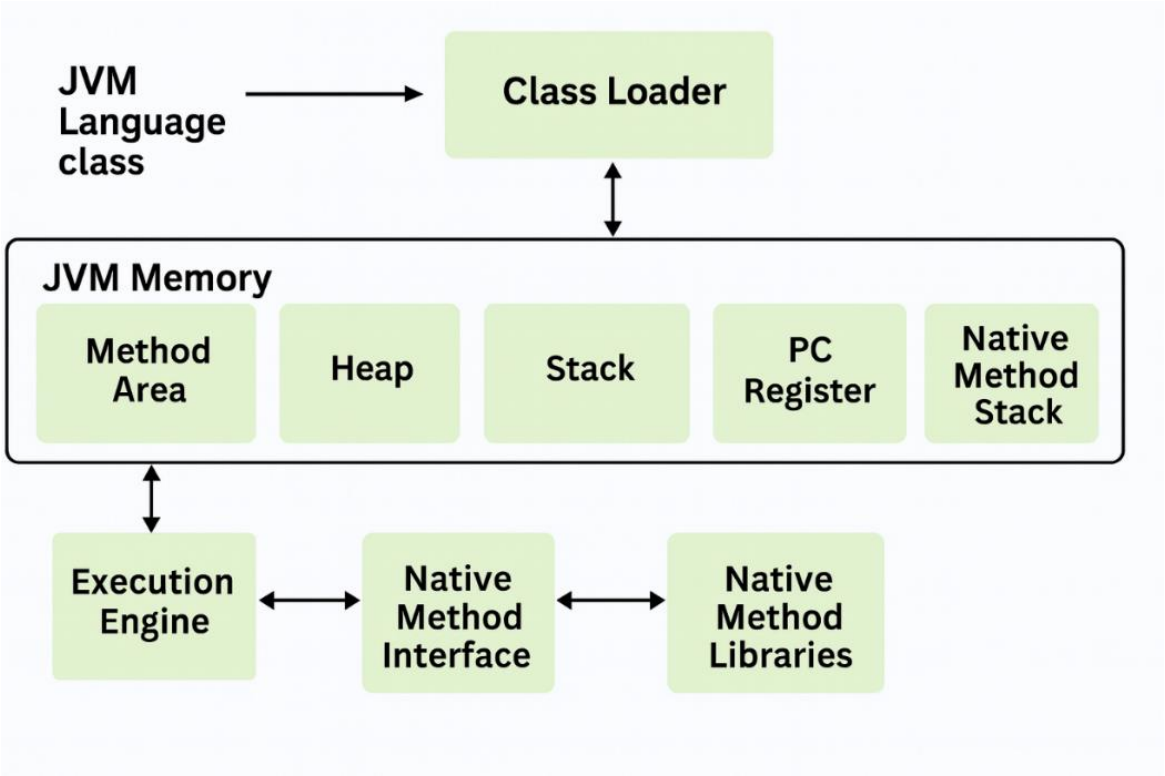


Fig. 1.1 Architecture of JVM.

Components of JVM Architecture

1. Class Loader Subsystem

It is mainly responsible for three activities.

1. Loading

During the **loading** phase, the **Class Loader** reads the .class file and stores the class metadata in the **Method Area**. It also creates a corresponding **Class object** in the **heap**, representing the loaded class in memory.

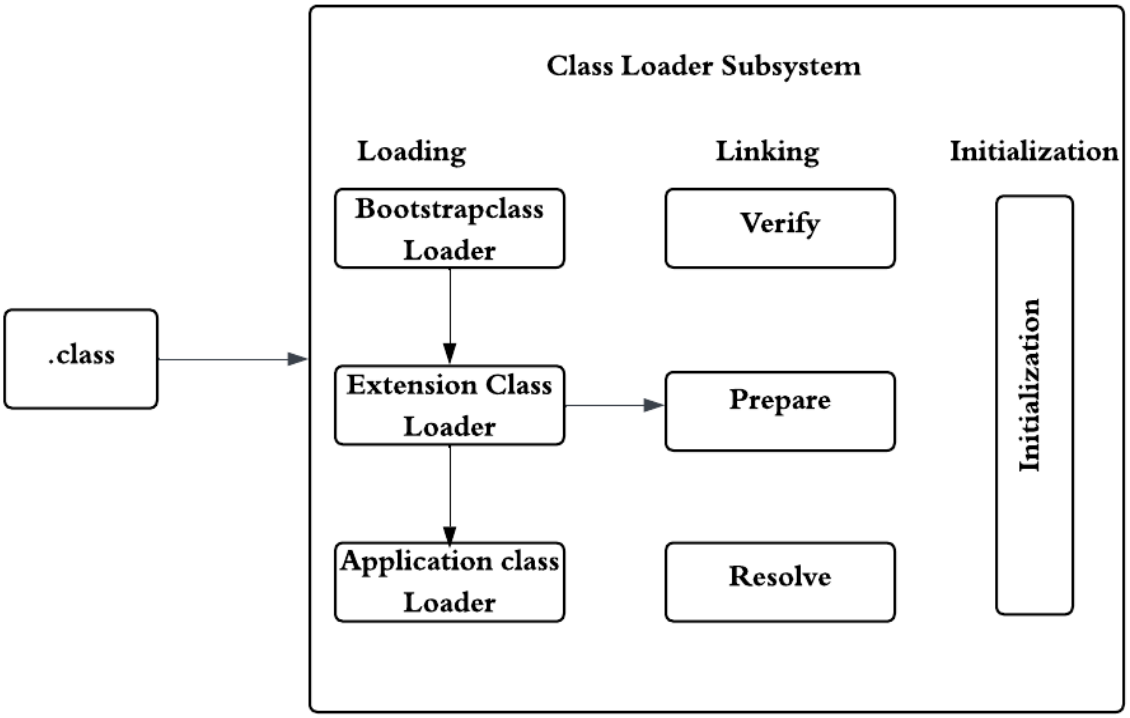


Fig. 1.2 Class Loader Subsystem.

Below is an example demonstrating how the JVM loads a class explicitly:

Example:

```
class Example {

    static {
        System.out.println("Example class has been loaded into the JVM!");
    }

    public void showMessage() {
        System.out.println("Method of Example class is now running.");
    }
}

public class LoaderTest {
    public static void main(String[] args) throws Exception {

        System.out.println("Program execution started.");

        // Explicitly loading the class using Class.forName()
```

```
        Class.forName("Example");

        System.out.println("Example class loaded successfully.");

        // Creating an object and invoking a method
        Example ex = new Example();
        ex.showMessage();
    }
}
```

Output:

```
Program execution started.
Example class has been loaded into the JVM!
Example class loaded successfully.
Method of Example class is now running.
```

2. Linking

The **linking** phase is responsible for preparing the loaded class so that it can be executed by the JVM. It ensures that the bytecode is valid, memory is allocated properly, and all references are correctly resolved. The linking process involves three main steps:

- **Verification:** Checks that the bytecode adheres to JVM specifications and is safe to execute.
- **Preparation:** Allocates memory for all static variables and assigns them default values.
- **Resolution:** Replaces symbolic references (names in the bytecode) with direct references to memory locations.

3. Initialization

During this phase, the JVM assigns actual values to static variables and executes all static blocks defined within the class. This is the final step before the class is ready for use.

Class Loader Types

1. **Bootstrap Class Loader:** Loads the core Java classes located in the JAVA_HOME/lib directory (e.g., java.lang, java.util).
2. **Extension Class Loader:** Loads classes from the JAVA_HOME/jre/lib/ext directory, which typically contains extension libraries.
3. **System/Application Class Loader:** Loads classes from the application's classpath (user-defined classes and libraries).

Example: Demonstrating the Class Loader Subsystem

```
public class Demo {
   public static void main(String[] args) {
      // String class is loaded by the Bootstrap Class Loader
      // Bootstrap loader is part of native JVM and not a Java object, so it returns null
      System.out.println("String class loader: " + String.class.getClassLoader());

      // Demo class is loaded by the Application (System) Class Loader
      System.out.println("Demo class loader: " + Demo.class.getClassLoader());
   }
}
```

Output:

```
String class loader: null
Demo class loader: jdk.internal.loader.ClassLoaders$AppClassLoader@7d4991ad
```

2. JVM Memory Areas

The **JVM memory** is divided into multiple areas to store various kinds of data during program execution:

- **Method Area:** Contains class-level data such as class names, parent classes, methods, variables, and static information. Shared among all threads.
- **Heap Area:** Used to store all Java objects and arrays. Shared across the entire JVM.
- **Stack Area:** Each thread has its own stack that holds method calls and local variables in stack frames. It is destroyed once the thread terminates.
- **Program Counter (PC) Register:** Maintains the address of the currently executing JVM instruction for each thread.
- **Native Method Stack:** Each thread has a separate stack to handle native (non-Java) method calls executed through JNI.

3. Execution Engine

The **Execution Engine** executes the bytecode (.class file) by reading and processing each instruction using data stored in different memory areas. It consists of three main components:

- **Interpreter:** Reads and executes bytecode instructions one by one. However, it reinterprets methods each time they are called, which can reduce performance.
- **Just-In-Time (JIT) Compiler:** Enhances performance by converting frequently executed bytecode into native machine code, allowing faster execution without reinterpreting.

- **Garbage Collector:** Automatically identifies and removes unused or unreferenced objects to free up memory.

4. Java Native Interface (JNI)

The **JNI** acts as a bridge between the JVM and native libraries written in languages like C or C++. It enables Java programs to call native methods and also allows native code to invoke Java methods. This interface is crucial for platform-specific functionality and hardware-level interactions.

5. Native Method Libraries

These are collections of **precompiled native libraries** (often written in C or C++) required for executing native methods through JNI. They allow the JVM to interact directly with the host operating system or hardware to perform low-level tasks efficiently.

1.2 Data Types

Java is a statically typed programming language, meaning that the type of each variable is determined at compile time.

In Java, the compiler knows the exact type of every variable and ensures that it is used correctly before the program runs. For instance, the statement int x = "GfG"; will cause a compilation error because a string value is being assigned to an integer variable, which is not allowed.

Data types in Java come in various sizes and ranges to suit different kinds of data and situations. This design allows developers to efficiently handle diverse data requirements.

Categories of Java Data Types

Primitive Data Types:

These are the most basic data types that store simple values directly in memory. Examples include boolean, char, byte, short, int, long, float, and double.

Non-Primitive Data Types (Reference Types):

These data types store references (memory addresses) of objects rather than the actual data. Common examples include String, Array, Class, Interface, and Object.

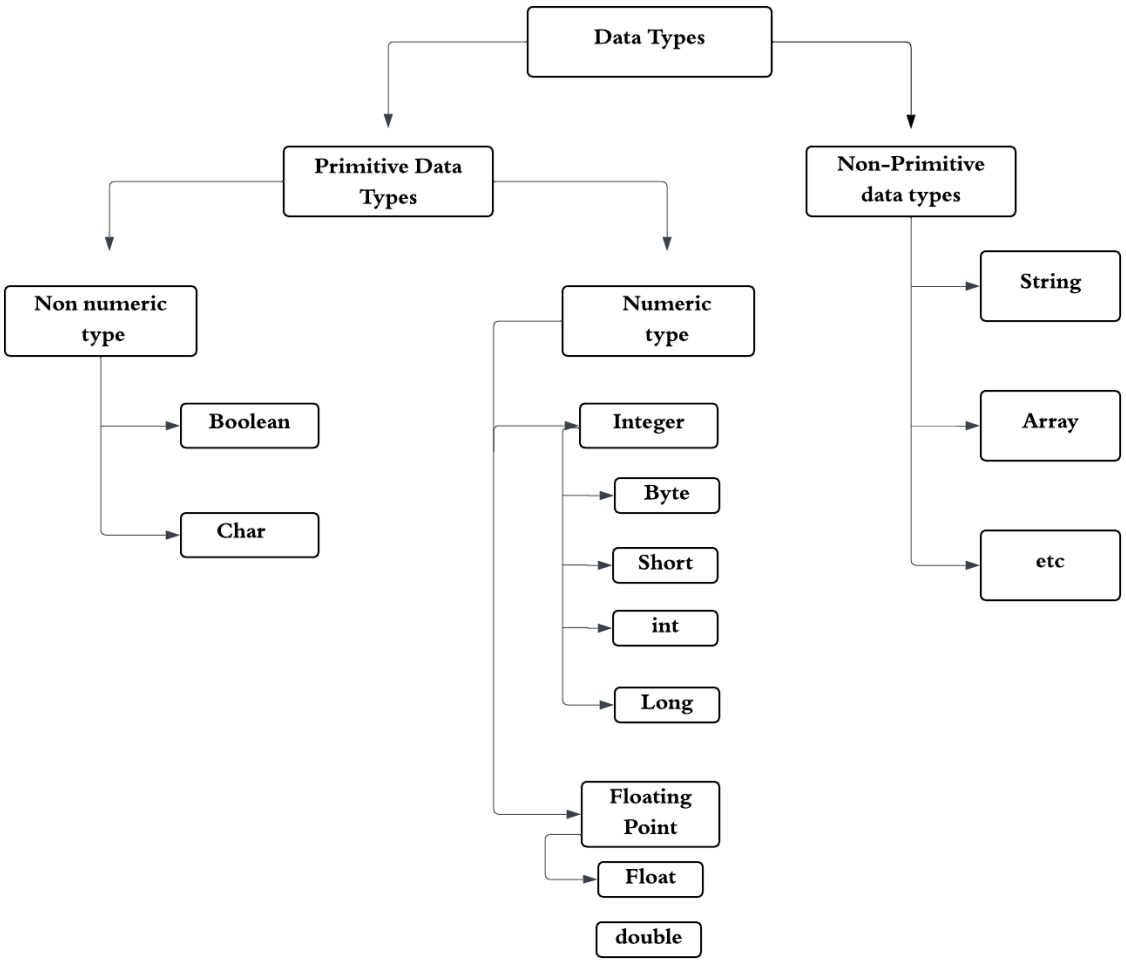


Fig. 1.3 Data types in Java.

Primitive Data Types

Primitive data store only single values and have no additional capabilities. There are 8 primitive data types. They are depicted below in tabular format below as follows:

Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	JVM-dependent (typically 1 byte)	true, false	true, false
byte	8-bit signed integer	0	1 byte	(none)	-128 to 127

char	Unicode character(16 bit)	\u0000	2 bytes	'a', '\u0041', '\101', '\\', '\', '\n', 'β'	0 to 65,535 (unsigned)
short	16-bit signed integer	0	2 bytes	(none)	-32,768 to 32,767
int	32-bit signed integer	0	4 bytes	-2,0,1	-2,147,483,648 to 2,147,483,647
long	64-bit signed integer	0L	8 bytes	-2L,0L,1L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	32-bit IEEE 754 floating-point	0.0f	4 bytes	3.14f, -1.23e-10f	~6-7 significant decimal digits
double	64-bit IEEE 754 floating-point	0.0d	8 bytes	3.1415d, 1.23e100d	~15-16 significant decimal digits

Table. 1.1 Primitive Data Types.

1. boolean Data Type

The **boolean** data type represents a logical value that can either be **true** or **false**. Conceptually, it stores a single bit of information, though in practice, the actual size used by the **Java Virtual Machine (JVM)** depends on the implementation (typically one byte or 8 bits). Boolean values cannot be implicitly or explicitly converted to other data types using casts, though manual conversions can be implemented if needed.

Syntax:

boolean flag;

Size: Virtual machine dependent (typically 1 byte or 8 bits)

Example: The following example shows how to use the **boolean** data type to represent logical conditions.

```
// Demonstrating boolean data type
public class BooleanExample {
```

```
    public static void main(String[] args) {
        boolean isRaining = false;
        boolean isSunny = true;

        System.out.println("Is it raining today? " + isRaining);
        System.out.println("Is it sunny today? " + isSunny);
    }
}
```

Output:

Is it raining today? false
Is it sunny today? true

2. byte Data Type

The **byte** data type is an **8-bit signed two's complement integer**. It is mainly used to save memory when dealing with large arrays of small numerical values, especially when data values fall within the byte range.

Syntax:

byte smallNum;

Size: 1 byte (8 bits)

Example: This example demonstrates how to use the **byte** data type to store and display small integer values.

```
// Demonstrating byte data type
public class ByteExample {
    public static void main(String[] args) {
        byte speed = 60;
        byte temperature = -5;

        System.out.println("Vehicle speed: " + speed + " km/h");
        System.out.println("Room temperature: " + temperature + "°C");
    }
}
```

Output:

Vehicle speed: 60 km/h
Room temperature: -5°C

3. short Data Type

The **short** data type is a **16-bit signed two's complement integer**. It is often used when memory savings are necessary, such as in large data arrays, and when the values fit within the short range.

Syntax:

short shortVar;

Size: 2 bytes (16 bits)

Example: The following program demonstrates how to use the **short** data type to represent moderately small integer values.

```
// Demonstrating short data type
public class ShortExample {
    public static void main(String[] args) {
        short population = 15000;
        short temperature = -15;

        System.out.println("Village population: " + population);
        System.out.println("Winter temperature: " + temperature + "°C");
    }
}
```

Output:

Village population: 15000
Winter temperature: -15°C

4. int Data Type

The **int** data type represents a **32-bit signed two's complement integer**. It is the most commonly used integer type in Java for handling whole numbers.

Syntax:

int intVar;

Size: 4 bytes (32 bits)

Example: This example demonstrates how to use the **int** data type to display large integer values.

```
// Demonstrating int data type
public class IntExample {
    public static void main(String[] args) {
        int population = 5000000;
        int distance = 120000000;

        System.out.println("City Population: " + population);
        System.out.println("Distance to Sun (in km): " + distance);
    }
}
```

Output:

```
City Population: 5000000
Distance to Sun (in km): 120000000
```

5. long Data Type

The **long** data type is a **64-bit signed two's complement integer**, used when an **int** is not sufficient to hold very large values.

Syntax:

```
long longVar;
```

Size: 8 bytes (64 bits)

Note: Since Java SE 8, long can also represent an **unsigned 64-bit value**, ranging from 0 to 264−12^{64} - 1264−1. The Long class includes utility methods for unsigned arithmetic such as compareUnsigned() and divideUnsigned().

Example: The following program demonstrates how to use the **long** data type to handle very large numbers.

```
// Demonstrating long data type
public class LongExample {
    public static void main(String[] args) {
        long stars = 9500000000L;
        long galaxyDistance = 120000000000000L;

        System.out.println("Number of Stars: " + stars);
        System.out.println("Galaxy Distance (in light years): " + galaxyDistance);
    }
}
```

Output:

Number of Stars: 9500000000
Galaxy Distance (in light years): 120000000000000

6. float Data Type

The **float** data type is a **single-precision 32-bit IEEE 754 floating-point** number. It's useful for saving memory in large arrays of floating-point data when precision is not critical.

Syntax:

float floatVar;

Size: 4 bytes (32 bits)

Example: This example shows how to use the **float** data type to represent decimal values.

```
// Demonstrating float data type
public class FloatExample {
    public static void main(String[] args) {
        float interestRate = 7.5f;
        float temperature = 36.6f;

        System.out.println("Bank Interest Rate: " + interestRate + "%");
        System.out.println("Body Temperature: " + temperature + "°C");
    }
}
```

Output:

Bank Interest Rate: 7.5%
Body Temperature: 36.6°C

7. double Data Type

The **double** data type represents a **double-precision 64-bit IEEE 754 floating-point** number. It is generally the default choice for representing decimal or fractional values in Java.

Syntax:

double doubleVar;

Size: 8 bytes (64 bits)

Example: This example demonstrates how to use the **double** data type for precise decimal calculations.

```
// Demonstrating double data type
public class DoubleExample {
   public static void main(String[] args) {
      double gravity = 9.80665;
      double avogadro = 6.022140857e23;

      System.out.println("Acceleration due to Gravity: " + gravity);
      System.out.println("Avogadro's Number: " + avogadro);
   }
}
```

Output:

Acceleration due to Gravity: 9.80665
Avogadro's Number: 6.022140857E23

8. char Data Type

The **char** data type stores a single **16-bit Unicode character**, allowing representation of a wide range of symbols and international characters.

Syntax:

```
char charVar;
```

Size: 2 bytes (16 bits)

Example: The following example demonstrates how to use the **char** data type to store individual characters.

```
// Demonstrating char data type
public class CharExample {
   public static void main(String[] args) {
      char grade = 'B';
      char symbol = '#';

      System.out.println("Student Grade: " + grade);
      System.out.println("Symbol: " + symbol);
   }}
```

Output:

Student Grade: B
Symbol: #

Why is the Size of char 2 Bytes in Java?

Unlike C or C++, which use the **ASCII** character set (limited to 8 bits), Java uses **Unicode**, a character set designed for **internationalization**.Unicode requires more than 8 bits to represent characters from multiple languages like Latin, Greek, Chinese, Arabic, and others. Therefore, Java uses **2 bytes (16 bits)** to store each character, ensuring compatibility across global languages.

Example: Demonstrating Various Primitive Data Types

```
// Java Program to Demonstrate Primitive Data Types
public class PrimitiveExample {
    public static void main(String[] args) {
        char letter = 'J';
        int number = 150;
        byte age = 25;
        short code = 1024;
        float price = 19.99f;
        double distance = 384400.75;
        long population = 1390000000L;

        System.out.println("char: " + letter);
        System.out.println("int: " + number);
        System.out.println("byte: " + age);
        System.out.println("short: " + code);
        System.out.println("float: " + price);
        System.out.println("double: " + distance);
        System.out.println("long: " + population);
    }
}
```

Output:

char: J
int: 150
byte: 25
short: 1024
float: 19.99
double: 384400.75

long: 1390000000

Non-Primitive (Reference) Data Types

Non-Primitive (Reference) Data Types store **memory addresses** of actual data instead of the data itself. They include **Strings**, **Objects**, **Arrays**, **Classes**, and **Interfaces**.

1. Strings

Strings are sequences of characters stored as objects. Unlike C/C++, Java strings are not null-terminated. Strings in Java are **immutable**, meaning once created, their value cannot be changed.

Syntax:

String str = "Hello, Java!";

Example:

```
// Demonstrating String data type
public class StringExample {
    public static void main(String[] args) {
        String name = "Alice";
        String greeting = "Welcome to Java Programming!";

        System.out.println("Name: " + name);
        System.out.println("Greeting: " + greeting);
    }
}
```

Output:

Name: Alice
Greeting: Welcome to Java Programming!

2. Class

A **Class** in Java is a blueprint for creating objects. It defines the properties (fields) and behaviors (methods) that its objects will have.

Example:

```
// Demonstrating how to create a class
class Laptop {
```

```
    String brand;
    int year;
    Laptop(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    void display() {
        System.out.println("Laptop: " + brand + " (" + year + ")");
    }
}

public class ClassExample {
    public static void main(String[] args) {
        Laptop myLaptop = new Laptop("Dell", 2022);
        myLaptop.display();
    }
}
```

Output:

Laptop: Dell (2022)

3. Object

An **Object** represents a real-world entity that has **state**, **behavior**, and **identity**. Objects are created from classes and interact by calling each other's methods.

Example:

```
// Define the Car class
class Vehicle {
    String model;
    int year;

    Vehicle(String model, int year) {
        this.model = model;
        this.year = year;
    }
}

// Demonstrating object creation
public class ObjectExample {
    public static void main(String[] args) {
        Vehicle myVehicle = new Vehicle("Tesla Model 3", 2023);
```

```
        System.out.println("Vehicle Model: " + myVehicle.model);
        System.out.println("Vehicle Year: " + myVehicle.year);
    }
}
```

Output:

Vehicle Model: Tesla Model 3
Vehicle Year: 2023

4. Interface

An **Interface** defines a contract of methods that a class must implement.It specifies what a class should do, but not *how* it does it.

Example:

```
// Demonstrating interface implementation
interface Animal {
    void sound();
}

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

public class InterfaceDemo {
    public static void main(String[] args) {
        Cat myCat = new Cat();
        myCat.sound();
    }
}
```

Output:

Meow

5. Array

An **Array** is a collection of elements of the same type, stored at contiguous memory locations. In Java, arrays are dynamically allocated and can be accessed using an index starting from 0.

Example:

```
// Demonstrating array usage
public class ArrayExample {
    public static void main(String[] args) {
        int[] scores = {90, 85, 78, 92, 88};
        String[] fruits = {"Apple", "Banana", "Cherry"};

        System.out.println("First Score: " + scores[0]);
        System.out.println("Second Fruit: " + fruits[1]);
    }
}
```

Output:

```
First Score: 90
Second Fruit: Banana
```

Primitive vs Non-Primitive Data Types

Aspect	Primitive Data Types	Non-Primitive Data Types
Memory	Stored on the **stack**	Stored on the **heap**
Speed	Faster in execution	Slower in comparison
Example	int x = 5;	String s = "Java";

Table. 1.2 Primitive vs Non-Primitive Data Types.

1.3 Variables and Keywords

Variable

A **variable** provides a named location in memory that allows programs to store and manipulate data. Each variable in Java has a **specific data type**, which determines:

- The **amount of memory** it occupies
- The **range of values** it can hold
- The **operations** that can be performed on it.

Variable Declaration and Initialization

Variables must be declared before they can be used.A variable declaration includes the **data type** followed by the **variable name**.To assign a value, use the **assignment (=)** operator. Each statement ends with a semicolon (;).

Syntax:

data_type variable_name [= value] [, variable_name [= value]] ...;

Here,

- data_type specifies the kind of value the variable can hold.
- variable_name is the name given to the variable.
- Multiple variables of the same type can be declared in one line using commas.

Examples of Valid Variable Declarations and Initializations

```
int x, y, z;          // Declares three integer variables
int a = 5, b = 15;    // Declares and initializes two integers
byte code = 20;       // Declares and initializes a byte variable
double pi = 3.14159;  // Declares and initializes a double
char grade = 'A';     // Declares and initializes a char variabl
```

Types of Variables in Java

Java supports **three main types** of variables:

1. **Local Variables**
2. **Instance Variables**
3. **Class (Static) Variables**

1. Local Variables

- Declared **inside** a method, constructor, or block.
- Created when the method is invoked and **destroyed** when it exits.
- **Access modifiers** (like public or private) cannot be used.
- Only **visible within their scope** (method, constructor, or block).
- Stored in the **stack memory**.
- Have **no default value** must be initialized before use.

Example 1: Local Variable with Initialization

```
public class DogAge {
```

```
   public void puppyAge() {
      int age = 2;  // Local variable initialized
      age = age + 5;
      System.out.println("Puppy age is: " + age);
   }

   public static void main(String[] args) {
      DogAge dog = new DogAge();
      dog.puppyAge();
   }
}
```

Output:

```
Puppy age is: 7
```

Example 2: Local Variable without Initialization

```
public class DogAgeError {
   public void puppyAge() {
      int age;  // Declared but not initialized
      age = age + 3; // Compilation error
      System.out.println("Puppy age is: " + age);
   }

   public static void main(String[] args) {
      DogAgeError dog = new DogAgeError();
      dog.puppyAge();
   }
}
```

Output (Compilation Error):

```
DogAgeError.java:4: variable age might not have been initialized
age = age + 3;
      ^
1 error
```

2. Instance Variables

- Declared **inside a class** but **outside** any method, constructor, or block.
- Each **object** of the class gets its own copy of instance variables.
- Created when an object is instantiated using new and destroyed when the object is destroyed.
- Used to represent **object properties or states**.

- Can have **access modifiers** (public, private, protected).
- Have **default values** (0 for numbers, false for boolean, and null for objects).
- Accessed using either the **object reference** or directly inside non-static methods.

Example: Java Instance Variables

```
public class Student {
    // Instance variables
    public String name;     // Visible to other classes
    private double marks;   // Private variable visible only within this class
    // Constructor to assign name
    public Student(String studentName) {
        name = studentName;
    }

    // Setter method for marks
    public void setMarks(double studentMarks) {
        marks = studentMarks;
    }

    // Method to print student details
    public void printDetails() {
        System.out.println("Student Name: " + name);
        System.out.println("Marks: " + marks);
    }

    public static void main(String[] args) {
        Student student1 = new Student("Priya");
        student1.setMarks(92.5);
        student1.printDetails();
    }
}
```

Output:

```
Student Name: Priya
Marks: 92.5
```

3. Class (Static) Variables

- Declared with the **static keyword** inside a class but **outside** any method, constructor, or block.
- There is **only one copy** of a static variable for the entire class, shared by all objects.
- Typically used for **constants** or **shared properties**.

- Stored in **static memory**, created when the program starts and destroyed when it ends.
- Usually declared as public static final for constants.
- Accessed using the **class name** (e.g., ClassName.variableName).
- Default values are the same as instance variables.

Example: Java Static Variables

```
public class Company {
   // Static variable
   private static double bonusPercentage;

   // Constant
   public static final String DEPARTMENT = "Human Resources";

   public static void main(String[] args) {
      bonusPercentage = 12.5;
      System.out.println(DEPARTMENT + " department bonus: " + bonusPercentage + "%");
   }
}
```

Output:

Human Resources department bonus: 12.5%

Note: If a static variable or constant is accessed from another class, it must be referenced using the class name, e.g., Company.DEPARTMENT.

Variable Type	Declared In	Memory Location	Default Value	Access Modifiers Allowed	Scope
Local Variable	Inside methods, constructors, or blocks	Stack	None (must initialize)	No	Only within the block/method
Instance Variable	Inside class but outside methods	Heap	Yes (depends on type)	Yes	Throughout the class
Static Variable	Inside class with static keyword	Static memory	Yes (depends on type)	Yes	Shared among all instances

Table. 1.3 Summary of Variables.

Keywords

Keywords are **reserved words** that have predefined meanings and are recognized by the **Java compiler** for specific internal processes or predefined actions. These words **cannot be used as identifiers** (i.e., variable names, method names, class names, or object names) because they are part of the language syntax.

Example: Demonstrating Java Keywords

```
// Java Program to demonstrate the use of keywords
class KeywordExample {
    public static void main(String[] args) {
        // Using final and int keywords
        final int score = 75;

        // Using if and else keywords
        if (score >= 50) {
            System.out.println("Passed the test successfully!");
        } else {
            System.out.println("Failed to pass the test.");
        }
    }
}
```

Output:

Passed the test successfully!

What Happens If We Use a Variable Name Same as a Keyword?

If you try to use a **keyword** as a variable name, the Java compiler will throw an error because keywords are **reserved** and cannot be redefined or repurposed.

Example: Attempting to Use a Keyword as a Variable Name

```
// Java Program to illustrate using a keyword as a variable name
class InvalidKeywordExample {
    public static void main(String[] args) {
        // The word "class" is a reserved keyword in Java
        String class = "Hello, Java!";
        System.out.println(class);
    }
}
```

Output (Compilation Error):

```
InvalidKeywordExample.java:5: error: not a statement
        String class = "Hello, Java!";
               ^
InvalidKeywordExample.java:5: error: ';' expected
        String class = "Hello, Java!";
                    ^
2 errors
```

Important Points About Java Keywords

- The keywords **const** and **goto** are reserved in Java but **not currently used**.
- The words **true**, **false**, and **null** are **literals**, not actual keywords however, they **cannot be used as identifiers**.
- Java keywords are **case-sensitive** using uppercase (like IF instead of if) will result in a compilation error.

Java Keywords List

As of **Java 21**, there are **53 keywords** in the Java programming language. Most IDEs highlight these words in a different color to differentiate them from user-defined identifiers.

Below is a list of all **Java keywords** and their **primary purpose**:

Keyword	Usage / Description
abstract	Specifies that a class or method must be implemented later in a subclass.
assert	Used to test assumptions during program execution for debugging.
boolean	Data type representing true or false values.
break	Exits a loop or switch statement.
byte	Data type for 8-bit signed integers.
case	Defines individual branches within a switch statement.
catch	Handles exceptions thrown by try blocks.
char	Data type for 16-bit Unicode characters.
class	Declares a new class.
const	Reserved but not used in Java.
continue	Skips the current iteration of a loop and moves to the next.
default	Specifies the default case in a switch statement.
do	Begins a do-while loop.
double	Data type for 64-bit floating-point numbers.
else	Defines an alternative branch in an if statement.
enum	Declares an enumerated type (set of named constants).

extends	Indicates that a class inherits from another class.
final	Declares constants, prevents method overriding, or inheritance.
finally	Defines a block of code that always executes after a try-catch.
float	Data type for 32-bit floating-point numbers.
for	Begins a for loop.
goto	Reserved but not used.
if	Tests a condition and executes code accordingly.
implements	Indicates that a class implements an interface.
import	Includes other Java classes or packages.
instanceof	Tests whether an object is an instance of a particular class.
int	Data type for 32-bit signed integers.
interface	Declares an interface (a collection of abstract methods).
long	Data type for 64-bit signed integers.
native	Specifies that a method is implemented in native code (e.g., C/C++).
new	Creates new objects.
null	Represents a reference that points to no object.
package	Declares a namespace for classes.
private	Access modifier that restricts access to within the same class.
protected	Access modifier allowing access within subclasses or same package.
public	Access modifier making members visible everywhere.
return	Exits a method and optionally returns a value.
short	Data type for 16-bit signed integers.
static	Defines class-level variables or methods.
strictfp	Ensures consistent floating-point calculations across platforms.
super	Refers to the parent class (used for inheritance and constructors).
switch	Executes one block of code among many options based on a test value.
synchronized	Used to handle thread synchronization in multithreading.
this	Refers to the current object in a class.
throw	Creates (throws) an exception manually.
throws	Declares exceptions a method might throw.
transient	Marks variables not to be serialized.
try	Starts a block of code that may throw exceptions.
void	Specifies that a method does not return a value.
volatile	Indicates that a variable may be modified unexpectedly (used in threads).
while	Begins a while loop.
sealed	Restricts which other classes can extend this class.
permits	Used in sealed classes to define which subclasses are allowed.

Example: Demonstrating sealed and permits (Java 21 Feature)

```
// Demonstrating the use of sealed and permits keywords
sealed class Shape permits Circle, Square { }

final class Circle extends Shape {
    void area() {
        System.out.println("Area of Circle = πr²");
    }
}

final class Square extends Shape {
    void area() {
        System.out.println("Area of Square = side × side");
    }
}

public class SealedExample {
    public static void main(String[] args) {
        Circle c = new Circle();
        Square s = new Square();

        c.area();
        s.area();
    }
}
```

Output:

```
Area of Circle = πr²
Area of Square = side × side
```

1.4 Operators

Operators are special symbols used to perform specific operations on variables or values. They play a crucial role in programming by enabling efficient data manipulation and computation.

Operators are special symbols used to perform specific operations on variables and values. They are fundamental in programming as they allow developers to perform arithmetic, logical, and comparison-based operations efficiently on data.

1. Arithmetic Operators

Arithmetic Operators are used to perform basic mathematical calculations such as addition, subtraction, multiplication, division, and modulus on numeric data types.

```
// Demonstrating Arithmetic Operators
public class ArithmeticExample {
    public static void main(String[] args) {
        int x = 15, y = 4;

        int sum = x + y;     // Addition
        int diff = x - y;    // Subtraction
        int product = x * y; // Multiplication
        int div = x / y;     // Division
        int mod = x % y;     // Modulus (remainder)

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + diff);
        System.out.println("Product: " + product);
        System.out.println("Division: " + div);
        System.out.println("Modulus: " + mod);
    }
}
```

Output:

```
Sum: 19
Difference: 11
Product: 60
Division: 3
Modulus: 3
```

2. Unary Operators

Unary Operators operate on a single operand. They are mainly used for **incrementing**, **decrementing**, or **negating** values.

```
// Demonstrating Unary Operators
public class UnaryExample {
    public static void main(String[] args) {
        int num1 = 7;
        int num2 = 7;

        System.out.println("Post-increment: " + (num1++));
```

```
        System.out.println("Pre-increment: " + (++num1));

        System.out.println("Post-decrement: " + (num2--));
        System.out.println("Pre-decrement: " + (--num2));
    }
}
```

Output:

```
Post-increment: 7
Pre-increment: 9
Post-decrement: 7
Pre-decrement: 5
```

3. Assignment Operators

The **Assignment Operator** (=) assigns values from the right-hand side to the left-hand variable. **Compound assignment operators** (+=, -=, *=, /=, %=) combine arithmetic with assignment for cleaner code.

```
// Demonstrating Assignment Operators
public class AssignmentExample {
    public static void main(String[] args) {
        int n = 8;

        n += 4;  // n = n + 4
        System.out.println("After += : " + n);

        n *= 3;  // n = n * 3
        System.out.println("After *= : " + n);

        n -= 6;  // n = n - 6
        System.out.println("After -= : " + n);

        n /= 2;  // n = n / 2
        System.out.println("After /= : " + n);

        n %= 5;  // n = n % 5
        System.out.println("After %= : " + n);
    }
}
```

Output:

After += : 12
After *= : 36
After -= : 30
After /= : 15
After %= : 0

4. Relational Operators

Relational Operators are used to compare two values. They return a **boolean** (true or false) result. These are commonly used in **conditional** and **looping** statements.

```
// Demonstrating Relational Operators
public class RelationalExample {
    public static void main(String[] args) {
        int p = 12;
        int q = 8;
        int r = 12;

        System.out.println("p > q: " + (p > q));
        System.out.println("p < q: " + (p < q));
        System.out.println("p >= q: " + (p >= q));
        System.out.println("p <= q: " + (p <= q));
        System.out.println("p == r: " + (p == r));
        System.out.println("p != r: " + (p != r));
    }
}
```

Output:

p > q: true
p < q: false
p >= q: true
p <= q: false
p == r: true
p != r: false

5. Logical Operators

Logical Operators are used to combine multiple boolean expressions.They behave like logic gates && (AND), || (OR), and ! (NOT).They also exhibit **short-circuiting** behavior (if the result is known from the first condition, the second is not evaluated).

```
// Demonstrating Logical Operators
public class LogicalExample {
    public static void main(String[] args) {
        boolean isJavaEasy = true;
        boolean isDifficult = false;

        System.out.println("isJavaEasy && isDifficult: " + (isJavaEasy && isDifficult));
        System.out.println("isJavaEasy || isDifficult: " + (isJavaEasy || isDifficult));
        System.out.println("!isJavaEasy: " + (!isJavaEasy));
    }
}
```

Output:

```
isJavaEasy && isDifficult: false
isJavaEasy || isDifficult: true
!isJavaEasy: false
```

6. Ternary Operator

The **Ternary Operator (?:)** is a concise alternative to the if-else statement.It takes three operands: a condition, a result if true, and a result if false.

```
// Demonstrating Ternary Operator
public class TernaryExample {
    public static void main(String[] args) {
        int x = 25, y = 40, z = 30;
        int max;

        // Finding the maximum value using nested ternary operators
        max = (x > y) ? (x > z ? x : z) : (y > z ? y : z);
        System.out.println("Largest number is: " + max);
    }
}
```

Output:

```
Largest number is: 40
```

7. Bitwise and Shift Operators

Bitwise Operators work on bits directly.They perform operations such as AND (&), OR (|), XOR (^), and complement (~).**Shift Operators** (<<, >>, >>>) move bits left or right, effectively multiplying or dividing by powers of two.

```
// Demonstrating Bitwise and Shift Operators
public class BitwiseExample {
    public static void main(String[] args) {
        int a = 0b1011; // 11 in binary
        int b = 0b1101; // 13 in binary

        System.out.println("a & b : " + (a & b));
        System.out.println("a | b : " + (a | b));
        System.out.println("a ^ b : " + (a ^ b));
        System.out.println("~a : " + (~a));
        System.out.println("a << 2 : " + (a << 2));
        System.out.println("b >> 1 : " + (b >> 1));
        System.out.println("b >>> 1 : " + (b >>> 1));
    }
}
```

Output:

```
a & b : 9
a | b : 15
a ^ b : 6
~a : -12
a << 2 : 44
b >> 1 : 6
b >>> 1 : 6
```

8. instanceof Operator

The **instanceof operator** checks whether an object belongs to a specific class or implements a particular interface.It returns a **boolean** (true or false).

```
// Demonstrating instanceof Operator
public class InstanceofExample {
    public static void main(String[] args) {
        String message = "Java Rocks!";
        System.out.println(message instanceof String);

        Object obj = new Double(45.6);
        System.out.println(obj instanceof Double);
        System.out.println(obj instanceof String);
    }
}
```

Output:

true
true
false

1.5 Expressions and Control Statements

Expressions

In **Java**, an **expression** is a combination of **literals**, **variables**, **operators**, and **method invocations** that the compiler evaluates to produce a **single value**.

Expressions are one of the most fundamental building blocks of Java programs. They represent computations, decisions, and data manipulations that make up the logic of your code. Every expression, when evaluated, yields a result such as a number, a boolean value, a string, or even an object reference.

Components of a Java Expression

A Java expression can contain one or more of the following components:

a) Literals

Literals are fixed constant values written directly into the code. They represent data such as numbers, characters, strings, or boolean values.

Examples:

```
10          // integer literal
3.14        // floating-point literal
"Hello"     // string literal
true        // boolean literal
```

These values remain constant during program execution and are directly used by the compiler.

b) Variables

Variables are named storage locations in memory that hold data values. When a variable appears inside an expression, its **current value** is used for evaluation.

Example:

```
int x = 8;
int y = x + 2;   // variable x participates in expression
System.out.println(y);
```

Output:

10

Here, the expression x + 2 evaluates to 8 + 2 = 10.

c) Operators

Operators are special symbols that perform actions or operations on operands. They define how values and variables are manipulated.

Common Operator Types:

- **Arithmetic:** +, -, *, /, %
- **Relational:** ==, !=, <, >, <=, >=
- **Logical:** &&, ||, !
- **Assignment:** =, +=, -=, *=, /=

Example:

```
int a = 10, b = 5;
int result = a * b + 2;
System.out.println(result);
```

Output:

52

The expression a * b + 2 is evaluated using operator precedence.
First multiplication (10 * 5 = 50), then addition (50 + 2 = 52).

d) Method Invocations

In Java, **method calls that return values** can also be part of expressions. When such a method is invoked, the return value replaces the method call during evaluation.

Example:

```
String word = "Java";
```

```
int length = word.length();    // method invocation expression
System.out.println("Length: " + length);
```

Output:

Length: 4

Here, the method length() returns an integer (4), which becomes the value of the expression.

Types of Expressions in Java

Expressions can be categorized based on the kind of operation they perform or the type of value they return.

a) Literal Expression

A literal by itself is a simple expression since it directly represents a value.

Syntax:

```
10;       // integer literal expression
true;     // boolean literal expression
"Hello";  // string literal expression
```

Example:

```
public class LiteralExample {
    public static void main(String[] args) {
        System.out.println(10);       // integer literal
        System.out.println("Java");   // string literal
    }
}
```

Output:

```
10
Java
```

b) Arithmetic Expression

An **arithmetic expression** performs mathematical calculations using numeric operands.

Syntax:

operand1 operator operand2

Example:

```
public class ArithmeticExpression {
   public static void main(String[] args) {
      int a = 20;
      int b = 4;
      int result = a + b * 2;
      System.out.println("Result: " + result);
   }
}
```

Output:

Result: 28

Explanation: According to **operator precedence**, multiplication (b * 2 = 8) is performed first, then addition (20 + 8 = 28).

c) Relational Expression

A **relational expression** compares two values and produces a boolean (true or false).

Syntax:

operand1 relational_operator operand2

Example:

```
public class RelationalExpression {
   public static void main(String[] args) {
      int age = 21;
      boolean eligible = age >= 18;
      System.out.println("Eligible to vote: " + eligible);
   }
}
```

Output:

Eligible to vote: true

Explanation: The expression age >= 18 evaluates to true because 21 is greater than 18.

d) Logical Expression

A **logical expression** combines multiple conditions and returns a boolean result. These are widely used in decision-making statements like if and while.

Syntax:

condition1 logical_operator condition2

Example:

```
public class LogicalExpression {
    public static void main(String[] args) {
        int marks = 75;
        boolean pass = (marks >= 50) && (marks <= 100);
        System.out.println("Pass Status: " + pass);
    }
}
```

Output:

Pass Status: true

Explanation: Both conditions (marks >= 50) and (marks <= 100) are true, so the && (logical AND) operator returns true.

e) Method Invocation Expression

A **method invocation expression** calls a method and uses its return value as part of the computation.

Example:

```
public class MethodExpression {
    public static void main(String[] args) {
        String text = "Programming";
        int length = text.length();  // returns 11
        System.out.println("Length: " + length);
    }
}
```

Output:

Length: 11

Explanation: The length() method returns the number of characters in the string "Programming", which is 11.

f) Complex Expression

A **complex expression** combines literals, variables, operators, and method calls into one expression. It's evaluated according to Java's **operator precedence** and **associativity** rules.

Example:

```
public class ComplexExpression {
   public static void main(String[] args) {
      int a = 10, b = 5;
      String name = "Java";
      int result = (a + b) / 3 + name.length();
      System.out.println("Final Result: " + result);
   }
}
```

Output:

Final Result: 9

Explanation:

1. (a + b) → 15
2. (a + b) / 3 → 5
3. name.length() → 4
4. 5 + 4 → 9

So the final result of the complex expression is 9.

Evaluation of Expressions

Expressions in Java are evaluated based on **operator precedence** and **associativity** rules. This determines **the order in which operations are performed** when an expression has multiple operators.

For example:

```
int value = 5 + 3 * 2;
System.out.println(value);
```

Output:

11

Explanation: Multiplication (3 * 2 = 6) has higher precedence than addition, so it's evaluated first.

Then 5 + 6 = 11.

If you want to change the order of execution, use **parentheses**:

```
int value = (5 + 3) * 2;  // Parentheses take precedence
System.out.println(value);
```

Output:

16

Characteristics of Java Expressions

- Every expression **evaluates to a single value**.
- Expressions can be **nested** inside other expressions.
- **Type conversion (casting)** may occur automatically or explicitly during evaluation.
- They follow **strict data type rules** type mismatches cause compile-time errors.
- Expressions can have **side effects**, like modifying variables (e.g., x++)

Control Statements

Control statements are instructions that regulate the flow of a program's execution based on specific conditions. They are used to make decisions, repeatedly execute blocks of code, or transfer control to different parts of the program as needed. Control statements are an essential feature of Java and other programming languages, allowing developers to create flexible, logical, and interactive programs.

Types of Control Statements

1. Decision-Making Statements
2. Looping Statements
3. Jump Statements

Decision-Making Statements

Flowchart

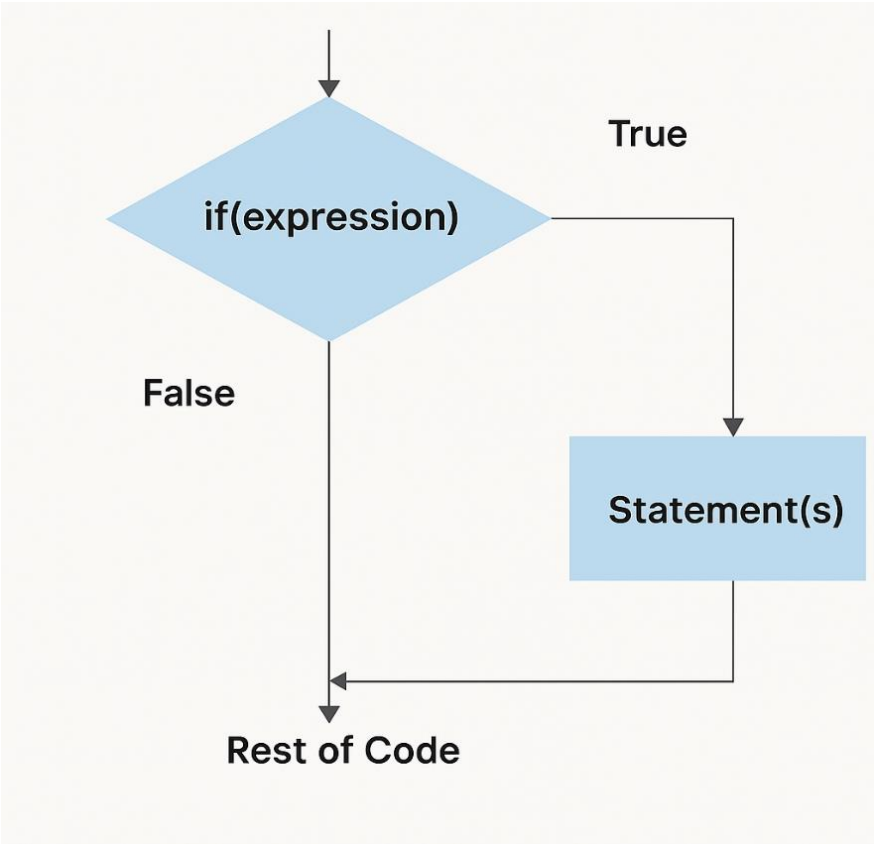


Fig. 1.4 Flow chart of if Statement.

Decision-making statements in Java are control structures that determine the flow of program execution based on specific conditions. They enable the program to choose different paths depending on whether a given condition (or set of conditions) evaluates to **true** or **false**.

In Java, the primary types of decision-making statements are:

- if statement
- if-else statement
- switch statement

'if' Statement

- The **if statement** in Java is used to evaluate a **boolean condition**.
- If the condition evaluates to **true**, the block of code inside the if statement is executed.

- ❖ It represents the simplest form of decision-making, allowing selective execution of code depending on the condition's result.

Syntax:

```
if (condition) {
   // Code to execute if the condition is true
}
```

Example: Online Shopping Discount System

A user makes a purchase worth a certain amount. The system checks if the user is eligible for a discount (for example, purchases above Rs.1000 get a 10% discount).

```
public class ShoppingDiscount {
   public static void main(String[] args) {
      int totalPurchase = 1200;  // Total purchase amount in Rupees
      double discount = 0.0;     // Discount value

      if (totalPurchase >= 1000) {
         discount = totalPurchase * 0.10; // 10% discount
         totalPurchase -= discount;
         System.out.println("Congratulations! You got a 10% discount.");
      }

      System.out.println("Final amount to pay: Rs. " + totalPurchase);
   }
}
```

Output

```
Congratulations! You got a 10% discount.
Final amount to pay: Rs. 1080.0
```

'if-else' Statement

The if-else statement in Java is used to handle two possible conditions in a program. It follows an if statement and provides an alternative set of actions when the if condition evaluates to false. If the if condition is true, the corresponding code block executes; otherwise, the code inside the else block runs.

Syntax:

```
if (condition) {
   // Code to execute when the condition is true
```

```
} else {
   // Code to execute when the condition is false
}
```

Flowchart

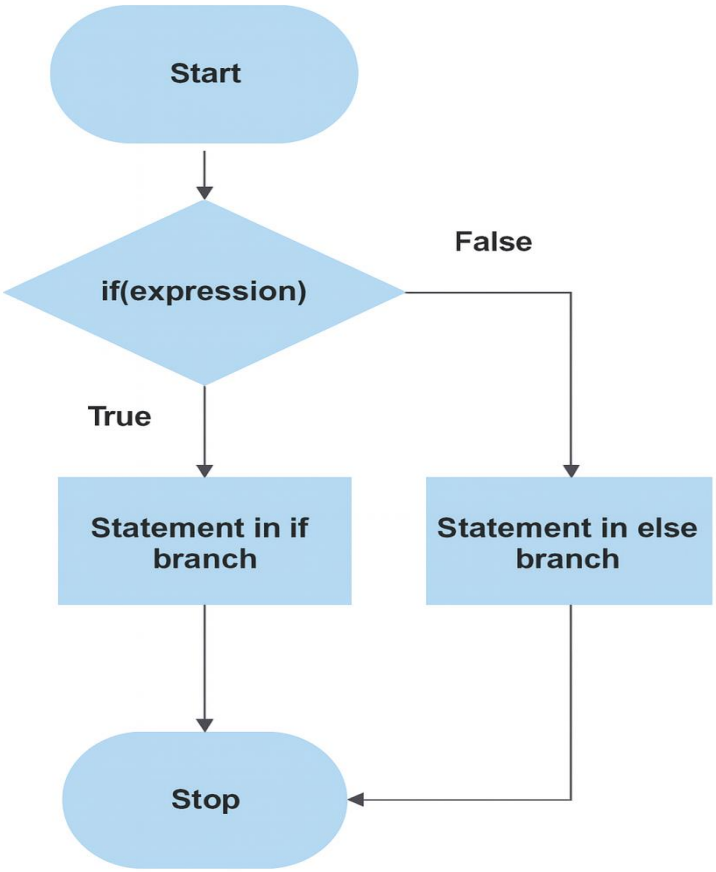


Fig. 1.5 Flowchart of if else statement.

Example: ATM Withdrawal System

A user tries to withdraw money from an ATM.

- If the account balance is greater than or equal to the withdrawal amount, the transaction is successful.
- If the balance is lower but still above Rs.100, the system suggests withdrawing a smaller amount.
- If the balance is too low, it notifies the user about insufficient funds.

```
public class ATMWithdrawal {
   public static void main(String[] args) {
      int accountBalance = 1500;   // User's account balance in Rupees
      int withdrawAmount = 2000;   // Desired withdrawal amount
      int smallerWithdraw = 500;   // Suggested smaller withdrawal amount

      if (accountBalance >= withdrawAmount) {
         accountBalance -= withdrawAmount;
         System.out.println("Withdrawal successful. Remaining balance: Rs. " +
accountBalance);
      } else if (accountBalance >= smallerWithdraw) {
         System.out.println("Insufficient balance for Rs. 2000 withdrawal. You can withdraw
Rs. 500 instead.");
      } else {
         System.out.println("Insufficient balance to perform any withdrawal.");
      }
   }
}
```

Output:

Insufficient balance for Rs. 2000 withdrawal. You can withdraw Rs. 500 instead.

'switch' Statement

The switch statement in Java is used to execute one block of code among multiple options based on the value of a variable or expression. It provides a cleaner and more structured alternative to using multiple if-else statements, making the code easier to read and maintain.

Syntax

```
switch (expression) {
   case value1:
      // Code to execute if expression equals value1
      break;
   case value2:
      // Code to execute if expression equals value2
      break;
   // ...
   default:
      // Code to execute if expression does not match any case
}
```

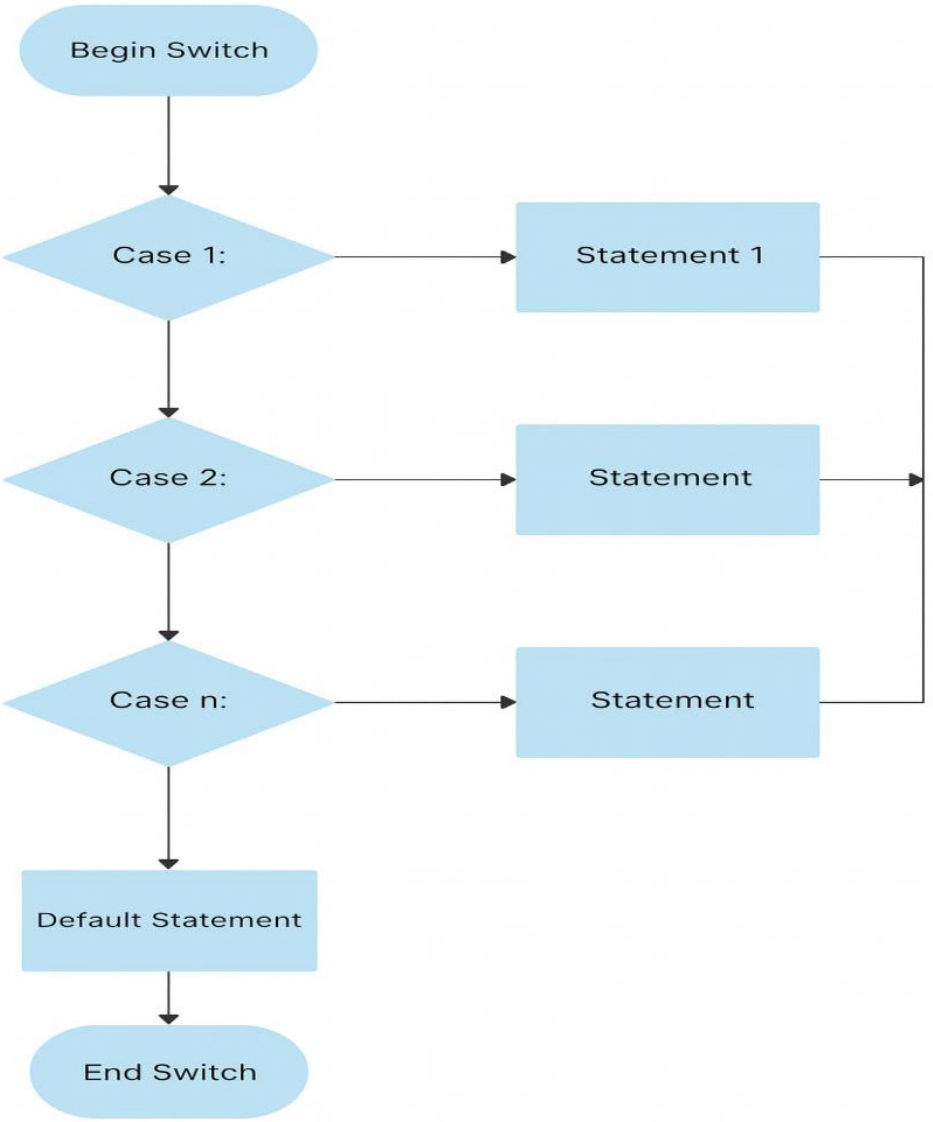


Fig. 1.6 Flowchart of Switch stement.

Example:Online Movie Ticket Booking System

A user selects a movie category Standard, Premium, or VIP. Each category has a different ticket price. The system calculates the total amount to be paid based on the selected category. (Here, category is set to "Premium".)

```
public class MovieTicketBooking {
    public static void main(String[] args) {
        String category = "Premium"; // User's selected movie category
        int ticketPrice;
```

```
        switch (category) {
            case "Standard":
                ticketPrice = 150; // Price for Standard category
                break;
            case "Premium":
                ticketPrice = 250; // Price for Premium category
                break;
            case "VIP":
                ticketPrice = 400; // Price for VIP category
                break;
            default:
                ticketPrice = 0; // Invalid category
                System.out.println("Invalid movie category selected.");
        }

        if (ticketPrice != 0) {
            System.out.println("Total ticket cost for " + category + " category: Rs. " + ticketPrice);
        }
    }
}
```

Output:

Total ticket cost for Premium category: Rs. 250

2. Looping Statements

Looping statements in Java are used to execute a block of code repeatedly as long as a specified condition is true. They are essential for performing repetitive tasks such as traversing arrays, processing data, and automating repeated actions.

The main types of looping statements in Java are:

- **for loop**
- **while loop**
- **do-while loop**

'for' Loop

Example: Calculating the Sum of Even Numbers

Let's write a program to calculate the sum of all even numbers between 1 and 50.

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

```
        int sum = 0;

        for (int i = 2; i <= 50; i += 2) {
            sum += i; // Add only even numbers
        }

        System.out.println("The sum of even numbers from 1 to 50 is: " + sum);
    }
}
```

Output:

The sum of even numbers from 1 to 50 is: 650

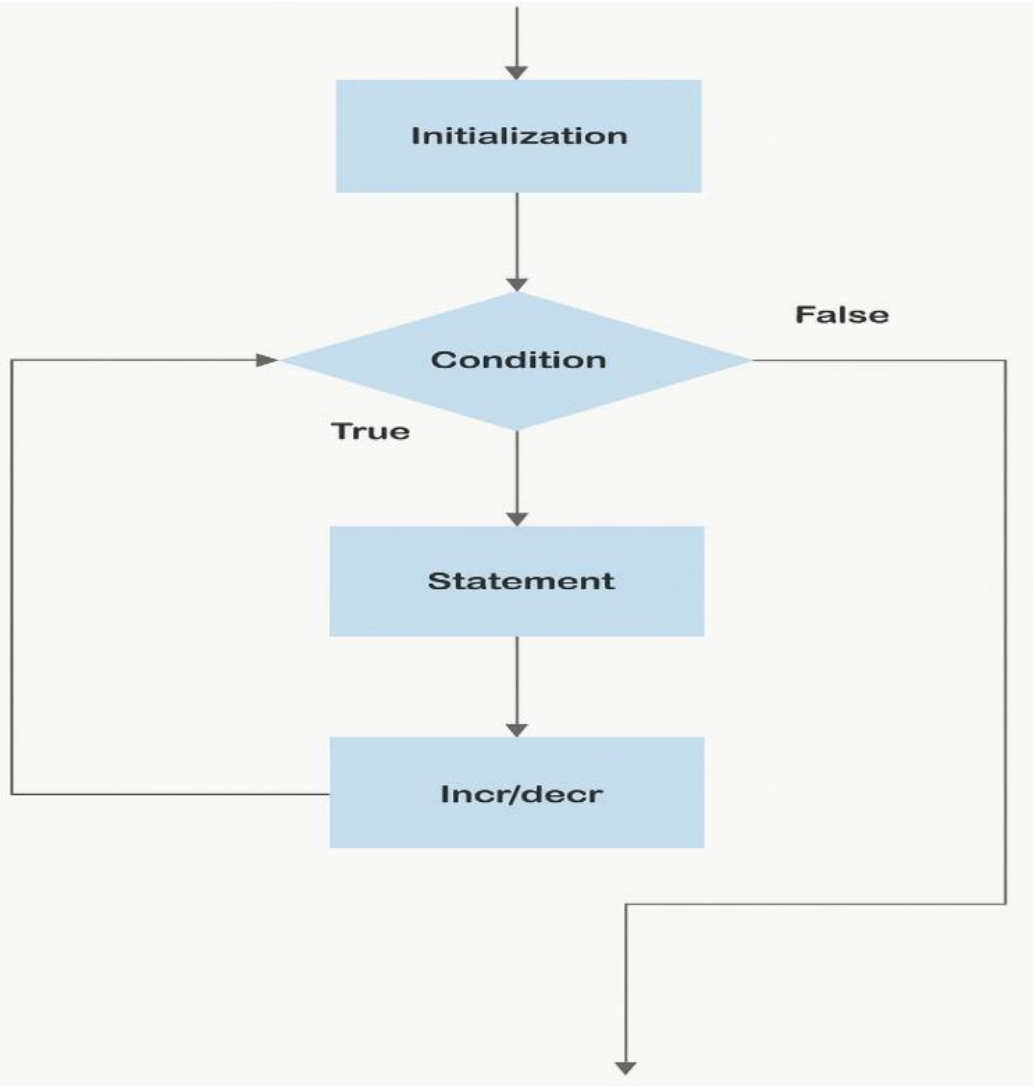


Fig. 1.7 Flow chart of for loop.

'while' Loop

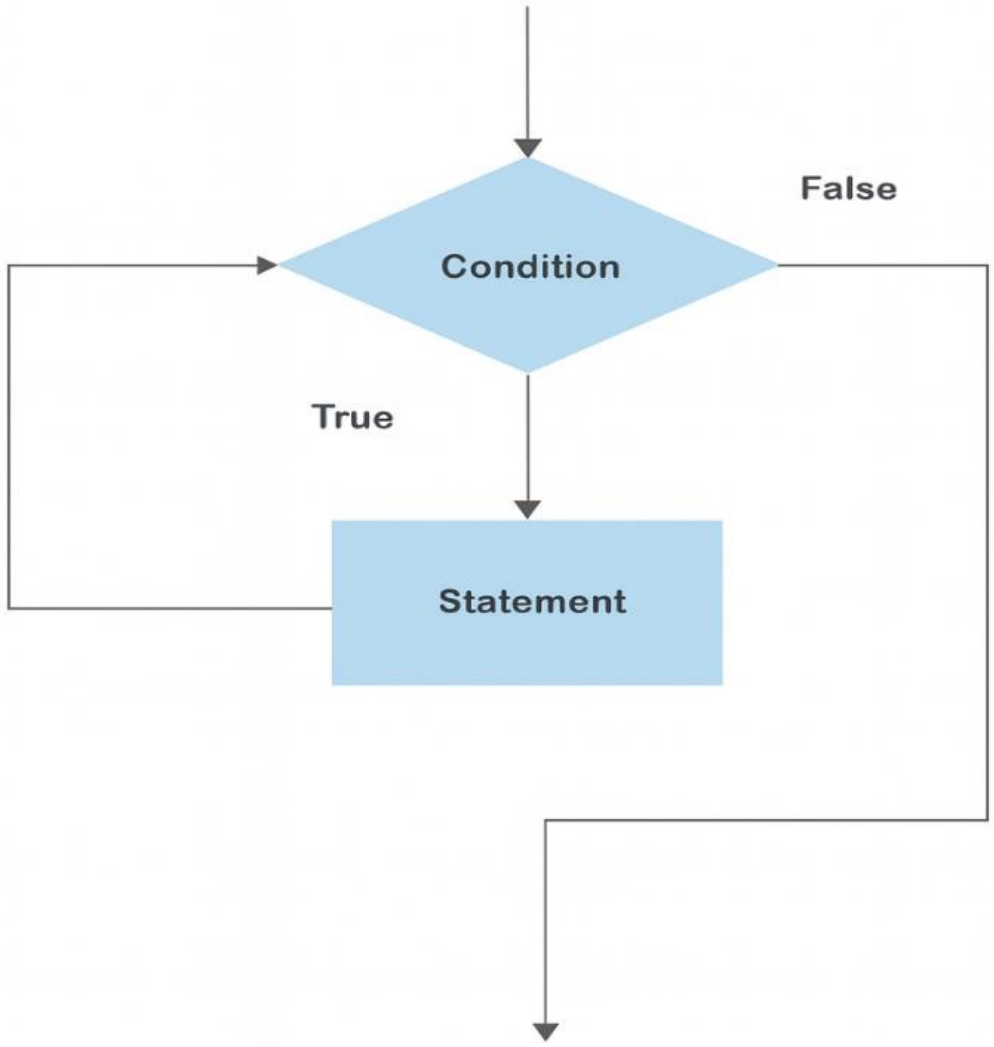


Fig. 1.8 Flowchart of While loop.

The **while loop** executes a block of code repeatedly as long as a specified condition remains true. It is especially useful when the number of iterations is not known in advance.

Syntax

```
while (condition) {
   // Code block to be executed
}
```

- **Condition:** The loop continues as long as this evaluates to true.
- **Code Block:** Executes repeatedly until the condition becomes false.

Example: Guess the Number Game

This program keeps asking the user to guess a number until they get it right.

```
import java.util.Scanner;

public class GuessTheNumber {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int secretNumber = 7;
        int guess = 0;

        System.out.println("Guess the secret number between 1 and 10!");

        while (guess != secretNumber) {
            System.out.print("Enter your guess: ");
            guess = scanner.nextInt();
            if (guess < secretNumber) {
                System.out.println("Too low! Try again.");
            } else if (guess > secretNumber) {
                System.out.println("Too high! Try again.");
            } else {
                System.out.println("Congratulations! You guessed the correct number.");
            }
        }

        scanner.close();
    }
}
```

Output:

```
Guess the secret number between 1 and 10!
Enter your guess: 5
Too low! Try again.
Enter your guess: 7
Congratulations! You guessed the correct number.
```

'do-while' Loop

The **do-while loop** is a **post-tested** loop. This means the loop body executes **at least once** before checking the condition. It's useful for menus or user-input-based programs where you must run the code once regardless of the condition.

Syntax

```
do {
   // Statements to execute
} while (condition);
```

- Executes the block first, then checks the condition.
- If the condition is true, it repeats; otherwise, it stops.

Flowchart

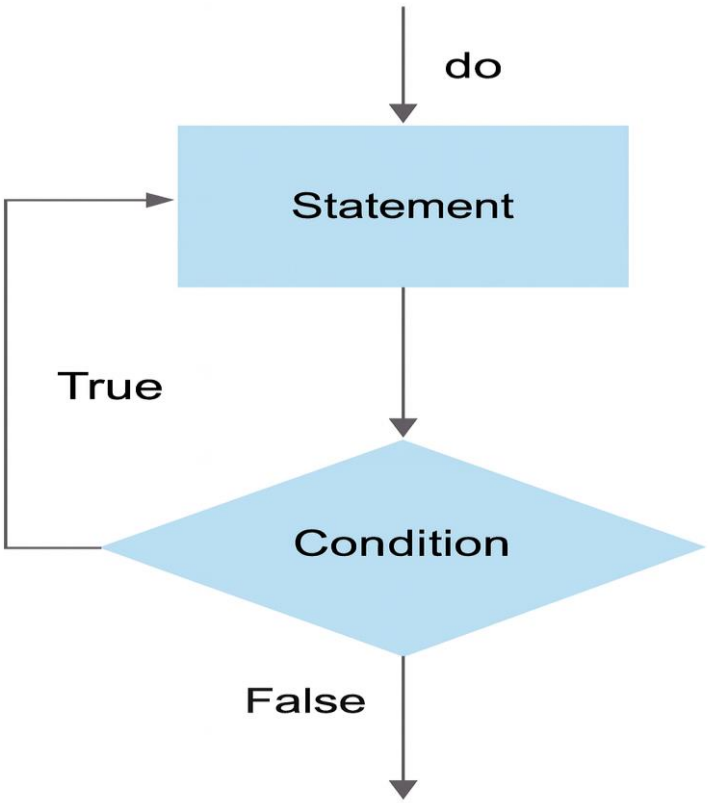


Fig. 1.9 Flowchart of do while loop.

Example: Simple Calculator Program

This program performs basic arithmetic operations until the user chooses to exit.

```
import java.util.Scanner;

public class SimpleCalculator {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
```

```
int choice;
double num1, num2, result;
do {
    System.out.println("\n=== Calculator Menu ===");
    System.out.println("1. Addition");
    System.out.println("2. Subtraction");
    System.out.println("3. Multiplication");
    System.out.println("4. Division");
    System.out.println("5. Exit");
    System.out.print("Enter your choice: ");
    choice = scanner.nextInt();

    if (choice >= 1 && choice <= 4) {
        System.out.print("Enter first number: ");
        num1 = scanner.nextDouble();
        System.out.print("Enter second number: ");
        num2 = scanner.nextDouble();

        switch (choice) {
            case 1:
                result = num1 + num2;
                System.out.println("Result: " + result);
                break;
            case 2:
                result = num1 - num2;
                System.out.println("Result: " + result);
                break;
            case 3:
                result = num1 * num2;
                System.out.println("Result: " + result);
                break;
            case 4:
                if (num2 != 0)
                    result = num1 / num2;
                else {
                    System.out.println("Error: Division by zero!");
                    break;
                }
                System.out.println("Result: " + result);
                break;
        }
    } else if (choice != 5) {
        System.out.println("Invalid choice. Please try again.");
    }
```

```
        } while (choice != 5);

        System.out.println("Exiting the calculator...");
        scanner.close();
    }
}
```

Output:

Calculator Menu
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit
Enter your choice: 1
Enter first number: 5
Enter second number: 3
Result: 8.0

Calculator Menu
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit
Enter your choice: 5
Exiting the calculator...

3. Jump Statements

Jump statements in Java are used to **transfer control** to another part of the program. They help in managing loop flow, skipping iterations, or exiting methods.

The three main types of jump statements are:

1. **break**
2. **continue**
3. **return**

'break' Statement

The **break statement** terminates a loop or switch statement immediately and transfers control to the next statement after the loop/switch.

Syntax

break;

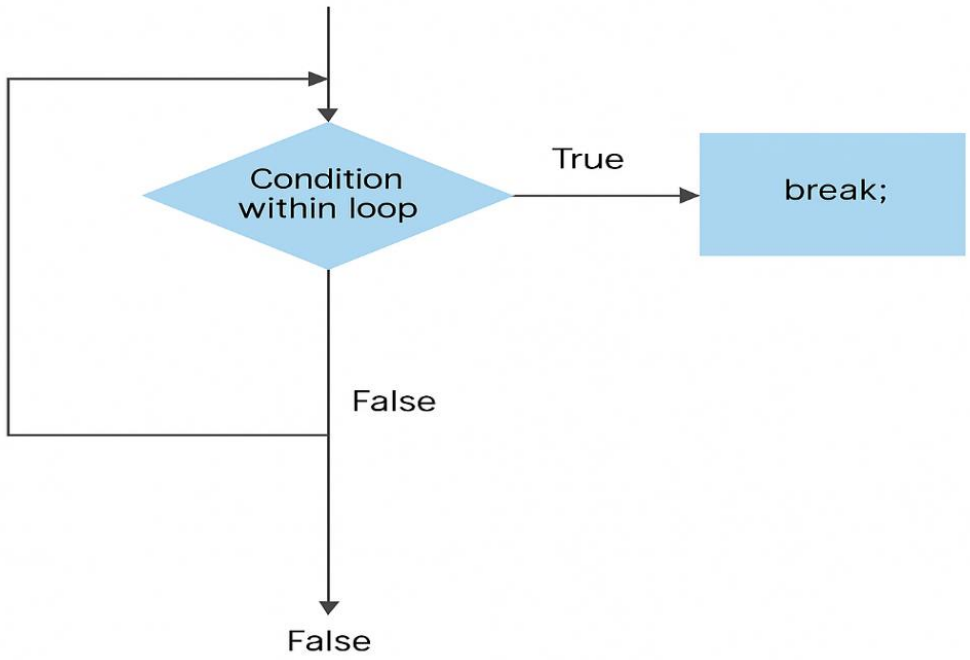


Fig. 1.10 Flowchart of Break statement.

Example: Stop Printing When a Specific Word is Found

```
public class StopAtWord {
    public static void main(String[] args) {
        String[] words = {"apple", "banana", "grape", "stop", "mango"};

        for (String word : words) {
            if (word.equals("stop")) {
                System.out.println("Stopping at the word 'stop'.");
                break;
            }
            System.out.println(word);
        }
    }
}
```

Output:

apple
banana
grape
Stopping at the word 'stop'.

'continue' Statement

The **continue statement** skips the current iteration of a loop and proceeds with the next one. It's often used when certain conditions need to be ignored.

Syntax

continue;

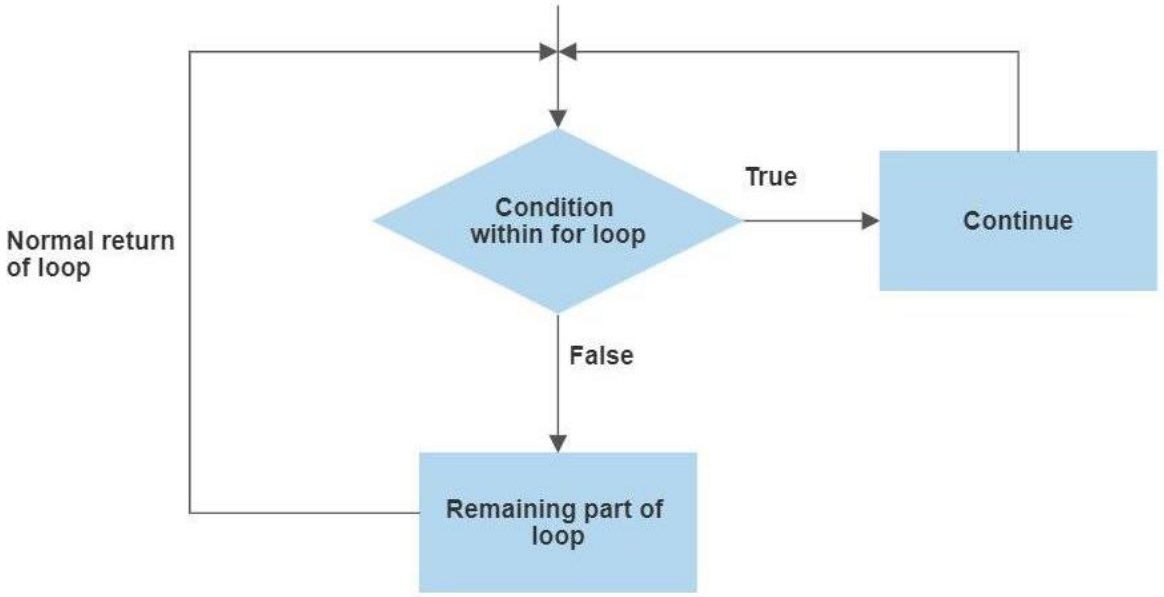


Fig. 1.11 Flowchart of Control Statement.

Example: Print Odd Numbers Only

```
public class PrintOddNumbers {
   public static void main(String[] args) {
      for (int i = 1; i <= 20; i++) {
         if (i % 2 == 0) {
            continue; // Skip even numbers
         }
         System.out.println(i);
      }
```

```
    }
}
```

Output:

```
1
3
5
7
9
11
13
15
17
19
```

1.6 Classes , Objects and Constructors

Classes

A **class** is a **blueprint** or **template** from which individual **objects** are created. It defines the **data** (attributes) and **behavior** (methods) that its objects will have. In Java, everything revolves around **classes and objects**.

For instance, if you want to represent cars in a program, the class will be **Car**, and each car (like car1, car2, etc.) will be an **object** of that class.

Properties of Java Classes

- A **class** does not take up any memory space until an object is created.
- It acts as a **blueprint** for real-world entities but is **not** itself a real-world object.
- A class mainly contains **data members** (variables) and **methods** (functions).
- Classes can also be **nested** inside other classes.
- Classes follow all **OOP principles** such as **inheritance**, **encapsulation**, and **abstraction**.

Types of Class Variables

A class can contain three main types of variables:

1. Local Variables

Defined **inside methods, constructors, or blocks**. They are created when the method starts and destroyed when the method ends.

2. Instance Variables

Defined **inside a class but outside any method**. They are created when the class is instantiated (object created).

3. Class Variables (Static Variables)

Declared **inside the class but outside any method**, and marked with the static keyword. They are shared by all objects of the class.

Creating (Declaring) a Java Class

You can declare a class using an **access modifier**, followed by the class keyword and the **class name**.

Syntax:

```
access_modifier class ClassName {
   data members;
   constructors;
   methods;
}
```

Example of a Java Class

Let's create a class named **Car**.
The class attributes are brand, year, and color.
The methods are setBrand(), setYear(), setColor(), and displayDetails().

```
// Creating a Java class
class Car {
   // Declaring and initializing attributes
   String brand;
   int year;
   String color;
```

```
    // Methods to set brand, year, and color
    public void setBrand(String brand) {
        this.brand = brand;
    }

    public void setYear(int year) {
        this.year = year;
    }
    public void setColor(String color) {
        this.color = color;
    }

    // Method to display car details
    public void displayDetails() {
        System.out.println("Car Details:");
        System.out.println("Brand: " + this.brand);
        System.out.println("Year: " + this.year);
        System.out.println("Color: " + this.color);
    }
}
```

Objects

An **object** is an **instance** of a class.It represents a **real-world entity** that has **state** (data) and **behavior** (methods).

For example, a Car object's **state** could be its brand, color, and year, while its **behavior** could be starting, driving, or stopping.

Creating (Declaring) a Java Object

A class provides the **blueprint**, and an object is created from it using the new keyword.

Three steps for object creation:

1. **Declaration** – Defining a variable with an object type.
2. **Instantiation** – Using new to create the object.
3. **Initialization** – Calling a constructor to set up the object.

Syntax:

ClassName objectName = new ClassName([parameters]);

Example: Creating and Using an Object

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of Car class
        Car myCar = new Car();

        // Setting attributes
        myCar.setBrand("Tesla");
        myCar.setYear(2023);
        myCar.setColor("Red");
        // Displaying details
        myCar.displayDetails();
    }
}
```

Output:

```
Car Details:
Brand: Tesla
Year: 2023
Color: Red
```

Accessing Instance Variables and Methods

Instance variables and methods are accessed through objects.

Syntax:

```
// Create an object
ObjectReference = new ClassName();

// Access instance variable
ObjectReference.variableName;

// Access instance method
ObjectReference.methodName();
```

Example:

Here's another example using a **Book** class.

```
public class Book {
    int pages;
```

```
    // Constructor
    public Book(String title) {
        System.out.println("Book title: " + title);
    }
    // Setter method
    public void setPages(int pages) {
        this.pages = pages;
    }
    // Getter method
    public int getPages() {
        System.out.println("Number of pages: " + pages);
        return pages;
    }

    public static void main(String[] args) {
        // Object creation
        Book myBook = new Book("Java Programming");

        // Set and get page count
        myBook.setPages(450);
        myBook.getPages();

        // Access variable directly
        System.out.println("Direct access: " + myBook.pages);
    }
}
```

Output:

```
Book title: Java Programming
Number of pages: 450
Direct access: 450
```

Rules for Using Classes and Objects

1. Only **one public class** can exist per source file.
2. The **filename** must match the **public class name**, followed by .java.
   Example: public class Student {} - File name: Student.java
3. Multiple **non-public classes** can exist in one file.
4. If a **package** is used, it must appear as the **first statement** in the file.
5. **Import statements** (if any) must appear **after** the package declaration.
6. Import and package statements apply to **all classes** in the file.
7. Classes can be **abstract**, **final**, or **nested**, depending on usage.

Another Example: Laptop Class

```
class Laptop {
    String brand;
    int ram;
    double price;

    // Constructor
    public Laptop(String brand, int ram, double price) {
        this.brand = brand;
        this.ram = ram;
        this.price = price;
    }

    // Display details
    public void showDetails() {
        System.out.println("Laptop Details:");
        System.out.println("Brand: " + brand);
        System.out.println("RAM: " + ram + "GB");
        System.out.println("Price: Rs. " + price);
    }

    public static void main(String[] args) {
        Laptop l1 = new Laptop("HP", 16, 85000);
        l1.showDetails();
    }
}
```

Output:

```
Laptop Details:
Brand: HP
RAM: 16GB
Price: Rs. 85000
```

Constructors

Constructors are **special methods** used to **initialize an object** when it is created. A constructor has the **same name as the class** and looks similar to a method, but it **does not have any return type**, not even void.

Typically, constructors are used to **set initial values** for instance variables or perform any **setup tasks** needed when an object is first created.

Every class in Java has at least one constructor. If you do not explicitly define one, Java automatically provides a **default constructor** that initializes all instance variables to their default values (like 0, null, or false). However, once you create your own constructor, the default one is no longer provided automatically.

Rules for Creating Java Constructors

When defining constructors in Java, keep these rules in mind:

1. The constructor's **name must match** the class name.
2. Constructors **do not have a return type** (not even void).
3. You can define **multiple constructors** with different parameter lists this is called **constructor overloading**.
4. You can use **access modifiers** (public, private, etc.) to control the visibility of constructors.
5. If no constructor is explicitly defined, Java automatically provides a **default constructor**.

Creating a Java Constructor

To create a constructor, write the class name followed by parentheses () and define the body inside curly braces {}.

Syntax:

```
class ClassName {
   ClassName() {
      // Constructor body
   }
}
```

Example: Simple Constructor

Here's a simple example where a constructor prints a message when an object is created:

```
public class Greeting {
  // Constructor
  Greeting() {
    System.out.println("Welcome to Java Programming!");
  }

  public static void main(String[] args) {
    System.out.println("Program started...");
```

```
 // Creating an object - constructor is automatically called
    Greeting message = new Greeting();
  }
}
```

Output:

```
Program started...
Welcome to Java Programming!
```

Types of Constructors in Java

Java provides **three main types** of constructors:

1. **Default Constructor**
2. **No-Argument Constructor**
3. **Parameterized Constructor**

1. Default Constructor

If you don't define any constructor in your class, Java automatically provides a **default constructor** that initializes all variables with default values.

Example: Default Constructor

```
public class Car {
  String brand;
  int year;

  public static void main(String[] args) {
    // Default constructor will be invoked automatically
    Car myCar = new Car();

    System.out.println("Brand: " + myCar.brand);
    System.out.println("Year: " + myCar.year);
  }
}
```

Output:

```
Brand: null
Year: 0
```

Here, since we didn't define any constructor, Java automatically created a default one and assigned the default values (null for String and 0 for int).

2. No-Argument Constructor

A **no-argument constructor** (or **no-args constructor**) is one that does not take any parameters. You can define it manually to assign initial values or perform setup tasks when an object is created.

Example: No-Args Constructor

```
public class Book {
  String title;
  double price;

  // No-argument constructor
  Book() {
    title = "Unknown";
    price = 0.0;
  }

  public static void main(String[] args) {
    // The no-args constructor is called automatically
    Book myBook = new Book();

    System.out.println("Book Title: " + myBook.title);
    System.out.println("Book Price: Rs. " + myBook.price);
  }
}
```

Output:

```
Book Title: Unknown
Book Price: Rs. 0.0
```

3. Parameterized Constructor

A **parameterized constructor** is used when you want to initialize an object with specific values at the time of creation. Parameters are added just like in methods.

Example 1: Parameterized Constructor

```
public class Book {
  String title;
  double price;

  // Parameterized constructor
  Book(String t, double p) {
```

```
    title = t;
    price = p;
  }

  public static void main(String[] args) {
    // Creating objects and initializing them with values
    Book b1 = new Book("Java Basics", 499.99);
    Book b2 = new Book("Advanced Java", 799.50);
    System.out.println("Book 1: " + b1.title + " - Rs. " + b1.price);
    System.out.println("Book 2: " + b2.title + " - Rs. " + b2.price);
  }
}
```

Output:

```
Book 1: Java Basics - Rs. 499.99
Book 2: Advanced Java - Rs. 799.5
```

Example 2: Parameterized Constructor

```
class Laptop {
   String brand;
   int ram;

   // Constructor with parameters
   Laptop(String b, int r) {
      brand = b;
      ram = r;
   }
}

public class Demo {
   public static void main(String[] args) {
      Laptop l1 = new Laptop("Dell", 8);
      Laptop l2 = new Laptop("HP", 16);

      System.out.println(l1.brand + " - " + l1.ram + "GB");
      System.out.println(l2.brand + " - " + l2.ram + "GB");
   }
}
```

Output:

```
Dell - 8GB
HP - 16GB
```

Constructor Overloading

Constructor overloading means having **multiple constructors** within a class, each with different parameter lists. Java determines which constructor to invoke based on the number and type of arguments passed during object creation.

Example: Constructor Overloading

```
// Example of Constructor Overloading
class Employee {
  String name;
  int age;
  String department;

  // No-args constructor
  Employee() {
    this.name = "Not Assigned";
    this.age = 0;
    this.department = "General";
  }

  // Constructor with one parameter
  Employee(String name) {
    this.name = name;
    this.age = 0;
    this.department = "General";
  }

  // Constructor with two parameters
  Employee(String name, int age) {
    this.name = name;
    this.age = age;
    this.department = "General";
  }

  // Constructor with three parameters
  Employee(String name, int age, String department) {
    this.name = name;
    this.age = age;
    this.department = department;
  }

  public void displayDetails() {
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
```

```
    System.out.println("Department: " + department);
    System.out.println();
  }
}

public class Main {
  public static void main(String[] args) {
    Employee e1 = new Employee();                    // invokes no-args constructor
    Employee e2 = new Employee("Priya");             // invokes single-arg constructor
    Employee e3 = new Employee("Ravi", 28);          // invokes two-arg constructor
    Employee e4 = new Employee("Kiran", 35, "HR");   // invokes three-arg constructor

    System.out.println("Employee 1:");
    e1.displayDetails();

    System.out.println("Employee 2:");
    e2.displayDetails();

    System.out.println("Employee 3:");
    e3.displayDetails();

    System.out.println("Employee 4:");
    e4.displayDetails();
  }
}
```

Output:

Employee 1:
Name: Not Assigned
Age: 0
Department: General

Employee 2:
Name: Priya
Age: 0
Department: General

Employee 3:
Name: Ravi
Age: 28
Department: General

Employee 4:
Name: Kiran

Age: 35
Department: HR

Access Control

Access control in Java is a fundamental concept that defines **how and where data members, methods, and classes can be accessed** within a program. It is one of the core principles of **object-oriented programming (OOP)**, supporting **encapsulation** the practice of keeping internal data hidden from outside interference and misuse.

Through access control, Java allows developers to restrict or permit access to certain parts of a program, thus improving code security, reliability, and maintainability. It ensures that only authorized code can modify or view the internal workings of a class, protecting the program from accidental or malicious errors.

Purpose of Access Control

The main goal of access control in Java is to **protect data integrity** and **encourage modular programming**. By limiting access to variables and methods, a class can prevent other classes from directly manipulating its internal data. This separation helps in maintaining **data consistency**, as only specific methods can alter the class's state.

Additionally, access control promotes **code reusability and clarity**, as developers can define clear boundaries between what is meant to be publicly used (e.g., APIs) and what is meant to remain internal (e.g., helper methods or implementation details). This structure makes large programs easier to understand and maintain.

How Access Control Works

Access control in Java is achieved through a set of **rules and mechanisms** that determine where a class or its members (fields, methods, or constructors) can be accessed from. These rules are enforced at **compile time** and are based on the **scope** of the element that is, whether it is being accessed within the same class, the same package, a subclass, or from a completely different package.

The **Java compiler** checks these rules whenever code tries to access a class or a class member. If access is not permitted according to Java's visibility rules, the compiler throws an **"illegal access"** error. This ensures that the developer cannot bypass visibility restrictions, thereby maintaining strict data protection.

Role of Access Modifiers in Access Control

Although access control is a concept, it is implemented in Java using **access modifiers** such as private, default (no modifier), protected, and public. These keywords define **different levels of visibility** for classes, variables, and methods.

For example:

- private restricts access to within the same class.
- default allows access within the same package.
- protected permits access to the same package and to subclasses.
- public allows access from anywhere in the program.

Thus, access modifiers act as **tools** for implementing access control, while the **access control system** ensures that those restrictions are followed consistently.

Levels of Access Control

Java defines several **levels of access control** that determine how widely a member can be used within a program. These levels include:

1. **Class-level access** – Determines which classes can be seen or used by other classes.
2. **Member-level access** – Determines which methods and variables of a class can be used by other classes or objects.

At the **class level**, only two types of access are allowed public (accessible everywhere) and default (accessible only within the same package).

At the **member level**, all four types of access control (private, default, protected, public) can be used to fine-tune visibility.

Importance of Access Control in Encapsulation

Encapsulation is the principle of **bundling data and behavior together** while restricting access to internal details. Access control plays a vital role in achieving this principle by allowing a class to expose only the information that is necessary.

For instance, a class might have private data members and public methods to modify or retrieve them. This ensures that the internal state of an object cannot be changed arbitrarily, but only through controlled operations. In this way, access control not only enhances data security but also encourages **clean, modular design**.

Advantages of Access Control

1. **Data Security:** Access control prevents unauthorized access and modification of private data, protecting the integrity of objects.
2. **Encapsulation:** It supports data hiding, ensuring that only relevant data and methods are exposed to the user.
3. **Code Maintenance:** By separating public interfaces from internal details, programs become easier to understand, update, and debug.
4. **Error Reduction:** It minimizes accidental interference between unrelated parts of code.
5. **API Design:** Access control allows developers to clearly define which parts of a program are intended for public use and which are internal.

1.7 Method Overloading

Method overloading in Java is a feature that allows a class to have **multiple methods with the same name but different parameter lists**. It is one of the ways Java implements **compile-time polymorphism** (also known as **static polymorphism**).

This means that the method to be executed is determined by the **compiler** at compile time, based on the **number, type, and order of arguments** passed during the method call.

Method overloading makes the code **more readable, flexible, and maintainable**, as the same method name can be reused for similar types of tasks that differ only in their input parameters.

Key Characteristics of Method Overloading

1. **Same method name, different parameters:** Overloaded methods must have the same name but a different number, type, or order of parameters.
2. **Return type alone doesn't matter:** You cannot overload methods by changing only the return type there must be a difference in the parameter list.
3. **Compile-time decision:** The Java compiler determines which version of the method to call based on the arguments provided.
4. **Increased readability:** Overloading allows developers to use one method name for conceptually similar operations.

Ways to Achieve Method Overloading

Method overloading can be achieved in three main ways:

1. Changing the **number of parameters**
2. Changing the **data type of parameters**

3. Changing the **order of parameters**

1. Changing the Number of Parameters

One of the most common ways to achieve method overloading is by defining multiple methods with the **same name** but a **different number of parameters**.

In this approach, the compiler distinguishes between methods by the **count of arguments** passed during the method call.

Example:

```
public class Calculator {

    // Method with two parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calling method with two arguments
        System.out.println("Sum of two numbers: " + calc.add(10, 20));

        // Calling method with three arguments
        System.out.println("Sum of three numbers: " + calc.add(5, 10, 15));
    }
}
```

Output:

```
Sum of two numbers: 30
Sum of three numbers: 30
```

Explanation: Both methods are named add(), but the compiler differentiates between them based on the number of parameters passed. When two arguments are given, the first method runs; when three are given, the second one runs.

2. Changing the Data Type of Parameters

Method overloading can also be achieved by changing the **data types** of the parameters in the method definition.

This allows a single method name to handle various types of inputs for example, integers, doubles, or strings depending on what the user passes.

Example:

```
public class Display {

    // Method with integer parameter
    public void show(int number) {
        System.out.println("Integer value: " + number);
    }

    // Overloaded method with string parameter
    public void show(String message) {
        System.out.println("String message: " + message);
    }

    // Overloaded method with double parameter
    public void show(double value) {
        System.out.println("Double value: " + value);
    }
}

public class Main {
    public static void main(String[] args) {
        Display obj = new Display();

        obj.show(25);              // Calls method with int
        obj.show("Hello Java");    // Calls method with String
        obj.show(19.99);           // Calls method with double
    }
}
```

Output:

```
Integer value: 25
String message: Hello Java
Double value: 19.99
```

Explanation: All three methods have the same name show(), but each handles a different data type. The compiler automatically calls the method whose parameter type matches the data type of the argument passed.

3. Changing the Order of Parameters

Method overloading can also occur when methods have the same name and the same number of parameters but **the order of parameter types differs**.

This technique is useful when you need to pass mixed data types in different sequences.

Example:

```
public class Person {

    // Method with String followed by int
    public void details(String name, int age) {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    // Overloaded method with int followed by String
    public void details(int age, String name) {
        System.out.println("Age: " + age + ", Name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person();

        p.details("Riya", 22);   // Calls first method
        p.details(25, "Karan");  // Calls second method
    }
}
```

Output:

```
Name: Riya, Age: 22
Age: 25, Name: Karan
```

Explanation: Here, both methods are named details() and have the same number of parameters but in different orders. The compiler identifies which one to call based on the sequence of the arguments provided.

Rules for Method Overloading

1. **Same name:** All overloaded methods must have the same method name.
2. **Different parameters:** Methods must differ in the number, type, or order of parameters.
3. **Return type does not matter:** Changing the return type alone will not constitute overloading.
4. **Compile-time binding:** The method call is resolved by the compiler, not at runtime.
5. **Access modifiers:** Overloaded methods can have different access modifiers (e.g., one public, another private), though it's not common.

Advantages of Method Overloading

1. **Improves code readability:** You can use the same method name for similar operations.
2. **Enhances reusability:** The same method name can work with different data types or parameter lists.
3. **Simplifies maintenance:** Reduces the number of method names you need to remember.
4. **Supports compile-time polymorphism:** The compiler decides which method to invoke, ensuring efficient execution.

Real-World Example: Method Overloading in Action

Consider an application that calculates **area** of different shapes a circle, a rectangle, and a triangle. Each method performs a similar operation (calculating area) but requires different parameters.

Example:

```
public class Shape {

   // Area of a circle
   public double area(double radius) {
      return 3.14 * radius * radius;
   }

   // Area of a rectangle
   public double area(double length, double width) {
      return length * width;
   }

   // Area of a triangle
   public double area(double base, int height) {
      return 0.5 * base * height;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Shape s = new Shape();

        System.out.println("Area of Circle: " + s.area(5.0));
        System.out.println("Area of Rectangle: " + s.area(4.0, 6.0));
        System.out.println("Area of Triangle: " + s.area(8.0, 5));
    }
}
```

Output:

```
Area of Circle: 78.5
Area of Rectangle: 24.0
Area of Triangle: 20.0
```

1.8 Static members, Arrays and Strings

Static members

In Java, the keyword **static** is used to define members (variables, methods, blocks, or nested classes) that belong to the **class itself**, rather than to any specific object of that class. This means that **static members are shared among all objects** of the class, instead of being created separately for each object.

The concept of static members helps in saving memory and allows accessing class-level data or behavior without creating an instance of the class. Static members are loaded into memory when the class is first loaded by the Java Virtual Machine (JVM), before any objects are created.

Why Use Static Members?

Static members are mainly used when data or behavior should be **common to all instances** of a class. For example, if multiple objects share the same constant value or need to perform the same utility operation, declaring those members as static prevents duplication and ensures consistency.

Types of Static Members

Java allows the use of the static keyword with four kinds of members:

1. **Static Variables (Class Variables)**
2. **Static Methods**
3. **Static Blocks**
4. **Static Nested Classes**

1. Static Variables

A **static variable** (also called a **class variable**) is declared with the static keyword inside a class but outside any method or constructor. Unlike instance variables that belong to individual objects, a static variable is **shared across all instances** of a class.

There is **only one copy** of a static variable, and it is stored in the class memory area.

Example:

```
public class Student {
    // static variable
    static String schoolName = "Sunrise Public School";

    // instance variable
    String studentName;

    // constructor
    Student(String name) {
        studentName = name;
    }

    public void showDetails() {
        System.out.println("Student Name: " + studentName);
        System.out.println("School Name: " + schoolName);
    }

    public static void main(String[] args) {
        Student s1 = new Student("Ravi");
        Student s2 = new Student("Anjali");

        s1.showDetails();
        s2.showDetails();
    }
}
```

Output:

Student Name: Ravi
School Name: Sunrise Public School
Student Name: Anjali
School Name: Sunrise Public School

Explanation: The variable schoolName is static, meaning both s1 and s2 share the same copy. If one object modifies it, the change reflects in all objects of the class.

2. Static Methods

A **static method** is a method that belongs to the class rather than an instance. It can be called directly using the **class name**, without creating an object.

Static methods can access only **static variables** and **other static methods** directly. They **cannot access instance variables or instance methods**, since they belong to the class and not to any specific object.

Example:

```
public class MathOperations {
    // static method
    public static int square(int number) {
        return number * number;
    }

    // non-static method
    public void displayMessage() {
        System.out.println("Non-static method called");
    }

    public static void main(String[] args) {
        // calling static method directly using class name
        int result = MathOperations.square(5);
        System.out.println("Square: " + result);

        // To call a non-static method, we need to create an object
        MathOperations obj = new MathOperations();
        obj.displayMessage();
    }
}
```

Output:

Square: 25
Non-static method called

Explanation: The method square() is static, so it can be invoked without creating an object. However, the non-static method displayMessage() requires an instance to be called.

3. Static Blocks

A **static block** in Java is used for **initializing static variables** or performing actions that need to be executed only once when the class is first loaded.

Static blocks run **before the main method** and before any object of the class is created. You can have multiple static blocks in a class, and they execute in the order they appear.

Example:

```
public class StaticBlockExample {
    static int count;

    // static block
    static {
        System.out.println("Static block executed.");
        count = 10;
    }

    public static void main(String[] args) {
        System.out.println("Main method executed.");
        System.out.println("Count value: " + count);
    }
}
```

Output:

Static block executed.
Main method executed.
Count value: 10

Explanation: The static block runs as soon as the class is loaded into memory, even before the main() method is called. It is typically used for setting up static resources.

4. Static Nested Classes

A **static nested class** is a nested (inner) class declared as static inside another class. Unlike regular inner classes, a static nested class does **not require an instance** of the outer class to be created.

It behaves like a normal top-level class but is logically grouped inside another class for better organization.

Example:

```
public class OuterClass {
    private static String message = "Hello from Outer Class";

    // static nested class
    static class NestedClass {
        public void display() {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        // creating object of static nested class
        OuterClass.NestedClass nested = new OuterClass.NestedClass();
        nested.display();
    }
}
```

Output:

Hello from Outer Class

Explanation: Since NestedClass is static, it can directly access the static variable message of OuterClass. We can create its object without creating an instance of the outer class.

Common Use Cases of Static Members

1. **Utility or Helper Methods** - Classes like Math or Collections use static methods (Math.pow(), Math.max(), etc.).
2. **Constants Declaration** - Use static final variables to define constants shared across all objects.
3. public static final double PI = 3.14159;
4. **Counters** -Use static variables to count the number of objects created.
5. static int objectCount = 0;

6. **Factory Methods** - Create static methods that return instances of a class (e.g., LocalDate.now()).

Example: Static Variable and Method Combined

```
public class Employee {
    private int id;
    private String name;
    private static String company = "TechCorp";
    private static int count = 0;

    // constructor
    public Employee(String name) {
        this.name = name;
        this.id = ++count;  // increment static counter
    }

    // static method
    public static void showCompany() {
        System.out.println("Company Name: " + company);
    }

    public void showDetails() {
        System.out.println("Employee ID: " + id + ", Name: " + name);
    }

    public static void main(String[] args) {
        Employee.showCompany();

        Employee e1 = new Employee("Ravi");
        Employee e2 = new Employee("Sneha");

        e1.showDetails();
        e2.showDetails();
    }
}
```

Output:

```
Company Name: TechCorp
Employee ID: 1, Name: Ravi
Employee ID: 2, Name: Sneha
```

Explanation: Here, company and count are static members shared by all employees. Each time a new employee is created, count is incremented once for the entire class.

Arrays

An **array** is a **data structure** that allows you to store a **fixed-size collection of elements** of the **same data type**. It provides a convenient way to group related data together instead of declaring multiple individual variables.

For example, instead of declaring separate variables such as number1, number2, number3, etc., you can declare a single array numbers and access its elements using an index, like numbers[0], numbers[1], and numbers[2]. This makes arrays an efficient and organized way to handle lists of values such as marks of students, temperatures, salaries, or any other collection of data.

Declaring Arrays in Java

Before using an array, you must first declare a **reference variable** that can store the address of the array. Declaring an array does not allocate memory for its elements it only defines the array's type and name.

There are two syntaxes for declaring an array in Java, but the preferred way is to use square brackets after the data type.

Syntax:

dataType[] arrayName; **// preferred way**

or

dataType arrayName[]; **// works, but not preferred**

Example:

int[] marks; **// preferred**
int rollNumbers[]; **// alternate way**
double[] prices; **// array of double type**

Here, marks is an array reference variable that can store a sequence of integers, while prices can store floating-point numbers.

Creating Arrays

Once an array is declared, you must **create** it using the new keyword, which allocates memory for the specified number of elements.

Syntax:

arrayName = new dataType[arraySize];

Here, arraySize represents the number of elements the array will hold.

Example:

int[] marks = new int[5];

This statement creates an array marks that can store 5 integer values indexed from 0 to 4.

You can also declare, create, and initialize an array in a single statement:

Example:

int[] marks = {85, 90, 78, 92, 88};

In this case, the array has 5 elements, automatically initialized with the given values.

Accessing Array Elements

Each element in an array is accessed using its **index**, which starts at **0** and goes up to arrayName.length - 1. You can read or modify elements using this index.

Example:

```
int[] marks = {85, 90, 78, 92, 88};
System.out.println(marks[0]); // Output: 85
marks[2] = 80;                // Update third element
```

Here, marks[0] accesses the first element, and marks[2] = 80 updates the third element.

Processing Arrays Using Loops

Since arrays are sequential and contain elements of the same type, they are commonly processed using **loops** especially for and foreach loops.

Example: Creating, Iterating, and Performing Operations

```
public class TestArray {
  public static void main(String[] args) {
    double[] numbers = {2.5, 3.6, 4.2, 5.0};
```

```
 // Printing all elements
      for (int i = 0; i < numbers.length; i++) {
        System.out.println(numbers[i]);
      }

      // Calculating total sum
      double total = 0;
      for (int i = 0; i < numbers.length; i++) {
        total += numbers[i];
      }
      System.out.println("Total = " + total);

      // Finding the maximum element
      double max = numbers[0];
      for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] > max) max = numbers[i];
      }
      System.out.println("Maximum = " + max);
   }
}
```

Output:

```
2.5
3.6
4.2
5.0
Total = 15.3
Maximum = 5.0
```

Explanation: Here, the program prints all array elements, calculates their sum, and finds the largest value using loops.

The Enhanced (Foreach) Loop

Java introduced the **enhanced for loop** (also known as the **foreach loop**) in **JDK 1.5**. This loop simplifies array traversal because it automatically handles iteration without using an index.

Example:

```
public class ForeachExample {
   public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40, 50};
```

```
    for (int num : numbers) {
      System.out.println(num);
    }
  }
}
```

Output:

```
10
20
30
40
50
```

Explanation: In each iteration, num stores one element of the array sequentially, eliminating the need for an index variable.

Passing Arrays to Methods

Just like primitive data types, arrays can also be **passed as arguments** to methods. When you pass an array to a method, you are actually passing its **reference**, which means any changes made to the array inside the method will affect the original array.

Example:

```
public class ArrayExample {
  public static void printArray(int[] arr) {
    for (int num : arr) {
      System.out.print(num + " ");
    }
  }

  public static void main(String[] args) {
    int[] values = {3, 5, 7, 9};
    printArray(values);
  }
}
```

Output:

```
3 5 7 9
```

Explanation: The array values is passed to the method printArray(), which prints each element. Since arrays are passed by reference, the original data can be modified if needed.

Returning Arrays from Methods

Methods in Java can also **return arrays**. This is useful when performing operations like reversing, sorting, or filtering elements.

Example:

```
public class ReverseArray {
  public static int[] reverse(int[] list) {
    int[] result = new int[list.length];
    for (int i = 0, j = list.length - 1; i < list.length; i++, j--) {
      result[j] = list[i];
    }
    return result;
  }

  public static void main(String[] args) {
    int[] original = {10, 20, 30, 40, 50};
    int[] reversed = reverse(original);

    for (int num : reversed) {
      System.out.print(num + " ");
    }
  }
}
```

Output:

50 40 30 20 10

Explanation: The method reverse() returns a new array containing the elements of the original array in reverse order.

The java.util.Arrays Class

Java provides a utility class called **java.util.Arrays** that contains several **static methods** to perform operations such as sorting, searching, comparing, and filling arrays. These methods make array handling easier and more efficient.

Commonly Used Methods:

Method	Description
Arrays.sort(array)	Sorts the array in ascending order.

Arrays.binarySearch(array, key)	Searches for a key using the binary search algorithm (array must be sorted).
Arrays.equals(array1, array2)	Returns true if both arrays are equal.
Arrays.fill(array, value)	Assigns the same value to every element in the array.

Example Using Arrays Class

```
import java.util.Arrays;

public class ArraysExample {
  public static void main(String[] args) {
    int[] numbers = {5, 2, 9, 1, 7};

    // Sorting array
    Arrays.sort(numbers);
    System.out.println("Sorted Array: " + Arrays.toString(numbers));

    // Searching an element
    int index = Arrays.binarySearch(numbers, 7);
    System.out.println("Index of 7: " + index);

    // Filling array
    Arrays.fill(numbers, 0);
    System.out.println("After Filling: " + Arrays.toString(numbers));
  }
}
```

Output:

```
Sorted Array: [1, 2, 5, 7, 9]
Index of 7: 3
After Filling: [0, 0, 0, 0, 0]
```

Explanation: Here, the Arrays class is used to sort the array, perform a binary search, and fill all elements with a single value.

Advantages of Using Arrays

1. **Efficient Data Storage:** Arrays provide a systematic way to store multiple elements of the same data type.
2. **Easy Access:** Elements can be accessed directly using indices.

3. **Compact Memory Usage:** Memory allocation is continuous and fixed in size.
4. **Fast Operations:** Access and modification of elements are very fast due to direct indexing.

Limitations of Arrays

1. **Fixed Size:** Once declared, the array size cannot be changed.
2. **Homogeneous Data:** Arrays can only hold elements of the same data type.
3. **Lack of Flexibility:** Insertion and deletion operations are complex compared to collections like ArrayList.

Strings

A String in Java is an object that represents a sequence of characters. Although it behaves like a primitive in many ways (you can write "hello" directly), String is a class (java.lang.String) and provides many utility methods to inspect, transform, and compare sequences of characters. Strings are used everywhere user input, file contents, messages, formatting, keys in maps, etc.

Creating Strings (literals and constructors)

You can create strings in two common ways: string literals and constructors.

- Literal: String s = "hello"; these are interned in the String Pool.
- Constructor: String s2 = new String("hello"); creates a new object even if "hello" exists in the pool.

Example:

```
String a = "Java";
String b = "Java";
String c = new String("Java");

System.out.println(a == b); // true - same pooled object
System.out.println(a == c); // false - different object
System.out.println(a.equals(c)); // true - content equality
```

Immutability of Strings

Strings in Java are **immutable**: once created, their contents cannot be changed. Any operation that seems to modify a string actually creates a new String object. Immutability is important for safety (thread-safety), security (internals not tampered), and use as keys in collections.

Example (immutability demonstration):

```
String s = "cat";
s.toUpperCase();          // returned "CAT" but s still points to "cat"
System.out.println(s);    // prints "cat"

s = s.toUpperCase();      // reassigns variable to new String "CAT"
System.out.println(s);    // prints "CAT"
```

The String Pool (interning)

Java keeps a special memory area called the **String Pool** (or intern pool). String literals are automatically placed there. Calling intern() on a String returns a pooled reference for equal content. Pooling reduces memory and speeds up equality checks via == for literals (but prefer equals() for content checks).

Example:

```
String s1 = "hello";
String s2 = new String("hello").intern();
System.out.println(s1 == s2); // true
```

Comparing Strings (equals, ==, compareTo)

Use equals() to compare contents. == checks reference identity (same object). compareTo() (from Comparable) lexicographically compares two strings and returns negative/zero/positive like most compare functions.

Example:

```
String a = "apple";
String b = "banana";
System.out.println(a.equals(b));      // false
System.out.println(a.compareTo(b));   // negative (because "apple" < "banana")
```

Commonly used String methods

Java String has many useful methods. Here are the most used with short descriptions:

- length() - number of characters.
- charAt(int index) - character at index.
- substring(int start, int end) - slice of the string (end exclusive).
- indexOf(...), lastIndexOf(...) - find positions.
- contains(CharSequence) - true if substring exists.
- startsWith(...), endsWith(...).

- toLowerCase(), toUpperCase().
- trim() - remove leading/trailing whitespace.
- replace(oldChar, newChar) and replace(CharSequence, CharSequence).
- split(regex) - break into array using regex.
- equalsIgnoreCase(...) - content compare ignoring case.
- format(...) - static formatting similar to String.format.
- valueOf(...) - convert primitive / object to String.

Examples:

```
String s = "  Hello, Java!  ";
System.out.println(s.length());              // 15 (counts spaces)
System.out.println(s.trim());                // "Hello, Java!"
System.out.println(s.substring(2, 7));       // "Hello"
System.out.println(s.contains("Java"));      // true
System.out.println(s.replace("Java", "World")); // "  Hello, World!  "
String[] parts = s.trim().split(",\\s*");    // ["Hello", "Java!"]
```

String concatenation and performance

Concatenation with + is easy: "a" + "b" produces "ab". The Java compiler optimizes many concatenations, but repeated concatenation in loops using + creates many temporary String objects and is inefficient. Use StringBuilder (or StringBuffer if thread-safety is required) when building strings dynamically, especially in loops.

Bad (slow in loops):

```
String s = "";
for (int i = 0; i < 1000; i++) {
   s += i; // creates new String each iteration
}
```

Good (fast):

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
   sb.append(i);
}
String result = sb.toString();
```

StringBuilder vs StringBuffer

- StringBuilder (since Java 5) is *not synchronized* -faster for single-threaded use.

- StringBuffer is synchronized thread-safe but slower. Prefer StringBuilder unless you need thread safety.

Formatting strings (String.format)

String.format uses printf-style formatting and is useful for constructing strings with numbers, padding, decimal precision, etc.

Example:

```
String name = "Priya";
int age = 25;
String s = String.format("Name: %s, Age: %d", name, age);
System.out.println(s); // "Name: Priya, Age: 25"

double pi = Math.PI;
System.out.println(String.format("Pi to 2 decimals: %.2f", pi)); // "3.14"
```

Regular expressions with strings

String and regex integrate via matches(), split(regex), and replaceAll(regex, replacement). matches() checks the whole string (not partial), so use .* if you want substring matching, or prefer Pattern/Matcher for complex needs.

Example:

```
String email = "bob@example.com";
boolean ok = email.matches("[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-z]{2,}");
System.out.println(ok); // true/false
```

Converting to/from other types

String ↔ primitives:

- Integer.parseInt("123"), Double.parseDouble("3.14") - convert from String to primitive.
- String.valueOf(123) or Integer.toString(123) - convert to String.

Example:

```
int n = Integer.parseInt("42");
String s = String.valueOf(3.1415);
```

Useful idioms and tips

- Always use equals() (or Objects.equals(a, b) if a may be null) to compare strings.
- If you need to sort strings case-insensitively, use String::compareToIgnoreCase.
- For constant concatenation ("a" + "b"), the compiler may optimize it at compile-time.
- Avoid new String(...) unless you have a strong reason (it bypasses interning and wastes memory).
- For heavy logging or repeated appends, prefer StringBuilder to reduce garbage.

Examples

1. Check palindrome (case-insensitive, ignore spaces):

```
public static boolean isPalindrome(String s) {
   String cleaned = s.replaceAll("\\s+", "").toLowerCase();
   String reversed = new StringBuilder(cleaned).reverse().toString();
   return cleaned.equals(reversed);
}
```

2. Count occurrences of a substring:

```
public static int countOccurrences(String text, String sub) {
   int count = 0;
   int idx = 0;
   while ((idx = text.indexOf(sub, idx)) != -1) {
      count++;
      idx += sub.length();
   }
   return count;
}
```

3. Build CSV row efficiently:

```
String[] fields = {"Alice", "25", "Engineer"};
StringBuilder sb = new StringBuilder();
for (int i = 0; i < fields.length; i++) {
   if (i > 0) sb.append(',');
   sb.append(fields[i]);
}
String csv = sb.toString(); // "Alice,25,Engineer"
```

When to use String vs StringBuilder

Use String for static text, keys, or when you don't mutate the content heavily. Use StringBuilder when you construct strings incrementally or inside loops. Use StringBuffer only when multiple threads will

1.9 Inheritance

Inheritance in Java is a core concept of Object-Oriented Programming (OOP). It allows one class to acquire the properties (fields) and behaviors (methods) of another class. In simple terms, inheritance enables the creation of new classes derived from existing ones, allowing code reuse and improved organization. A subclass that inherits from another class can utilize the fields and methods defined in its parent class.

Example:

In the example below, Animal serves as the base (parent) class, while Dog, Cat, and Cow are subclasses that extend the Animal class.

Implementation:

// Parent class

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

// Child class

class Dog extends Animal {

void sound() {

System.out.println("Dog barks"); } }

```
// Child class
class Cat extends Animal {
void sound() {
System.out.println("Cat meows");
}
}
// Child class
class Cow extends Animal {
void sound() {
System.out.println("Cow moos");
}
}
// Main class
public class Geeks {
public static void main(String[] args) {
Animal a;
a = new Dog();
a.sound();
a = new Cat();
a.sound();
a = new Cow();
a.sound();
}
```

}

Output

Dog barks

Cat meows

Cow moos

Explanation:

- Animal is the base class.
- Dog, Cat and Cow are derived classes that extend Animal class and provide specific implementations of the sound() method.
- The Geeks class is the driver class that creates objects and demonstrates runtime polymorphism using method overriding.

Syntax

class ChildClass extends ParentClass {

// Additional fields and methods

}

Why We use Inheritance in Java?

Code Reusability

Inheritance promotes code reusability, as the code written in a superclass can be shared among all its subclasses. This allows child classes to directly access and use the functionality defined in the parent class, reducing redundancy.

Method Overriding

Inheritance makes method overriding possible, which is essential for achieving Run-Time Polymorphism in Java. Through overriding, a subclass can provide its own implementation of a method that already exists in the superclass.

Abstraction

Inheritance also supports abstraction, allowing developers to hide unnecessary details and show only the essential functionality to the user. This helps simplify complex systems by focusing on what an object does rather than how it does it.

Key Terminologies in Java Inheritance

Class

A class is a blueprint or template used to create objects that share similar attributes and behaviors. It is not a real-world entity but a structural representation that defines the properties and methods common to a group of objects.

Super Class / Parent Class

The superclass (also called the base or parent class) is the class whose properties and methods are inherited by another class.

Sub Class / Child Class

The subclass (also called the derived, extended, or child class) is the class that inherits from another class. It can use the parent class's members and also define its own additional fields and methods.

Extends Keyword

The extends keyword is used in Java to enable one class to inherit from another class.

How Inheritance Works in Java

In Java, inheritance is implemented using the extends keyword. When a class extends another, it automatically acquires all the non-private members (fields and methods) of the parent class. The subclass can then use these inherited members, override existing methods, or introduce new ones to extend or modify the parent class's functionality.

Inheritance is one of the core principles of Object-Oriented Programming (OOP) in Java. It allows one class to acquire the properties and behaviors (fields and methods) of another class. Depending on how classes inherit from one another, Java supports several types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance (through Interfaces)**
5. **Hybrid Inheritance (through Interfaces)**

Types of Inheritance in Java

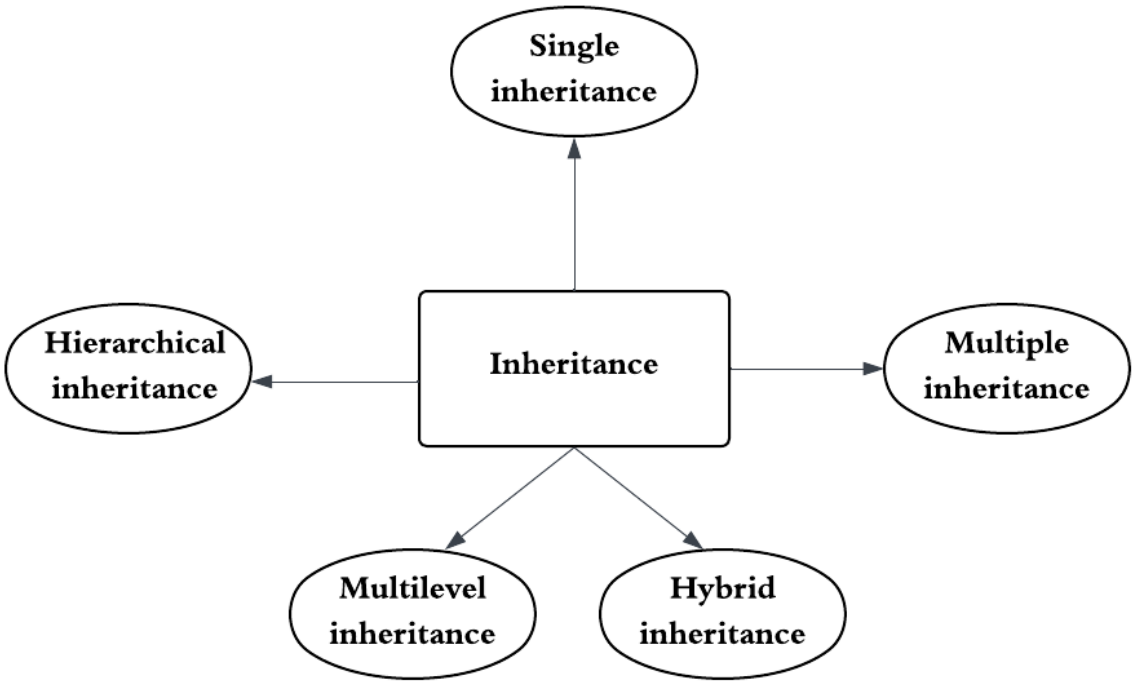


Fig. 1.12 Types of Inheritance.

1.Single Inheritance

In single inheritance, a subclass inherits from only one superclass. It means that a child class can directly access the fields and methods of a single parent class. This is the most basic and common form of inheritance in Java.

Example:

```
// Parent class
class Animal {
    Animal() {
        System.out.println("This is an Animal");
    }

    void eat() {
        System.out.println("Animals can eat");
    }
}
```

```
// Child class
class Dog extends Animal {
    Dog() {
        System.out.println("This Animal is a Dog");
    }

    void bark() {
        System.out.println("Dog barks");
    }
}

public class SingleInheritanceDemo {
    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.eat();   // inherited method
        obj.bark();  // subclass method
    }
}
```

Output:

```
This is an Animal
This Animal is a Dog
Animals can eat
Dog barks
```

2.Multilevel Inheritance

In multilevel inheritance, a class inherits from another class, which itself is inherited from a different class. In other words, the derived class acts as a parent for another subclass, forming a multi-level chain.

Example:

```
class LivingBeing {
    LivingBeing() {
        System.out.println("All living beings breathe");
    }
}
class Animal extends LivingBeing {
    Animal() {
        System.out.println("Animals are living beings");
    }
}
```

```
class Dog extends Animal {
    Dog() {
        System.out.println("Dog is an Animal");
    }
}

public class MultilevelInheritanceDemo {
    public static void main(String[] args) {
        Dog obj = new Dog();
    }
}
```

Output:

```
All living beings breathe
Animals are living beings
Dog is an Animal
```

3.Hierarchical Inheritance

In hierarchical inheritance, multiple subclasses inherit from a single superclass. This allows multiple child classes to share the common properties and behaviors defined in their parent class.

Example:

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    void area() {
        System.out.println("Area of Circle = πr²");
    }
}

class Rectangle extends Shape {
    void area() {
        System.out.println("Area of Rectangle = l × b");
    }
}
```

```
public class HierarchicalInheritanceDemo {
    public static void main(String[] args) {
        Circle c = new Circle();
        Rectangle r = new Rectangle();

        c.draw();
        c.area();

        r.draw();
        r.area();
    }
}
```

Output:

```
Drawing a shape
Area of Circle = πr²
Drawing a shape
Area of Rectangle = l × b
```

4.Multiple Inheritance

Java does **not** support multiple inheritance using classes, as it can lead to ambiguity (commonly known as the Diamond Problem). However, it allows multiple inheritance **through interfaces**. This means a single class can implement multiple interfaces, thereby inheriting abstract behaviors from all of them.

Example:

```
interface Printable {
    default void print() {
        System.out.println("Printing the document");
    }
}

interface Scannable {
    default void scan() {
        System.out.println("Scanning the document");
    }
}

class AllInOnePrinter implements Printable, Scannable {
    void info() {
        System.out.println("This is an All-In-One Printer");
    }
```

```
}

public class MultipleInheritanceDemo {
    public static void main(String[] args) {
        AllInOnePrinter printer = new AllInOnePrinter();
        printer.info();
        printer.print();
        printer.scan();
    }
}
```

Output:

```
This is an All-In-One Printer
Printing the document
Scanning the document
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance (for example, single and multiple inheritance). Since Java does not support multiple inheritance with classes, hybrid inheritance can only be achieved using **interfaces**.

Example:

```
class Machine {
    void start() {
        System.out.println("Machine is starting...");
    }
}

interface Electric {
    void charge();
}
interface Manual {
    void operate();
}

class Robot extends Machine implements Electric, Manual {
    public void charge() {
        System.out.println("Robot is charging...");
    }

    public void operate() {
        System.out.println("Robot is operating manually");
```

```
    }
}

public class HybridInheritanceDemo {
    public static void main(String[] args) {
        Robot r = new Robot();
        r.start();
        r.charge();
        r.operate();
    }
}
```

Output:

```
Machine is starting...
Robot is charging...
Robot is operating manually
```

IS-A Relationship in Java

The **IS-A** relationship represents inheritance in Java. When one class extends another, it signifies that the subclass is a type of the superclass.

Example:

```
class Vehicle { }
class Car extends Vehicle { }
class Bike extends Vehicle { }
class ElectricCar extends Car { }

public class RelationshipDemo {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        Car c = new Car();
        Bike b = new Bike();
        System.out.println(v instanceof Vehicle);
        System.out.println(c instanceof Car);
        System.out.println(b instanceof Vehicle);
    }
}
```

Output:

```
true
true
```

true

In the above example:

- Vehicle is the superclass of Car and Bike.
- Car and Bike are subclasses of Vehicle.
- ElectricCar is a subclass of both Car and Vehicle.

What Can Be Done in a Subclass?

In a subclass, you can perform several actions involving the inherited members:

1. **Use inherited fields and methods** directly as if they were defined in the subclass.
2. **Declare new fields** specific to the subclass.
3. **Override methods** from the superclass to provide a new implementation.
4. **Hide static methods** in the subclass using the same method name.
5. **Add new methods** that don't exist in the parent class.
6. **Invoke the superclass constructor** using the super keyword.

Example:

```
class Employee {
    String name = "Ravi";
    void work() {
        System.out.println(name + " works 8 hours a day");
    }
}

class Manager extends Employee {
    String department = "Sales";

    @Override
    void work() {
        System.out.println(name + " manages the " + department + " department");
    }

    void showDetails() {
        super.work(); // calling parent method
        work();       // calling overridden method
    }
}

public class SubclassDemo {
    public static void main(String[] args) {
        Manager m = new Manager();
```

```
        m.showDetails();
    }
}
```

Output:

Ravi works 8 hours a day
Ravi manages the Sales department

Advantages of Inheritance in Java

1. **Code Reusability:** Inheritance allows you to reuse existing code. Subclasses can utilize fields and methods from their parent classes, reducing duplication.
2. **Abstraction:** It supports abstraction by allowing developers to define generalized behavior in abstract classes and implement specific functionality in subclasses.
3. **Class Hierarchy:** Inheritance helps organize code into a structured class hierarchy, reflecting real-world relationships between objects.
4. **Polymorphism:** Through inheritance, Java enables polymorphism, allowing objects to take multiple forms and execute overridden methods dynamically at runtime.

Disadvantages of Inheritance in Java

1. **Increased Complexity:** Deep or poorly designed inheritance hierarchies can make code more complicated and harder to debug.
2. **Tight Coupling:** Subclasses are tightly dependent on their parent classes. Any modification in the superclass can unintentionally affect the behavior of the subclasses.

Constructors in inheritance

In Java, when one class inherits another, both the parent and child classes can define their own constructors. However, the **constructor of the parent class is always executed first**, followed by the constructor of the child class. This ensures that the base (parent) class is properly initialized before the subclass adds its own additional functionality.

How Constructors Work in Inheritance

When an object of a subclass is created, Java automatically calls the constructor of its superclass **before** executing the subclass constructor. This happens so that the inherited properties of the parent class are set up first.

If the subclass does not explicitly call a parent constructor using super(), the Java compiler automatically inserts a call to the **default constructor** of the parent class (if it exists).

If the parent class has **no default constructor**, you must explicitly call one of its parameterized constructors using super(arguments); otherwise, the code will fail to compile.

Example 1: Default Constructor in Inheritance

When a subclass object is created, the **default constructor** of the parent class executes automatically before the child constructor.

```
class Animal {
   Animal() {
      System.out.println("Parent class constructor: Animal");
   }
}

class Dog extends Animal {
   Dog() {
      System.out.println("Child class constructor: Dog");
   }
}

public class Demo1 {
   public static void main(String[] args) {
      Dog obj = new Dog();
   }
}
```

Output

```
Parent class constructor: Animal
Child class constructor: Dog
```

Explanation

When the Dog object is created:

1. Java automatically calls the parent (Animal) constructor first.
2. Once the parent class is initialized, the Dog constructor executes.
3. This ensures proper initialization from top to bottom in the inheritance chain.

Example 2: Using super() **to Call a Parameterized Constructor**

By default, Java calls the **no-argument constructor** of the superclass. However, if the superclass defines only a **parameterized constructor**, you must explicitly call it using the super() keyword from the subclass constructor.

```
class Employee {
    Employee(String name) {
        System.out.println("Employee constructor called. Name: " + name);
    }
}

class Manager extends Employee {
    Manager(String name, String department) {
        super(name);  // Explicitly calling parent class constructor
        System.out.println("Manager constructor called. Department: " + department);
    }
}

public class Demo2 {
    public static void main(String[] args) {
        Manager obj = new Manager("Alice", "Finance");
    }
}
```

Output

```
Employee constructor called. Name: Alice
Manager constructor called. Department: Finance
```

Explanation

- The super(name) statement explicitly calls the parent class (Employee) constructor with a parameter.
- This ensures that the Employee part of the object is properly initialized before executing the Manager constructor.

Constructor Chaining in Inheritance

Constructor chaining refers to the process where one constructor calls another constructor either within the same class (using this()) or from its parent class (using super()). This chaining continues until the topmost constructor (of the base class) is executed.

Example 3: Constructor Chaining Across Multiple Classes

```
class Device {
    Device() {
        System.out.println("Constructor of Device");
    }
}
```

```
class Mobile extends Device {
    Mobile() {
        super(); // Calls Device constructor
        System.out.println("Constructor of Mobile");
    }
}

class Smartphone extends Mobile {
    Smartphone() {
        super(); // Calls Mobile constructor
        System.out.println("Constructor of Smartphone");
    }
}

public class Demo3 {
    public static void main(String[] args) {
        Smartphone obj = new Smartphone();
    }
}
```

Output

```
Constructor of Device
Constructor of Mobile
Constructor of Smartphone
```

Explanation

- When the Smartphone object is created, the constructor of Device runs first, followed by Mobile, and finally Smartphone.
- The super() calls ensure constructors execute from **top to bottom** in the inheritance hierarchy.

Example 4: Constructor Chaining with this() **and** super()

In Java, this() is used to call another constructor in the same class, while super() calls the constructor of the superclass. Both cannot be used together in the same constructor, as each must be the **first statement** in the constructor body.

```
class Vehicle {
    Vehicle(String type) {
        System.out.println("Vehicle Type: " + type);
    }
}
```

```
class Car extends Vehicle {
    Car() {
        this("Sedan"); // Calls another constructor in the same class
        System.out.println("Default Car constructor");
    }

    Car(String model) {
        super("Car"); // Calls parent constructor
        System.out.println("Car model: " + model);
    }
}

public class Demo4 {
    public static void main(String[] args) {
        Car obj = new Car();
    }
}
```

Output

```
Vehicle Type: Car
Car model: Sedan
Default Car constructor
```

Explanation

1. The default Car() constructor first calls another constructor of the same class using this("Sedan").
2. That constructor (Car(String model)) then calls the parent (Vehicle) constructor using super("Car").
3. Execution flows upward to the base class first, then moves down the chain.

Example 5: Constructor Execution Order in Multi-Level Inheritance

```
class University {
    University() {
        System.out.println("Constructor of University");
    }
}

class Department extends University {
    Department() {
        System.out.println("Constructor of Department");
    }
}
```

```
class Student extends Department {
    Student() {
        System.out.println("Constructor of Student");
    }
}

public class Demo5 {
    public static void main(String[] args) {
        Student s = new Student();
    }
}
```

Output

```
Constructor of University
Constructor of Department
Constructor of Student
```

Explanation

Constructors are executed in **hierarchical order**:

1. University (base class)
2. Department (intermediate class)
3. Student (derived class)

This ensures that initialization happens step-by-step starting from the most general class to the most specific.

Key Points

- The constructor of the parent class always executes **before** the child class constructor.
- If there is **no explicit super() call**, Java automatically calls the parent's **default constructor**.
- If the parent class doesn't have a default constructor, you **must** call one of its constructors using super(arguments).
- super() and this() cannot appear in the same constructor because both must be the **first statement**.
- Constructor chaining allows multiple constructors to execute in an ordered manner from the parent class to the child class.

Method Overriding

Method overriding is a core OOP feature that lets a subclass provide a specific implementation for a method that is already defined in its superclass. Overriding is the

mechanism Java uses to support runtime polymorphism: a reference of the parent type can refer to an object of the child type, and the child's overridden method will run.

Method overriding occurs when a subclass defines a method with the **same name, same parameter list (signature)** and **compatible return type** as a method in its superclass. The subclass's method replaces (overrides) the superclass method for instances of the subclass. This lets subclasses modify or extend behaviour defined in the parent class.

Why override methods?

Overriding is used to:

- Customize or extend behavior for subclasses without changing the superclass.
- Implement polymorphic behavior so that code can work with parent-type references but execute child-specific implementations at runtime.
- Provide specific implementations for abstract methods declared in abstract classes or interfaces.

Rules for overriding

- **Same method signature**: name and parameter types must match exactly.
- **Return type**: must be the same or a covariant return type (a subtype of the original return type).
- **Access level**: subclass method cannot have a more restrictive access modifier than the superclass method (e.g., a public method cannot be overridden as protected).
- **Exceptions**: an overriding method cannot throw broader checked exceptions than the overridden method; it may throw fewer or narrower checked exceptions, or any unchecked exceptions.
- **final methods**: cannot be overridden.
- **static methods**: are not overridden they are hidden. The compile-time type determines which static method is called.
- **Constructors**: cannot be overridden.
- Use @Override annotation (recommended) it helps the compiler catch signature mistakes.

Basic example: simple override

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

class Dog extends Animal {
   @Override
```

```
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Demo {
    public static void main(String[] args) {
        Animal a = new Dog(); // parent reference, child object
        a.sound();            // prints "Dog barks" child method executed
    }
}
```

Explanation: Although the reference type is Animal, the JVM dispatches to Dog.sound() at runtime because the actual object is a Dog.

Overriding and runtime polymorphism

Method overriding enables writing general code that works for many types:

```
class Shape {
    void draw() { System.out.println("Drawing shape"); }
}

class Circle extends Shape {
    @Override
    void draw() { System.out.println("Drawing circle"); }
}
class Square extends Shape {
    @Override
    void draw() { System.out.println("Drawing square"); }
}
public class Renderer {
    static void render(Shape s) {
        s.draw(); // runtime chooses correct draw()
    }
    public static void main(String[] args) {
        render(new Circle()); // "Drawing circle"
        render(new Square()); // "Drawing square"
    }
}
```

You can add new subclasses without changing render(); polymorphism does the dispatching.

Covariant return types

Java allows a subclass overriding method to return a subtype of the original return type:

```
class Parent {
   Number getValue() { return 42; }
}

class Child extends Parent {
   @Override
   Integer getValue() { return 42; } // Integer is subtype of Number
}
```

This is legal and often useful for more specific return values in subclasses.

Access modifiers and overriding

You cannot reduce visibility when overriding:

```
class A {
   protected void show() { }
}
class B extends A {
   // public is allowed (wider), protected is allowed (same), private is not allowed (narrower)
   @Override
   public void show() { } // OK
}
```

If you try to change protected to private, the compiler will error.

Exceptions in overriding

An overriding method cannot declare new or broader checked exceptions than the overridden method:

```
class X {
   void process() throws java.io.IOException { }
}

class Y extends X {
   @Override
   void process() throws java.io.FileNotFoundException { } // OK (narrower)
   // void process() throws Exception { } // NOT allowed broader checked exception
}
```

Unchecked exceptions (subclasses of RuntimeException) are not restricted.

Calling the superclass method with super

Sometimes you want to extend behavior, not replace it completely:

```
class Logger {
   void log(String msg) { System.out.println("Log: " + msg); }
}

class TimestampLogger extends Logger {
   @Override
   void log(String msg) {
      String stamped = "[" + System.currentTimeMillis() + "] " + msg;
      super.log(stamped); // reuse parent behavior after adding timestamp
   }
}
```

super.method() explicitly invokes the parent-class implementation.

Static methods: hiding (not overriding)

Static methods are resolved at compile time based on the reference type:

```
class Parent {
   static void info() { System.out.println("Parent info"); }
}

class Child extends Parent {
   static void info() { System.out.println("Child info"); }
}

public class Test {
   public static void main(String[] args) {
      Parent p = new Child();
      p.info(); // prints "Parent info" - static method of reference type
      Child.info(); // prints "Child info"
   }
}
```

This is method hiding, not overriding.

Overriding abstract methods

Abstract methods in a superclass (or interface) must be implemented (overridden) by concrete subclasses:

```
abstract class Vehicle {
   abstract void accelerate();
}

class Bike extends Vehicle {
   @Override
   void accelerate() {
      System.out.println("Bike accelerates by pedaling");
   }
}
```

This is a common and important use-case.

When not to override

- If a method is final it cannot be overridden.
- When behavior must remain consistent across subclasses.
- Static methods overriding does not apply.

Quick checklist before overriding

- Same name + parameter list.
- Compatible (same or covariant) return type.
- Don’t reduce visibility.
- Don’t throw broader checked exceptions.
- Use @Override to catch mistakes early.
- Know the difference between instance method overriding and static method hiding.

Final example illustrating many points

```
class Parent {
   public Number calculate() { return 10; }
   public static void staticMethod() { System.out.println("Parent static"); }
   public final void finalMethod() { System.out.println("Cannot override me"); }
}

class Child extends Parent {
   @Override
   public Integer calculate() { return 20; } // covariant return
   public static void staticMethod() { System.out.println("Child static"); } // hiding
```

```
    // public void finalMethod() { } // compile error if uncommented
}

public class Demo {
    public static void main(String[] args) {
        Parent p = new Child();
        System.out.println(p.calculate()); // calls Child.calculate() -> 20
        p.staticMethod();               // Parent.staticMethod() because static binding
        Child.staticMethod();           // Child.staticMethod()
    }
}
```

Output

```
20
Parent static
Child static
```

Use of super

The super keyword in Java provides a way for a subclass to explicitly refer to members (constructors, methods, or fields) of its immediate superclass. It exists because inheritance can introduce name hiding (a subclass defines a field or method with the same name as its parent) or because the subclass needs to invoke initialization logic from the parent (i.e., call a parent constructor). Using super makes intent explicit: "use the parent's version of this thing." This is essential for correct initialization and for extending rather than replacing behavior.

Common uses of super

Call a parent constructor: super() or super(args) must be the first statement in a constructor when used. If you don't write it, Java inserts a no-arg super() automatically (only if the superclass has a no-arg constructor).

Invoke a parent method: super.methodName(args) invokes the superclass implementation even if the subclass has overridden that method. This is useful when you want to augment, not replace, behavior.

Access a hidden field: super.fieldName accesses a field in the superclass if the subclass has a field with the same name.

Important constraints: super cannot be used in a static context (static methods or static initializers), and super() must appear as the first line of a constructor if used.

Example: using super to call the parent constructor and parent method

```
class Person {
    String name;
    Person(String name) {
        this.name = name;
        System.out.println("Person constructed: " + name);
    }
    void introduce() {
        System.out.println("I am " + name);
    }
}

class Employee extends Person {
    String role;
    Employee(String name, String role) {
        super(name);              // call parent constructor first
        this.role = role;
        System.out.println("Employee constructed: " + role);
    }
    @Override
    void introduce() {
        super.introduce();        // reuse parent behavior
        System.out.println("I work as a " + role);
    }
}
public class SuperDemo {
    public static void main(String[] args) {
        Employee e = new Employee("Anita", "Engineer");
        e.introduce();
    }
}
```

Output

```
Person constructed: Anita
Employee constructed: Engineer
I am Anita
I work as a Engineer
```

This shows super(name) ensuring the Person portion of the object is initialized, and super.introduce() reusing the parent behavior before adding the subclass-specific line.

Abstract classes

An abstract class is a class that cannot be instantiated on its own and can contain abstract methods (methods declared without a body) as well as concrete methods (methods with implementations), static fields, constructors, and member fields. Abstract classes let you define a common contract and partial implementation for a family of related classes. They are ideal when you want to provide shared code (fields or helper methods) together with methods that each subclass must implement.

When to choose an abstract class

Use an abstract class when you have a common base of state/behavior that multiple subclasses should share and you want some methods to be enforced (abstract) while others are reused (concrete). Abstract classes are appropriate if subclasses share state (fields) or helper methods. If you only need to declare a contract (no shared implementation or fields), modern Java interfaces (with default/static methods) may also be suitable.

Rules

- An abstract class can have constructors these are used when a concrete subclass is instantiated and the superclass portion still needs initialization.
- If a class contains at least one abstract method, it must be declared abstract.
- A concrete subclass must implement all inherited abstract methods, or itself be declared abstract.
- Abstract methods cannot be final (contradiction). Abstract classes can have final concrete methods though.

Example: abstract class with abstract and concrete methods

```
abstract class Shape {
   String color;
   Shape(String color) {          // abstract class can have constructor
      this.color = color;
   }

   // abstract method must be implemented by concrete subclasses
   abstract double area();

   // concrete method shared behavior
   void display() {
      System.out.println("Color: " + color + ", Area: " + area());
   }
}
```

```
class Circle extends Shape {
    double radius;
    Circle(double radius, String color) {
        super(color);            // call Shape constructor
        this.radius = radius;
    }
    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double w, h;
    Rectangle(double w, double h, String color) {
        super(color);
        this.w = w; this.h = h;
    }
    @Override
    double area() {
        return w * h;
    }
}

public class AbstractDemo {
    public static void main(String[] args) {
        Shape s1 = new Circle(2.0, "Red");
        Shape s2 = new Rectangle(3.0, 4.0, "Blue");
        s1.display();
        s2.display();
    }
}
```

Possible output

```
Color: Red, Area: 12.566370614359172
Color: Blue, Area: 12.0
```

This example shows:

Shape cannot be instantiated directly.

Shape provides shared code (color field and display() method).

Circle and Rectangle must implement area().

Combining super and abstract classes (common pattern)

It's common for an abstract class to define constructor logic or helper methods that subclasses use through super(...). The abstract base initializes shared state; each concrete subclass calls super(...) to ensure that shared initialization happens.

Example: abstract base with constructor + subclass using super

```
abstract class Vehicle {
    String brand;
    Vehicle(String brand) { this.brand = brand; }

    abstract void start();

    void showBrand() {
        System.out.println("Brand: " + brand);
    }
}

class Bike extends Vehicle {
    Bike(String brand) {
        super(brand); // initialize Vehicle
    }
    @Override
    void start() {
        System.out.println("Kick start the bike");
    }
}

public class CombinedDemo {
    public static void main(String[] args) {
        Vehicle v = new Bike("Yamaha");
        v.showBrand(); // uses concrete method from abstract class
        v.start();     // subclass implementation
    }
}
```

Useful patterns and design idioms

Template Method Pattern: An abstract class defines a template method (concrete) that calls several abstract steps. Subclasses provide those steps. This is a textbook pattern that leverages both abstract methods and concrete helpers in the base class.

Partial implementation: Use abstract classes when you want to force certain implementations but provide shared utilities (e.g., protected helper methods or cached fields).

Constructor initialization: Use super(...) in subclass constructors to ensure base state is set consistently.

Template Method quick example

```
abstract class DataProcessor {
   // template method (final so subclasses can't change the sequence)
   public final void process() {
      read();
      transform();
      write();
   }
   abstract void read();
   abstract void transform();
   void write() { System.out.println("Default write to console"); }
}

class CSVProcessor extends DataProcessor {
   void read() { System.out.println("Read CSV"); }
   void transform() { System.out.println("Transform CSV rows"); }
}
```

1.10 Interfaces

An interface in Java is a reference type that declares a contract: a set of methods (and constants) that a class can implement. Interfaces describe what an implementing class must do, not how it does it. They are central to Java's type system and are the primary tool for achieving multiple-inheritance-like design, loose coupling, and polymorphism.

An interface is a collection of abstract method signatures and constants. A class that implements an interface agrees to provide concrete implementations of its abstract methods.

Interfaces are a way to specify capabilities (e.g., Comparable, Runnable) without prescribing a class hierarchy.

Key implicit modifiers

- Methods declared in an interface are implicitly public (and prior to Java 8: abstract).
- Fields declared in an interface are implicitly public static final (constants).

Why use interfaces?

Multiple-type inheritance: A class can implement many interfaces, allowing it to present multiple capabilities without the complications of multiple class inheritance.

Decoupling and flexibility: Code can depend on interface types, not concrete classes, which makes testing and swapping implementations easier.

Polymorphism: You can write methods that accept interface types and work with any implementing class.

API contracts: Interfaces define clear contracts for library consumers.

Basic example: defining and implementing an interface

```
interface Drivable {

void accelerate(int increment);

void brake(int decrement);

}

class Car implements Drivable {

private int speed = 0;

@Override

public void accelerate(int increment) {

speed += increment;

System.out.println("Car speed: " + speed);

}
```

```
@Override

public void brake(int decrement) {

speed = Math.max(0, speed - decrement);

System.out.println("Car speed: " + speed);

}

}
```

Here, Car implements Drivable and must provide both methods.

Default and static methods

To evolve interfaces without breaking existing implementations, Java 8 introduced default methods (concrete methods with default keyword) and static methods inside interfaces.

default methods let interfaces provide a default implementation that implementing classes may override or reuse.

static methods are utility methods related to the interface and are called on the interface itself.

```
interface Logger {

default void log(String msg) {

System.out.println("[LOG] " + msg);

}

static void info(String msg) {

System.out.println("[INFO] " + msg);

}

}

class App implements Logger {

// inherits default log(), can override if needed

}
```

Use: Logger.info("starting") or new App().log("hello").

Private methods in interfaces

Interfaces may contain private methods to reuse code between multiple default methods. These are only callable from inside interface methods.

interface Formatter {

default void print(String s) {

System.out.println(format(s));

}

private String format(String s) {

return "[" + s.trim() + "]";

}

}

Multiple inheritance via interfaces and conflict resolution

A class can implement multiple interfaces. If two interfaces provide default methods with the same signature, the implementing class must resolve the conflict either override the method or specify which super-interface's default to use via InterfaceName.super.method().

interface A { default void hello() { System.out.println("A"); } }

interface B { default void hello() { System.out.println("B"); } }

class MyClass implements A, B {

@Override

public void hello() {

A.super.hello(); **// choose A's default**

// or B.super.hello();

}

}

This explicit resolution prevents ambiguity (the diamond problem).

Functional interfaces and lambdas

A functional interface is an interface with exactly one abstract method (it may have default/static methods as well). They are targets for lambda expressions and method references.

Common examples: Runnable, Callable<V>, Comparator<T>, Function<T,R>.

@FunctionalInterface

interface Transformer {

String transform(String input);

}

public class LambdaDemo {

public static void main(String[] args) {

Transformer toUpper = s -> s.toUpperCase();

System.out.println(toUpper.transform("hello")); // **HELLO**

}

}

Using @FunctionalInterface is optional but documents intent and enables compile-time checks.

Marker interfaces

A marker interface contains no methods; it signals metadata about a class to the runtime or frameworks (e.g., Serializable, Cloneable). Marker interfaces are an older pattern; annotations are often used instead today.

class MyData implements java.io.Serializable { /* no methods to implement */ }

Interface inheritance (interfaces extending interfaces)

An interface can extend one or more other interfaces, inheriting their abstract and default methods. This allows composing contracts.

interface Readable { void read(); }

interface Writable { void write(); }

interface ReadWritable extends Readable, Writable { /* combines both */ }

Interfaces vs Abstract Classes - when to use which?

Use an interface when you want to define a capability or contract that many unrelated classes might implement (e.g., Comparable, Iterable), or when you need multiple inheritance of type.

Use an abstract class when you want to share state (fields) or partial implementation among closely related classes. An abstract class can have instance fields; interfaces cannot (only constants).

Since Java 8/9 interfaces can have default/static/private methods, the line has blurred but interfaces still cannot hold instance state.

Practical examples

1) Multiple capability example

interface Flyable { void fly(); }

interface Swimmable { void swim(); }

class Duck implements Flyable, Swimmable {

public void fly() { System.out.println("Duck flies"); }

public void swim() { System.out.println("Duck swims"); }

}

Duck can be used anywhere a Flyable or Swimmable is needed.

2) Default method override and reuse

interface Validator {

default boolean isValid(String s) { return s != null && !s.isBlank(); }

```
}

class NameValidator implements Validator {

@Override

public boolean isValid(String s) {

return Validator.super.isValid(s) && s.length() <= 50;

}

}
```

NameValidator reuses the interface's default check and adds extra rules.

3) Functional interface with a lambda

```
import java.util.function.Function;

public class FuncExample {

public static void main(String[] args) {

Function<Integer, String> toHex = i -> Integer.toHexString(i);

System.out.println(toHex.apply(255)); // ff

}

}
```

Dynamic method dispatch

Dynamic Method Dispatch is a **core concept in Object-Oriented Programming (OOP)** and one of the most powerful features of Java's runtime polymorphism. It refers to the process of resolving which version of an overridden method to call **at runtime**, rather than at compile time. This mechanism allows Java to support polymorphic behavior, where the method that gets executed depends on the **actual object type** being referenced, not the **reference variable type**.

Dynamic Method Dispatch, also known as **Runtime Polymorphism**, occurs when a **superclass reference variable** is used to refer to an **object of a subclass**. When an overridden method is called through the superclass reference, Java determines which version of the

method to execute based on the **object** that the reference variable points to not on the type of the reference itself.

The **method call is resolved at runtime**, depending on the actual object being referred to.

This contrasts with **compile-time polymorphism**, such as method overloading, where the decision about which method to call is made during compilation.

How Dynamic Method Dispatch Works

1. A superclass reference variable can hold a reference to any of its subclass objects.
2. When an overridden method is invoked on this reference, the **JVM** determines which version of the method to call based on the actual object (subclass) being referred to at runtime.
3. This enables Java to exhibit different behaviors depending on the object type, even when using a single reference variable.

Example: Demonstrating Dynamic Method Dispatch

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

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

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

public class DynamicDispatchExample {
    public static void main(String[] args) {
        Animal a; // reference of superclass

        a = new Dog(); // refers to Dog object
        a.sound();     // calls Dog's version of sound()
```

```
        a = new Cat(); // refers to Cat object
        a.sound();     // calls Cat's version of sound()
    }
}
```

Output

Dog barks
Cat meows

Explanation

- The reference variable a is of type Animal (the parent class).
- Initially, it points to a Dog object, so Dog's version of sound() executes.
- Later, the same reference a points to a Cat object, so Cat's version of sound() executes.

Here, the method call a.sound() is resolved **at runtime**, depending on the actual object (Dog or Cat) this is what makes it dynamic.

Importance of Dynamic Method Dispatch

1. **Achieves Runtime Polymorphism:** It allows one interface (like a parent class reference) to be used for multiple actual types (child class objects).
2. **Flexibility and Extensibility:** It enables programs to be more flexible and easily extendable. New subclasses can be added without changing the existing code that uses the superclass reference.
3. **Code Reusability:** It allows developers to write code that can operate on objects of different classes through a common interface or parent class.

Packages

Packages in Java are one of the most important mechanisms for organizing and managing large programs. They serve as containers for classes, interfaces, enumerations, and annotations, allowing developers to group related types together in a single namespace.

Packages help in avoiding naming conflicts, controlling access, and making code modular and easier to maintain. In short, packages in Java are similar to folders in a file system they provide a structured way to organize your source files and compiled class files.

A Java package is a namespace that groups related classes and interfaces together. Packages help in logically categorizing Java classes based on their functionality so that classes, interfaces, and other components can be easily located and reused.

In other words, a package acts as a container that provides access protection and namespace management. It ensures that two classes with the same name can exist in different packages

without any conflict.

Why Packages are Used in Java

Java packages are used for several purposes:

Avoiding Name Conflicts: Two classes with the same name can exist in different packages (for example, java.util.Date and java.sql.Date).

Access Control: Packages provide controlled access to classes and methods using access modifiers like public, protected, and default (package-private).

Code Reusability: Related classes can be grouped into one package for reuse in other programs.

Easier Maintenance: Classes with related functionality are grouped together, making code organization cleaner and easier to maintain.

Encapsulation: Packages help encapsulate code logically, hiding implementation details and exposing only what is necessary through public interfaces.

Types of Packages in Java

Java supports two main types of packages:

1. Built-in Packages (Predefined Packages)

These are packages that come with the Java Development Kit (JDK). They provide a wide range of ready-to-use classes and interfaces for various purposes like input/output operations, data structures, networking, and more.

Some commonly used built-in packages are:

java.lang - Contains core classes like String, Math, Object, and System. (This package is imported automatically.)

java.io - Contains classes for input and output operations like File, InputStream, and OutputStream.

java.util - Provides utility classes like ArrayList, HashMap, Date, and Collections.

java.net - Contains classes for networking such as Socket, URL, and InetAddress.

java.sql - Provides classes for database connectivity (JDBC).

2. User-defined Packages

These are packages created by the programmer to group related classes and interfaces that they develop.User-defined packages help maintain modularity and prevent naming conflicts when working on large-scale applications.

Creating a User-defined Package

- To create a package, use the package keyword followed by the package name at the top of the Java source file.
- The package statement must be the first line in the file (before any import or class declarations).

Example:

// File name: Animal.java

```
package animals;

public interface Animal {
   void eat();
   void travel();
}
```

In this example, we have created a package named animals that contains an interface Animal.Now, we can create another class in the same package that implements this interface.

// File name: MammalInt.java

```
package animals;

public class MammalInt implements Animal {
   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public static void main(String[] args) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
```

}

Compiling the Package

To compile these files, you must specify the destination directory using the -d option with javac.

Example:

javac -d . Animal.java
javac -d . MammalInt.java

This will create a directory structure that reflects the package name.
The compiled class files will be stored inside a folder named animals.

Output:

Mammal eats
Mammal travels

Importing Packages in Java

Once a package is created, you can use its classes in another program by importing them using the import statement.

There are three main ways to use a class from another package:

1. Using Fully Qualified Class Name

You can refer to a class in another package using its full name (package name + class name).

payroll.Employee emp = new payroll.Employee();

2. Importing an Entire Package

You can import all the classes of a package using the * wildcard.

import payroll.*;

This allows you to use all classes in the payroll package without specifying the package name each time.

3. Importing a Specific Class

You can import only the required class instead of the entire package.

```
import payroll.Employee;
```

Example: Importing Packages

// File: Employee.java

```
package payroll;

public class Employee {
   public void mailCheck() {
      System.out.println("Pay received.");
   }
}
```

// File: Boss.java

```
package payroll;
import payroll.Employee;
public class Boss {
   public void payEmployee(Employee e) {
      e.mailCheck();
   }

   public static void main(String[] args) {
      Boss boss = new Boss();
      Employee emp = new Employee();
      boss.payEmployee(emp);
   }
}
```

Output:

Pay received.

Note:

- Import statements must appear after the package statement and before the class declaration.
- A file can contain any number of import statements.
- Directory Structure of a Java Package
- When you create a package, the package name becomes part of the directory structure where your source and compiled class files are stored.

Example:

// File name: Car.java
```
package vehicle;

public class Car {
  public void display() {
    System.out.println("This is a car.");
  }
}
```

This file should be placed inside a directory named vehicle:

/vehicle/Car.java
When compiled:

javac -d . Car.java

The compiler creates the directory structure and places the class file inside it:
/vehicle/Car.class

So the qualified class name is vehicle.Car, and the path name is /vehicle/Car.java.

Package Naming Convention

To avoid naming conflicts between packages from different organizations, Java follows a unique naming convention that uses a company's reversed internet domain name as the base package name.

Example:

Company domain name: apple.com

Package name: com.apple.projectname

Example file structure:

com/apple/computers/Dell.java

Example

// File name: Dell.java
```
package com.apple.computers;
```

```
public class Dell {
   public void display() {
      System.out.println("This is a Dell computer.");
   }
}
```

Compile using:

javac -d . Dell.java
This will create:
.\com\apple\computers\Dell.class

Classpath and Package Access

When running a program, the JVM needs to locate the compiled .class files.The classpath is the path that tells the Java compiler and JVM where to look for compiled class files.

If your classpath is set to:
<path>\classes
And your package is:
com.apple.computers
Then the JVM looks for class files at:

<path>\classes\com\apple\computers

You can set multiple class paths by separating them using a semicolon (Windows) or a colon (Unix).

Advantages of Using Packages in Java

- **Organized Code:** Helps structure large programs logically.
- **Namespace Management:** Avoids name clashes between classes.
- **Access Control:** Provides better encapsulation using access modifiers.
- **Reusability:** Promotes code reuse by grouping related classes.
- **Easier Maintenance:** Related classes are easier to manage and locate.
- **Scalability:** Makes adding new functionality easier without affecting existing code.

Exception handling

Exception handling is a vital part of Java programming that ensures the smooth execution of programs even when unexpected situations (errors or abnormal conditions) occur during runtime. Instead of abruptly terminating a program when an error occurs, Java provides a

powerful mechanism called **exception handling** that allows developers to **detect, handle, and recover** from runtime errors gracefully.

An **exception** is an event that disrupts the normal flow of a program's execution. It occurs during the execution of a program and indicates that something unexpected has happened, such as dividing by zero, accessing an invalid array index, or attempting to open a file that doesn't exist.

In Java, exceptions are objects derived from the class java.lang.Throwable. When an exception occurs, the Java runtime system creates an exception object and hands it over to the **runtime environment** to handle the situation.

If the exception is not properly handled, the program terminates abruptly, displaying an error message and a stack trace.

Types of Exceptions

Java classifies exceptions into three main categories:

1. Checked Exceptions

Checked exceptions are those that the compiler checks at compile-time. These exceptions must be either **caught** using a try-catch block or **declared** in the method signature using the throws keyword. They usually occur due to external factors like file handling, network connections, or input/output operations.

Example: IOException, SQLException, FileNotFoundException

```
import java.io.*;

public class CheckedExample {
   public static void main(String[] args) {
      try {
         FileReader fr = new FileReader("data.txt");
      } catch (FileNotFoundException e) {
         System.out.println("File not found: " + e);
      }
   }
}
```

Explanation: The compiler forces you to handle FileNotFoundException, because file operations depend on external resources and may fail.

2. Unchecked Exceptions

Unchecked exceptions (also called **runtime exceptions**) are not checked at compile time. These exceptions usually occur due to programming logic errors and can be avoided through better coding practices. If an unchecked exception occurs and is not handled, the program terminates abnormally.

Example: ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException

```
public class UncheckedExample {
   public static void main(String[] args) {
      int[] arr = {1, 2, 3};
      try {
         System.out.println(arr[5]); // invalid index
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception caught: " + e);
      }
      System.out.println("Program continues...");
   }
}
```

Output:

```
Exception caught: java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
Program continues...
```

3. Errors

Errors are serious issues that occur beyond the control of the programmer. They are not meant to be handled by programs, as they represent system-level problems like hardware failure, JVM crashes, or memory exhaustion.

Example: OutOfMemoryError, StackOverflowError

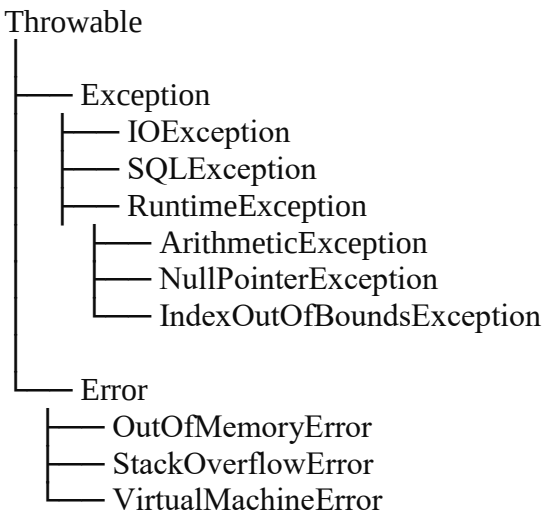
```
public class ErrorExample {
   public static void main(String[] args) {
      try {
         ErrorExample.recursive(); // infinite recursion
      } catch (StackOverflowError e) {
         System.out.println("Stack overflow occurred!");
      }
   }

   static void recursive() {
```

```
        recursive(); // causes StackOverflowError
    }
}
```

Exception Hierarchy

All exceptions and errors in Java are derived from the class **Throwable**, which has two main subclasses:



Java Exception Handling Mechanism

Java uses **five keywords** for exception handling:

1. **try** – Block of code that may throw an exception.
2. **catch** – Used to handle the exception that occurs in the try block.
3. **finally** – Used to execute important code such as resource cleanup, regardless of whether an exception occurs.
4. **throw** – Used to explicitly throw an exception.
5. **throws** – Declares exceptions that can be thrown by a method.

Basic Syntax of Exception Handling

```
try {
    // risky code that may throw exception
} catch (ExceptionType e) {
    // handle the exception
} finally {
    // optional block, always executed
}
```

Example: Basic Try-Catch Block

```
public class DivisionExample {
    public static void main(String[] args) {
        int a = 10, b = 0;
        try {
            int result = a / b; // division by zero
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero!");
        }
        System.out.println("Program continues...");
    }
}
```

Output:

```
Error: Cannot divide by zero!
Program continues...
```

Multiple Catch Blocks

You can have multiple catch blocks to handle different types of exceptions thrown in the same try block. Each block catches a specific type of exception.

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] nums = {1, 2, 3};
            System.out.println(nums[5]);
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error occurred!");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is invalid!");
        } catch (Exception e) {
            System.out.println("General exception occurred!");
        }
    }
}
```

Output:

```
Array index is invalid!
```

Nested Try-Catch Blocks

A try block can be nested inside another try block. This is useful when different sections of code can throw different exceptions.

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            try {
                int[] arr = new int[3];
                arr[5] = 10; // inner try block
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Inner catch: Invalid index!");
            }
            int x = 10 / 0; // outer try block
        } catch (ArithmeticException e) {
            System.out.println("Outer catch: Division by zero!");
        }
    }
}
```

Output:

```
Inner catch: Invalid index!
Outer catch: Division by zero!
```

The finally **Block**

The finally block always executes, regardless of whether an exception occurs or not. It is typically used for resource cleanup operations, such as closing files or database connections.

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int num = 5 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

Output:

Exception caught: java.lang.ArithmeticException: / by zero
This block always executes.

Throwing Exceptions Using throw

The throw keyword is used to explicitly throw an exception from a method or any block of code. It is followed by an instance of Throwable or its subclasses.

```
public class ThrowExample {
   static void checkAge(int age) {
      if (age < 18)
         throw new ArithmeticException("Access denied - You must be 18 or older.");
      else
         System.out.println("Access granted - You are old enough!");
   }
   public static void main(String[] args) {
      try {
         checkAge(16);
      } catch (ArithmeticException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

Output:

Access denied - You must be 18 or older.

Declaring Exceptions Using throws

The throws keyword is used in method declarations to specify the types of exceptions that the method may throw. This alerts the caller that it must handle or declare the exception.

```
import java.io.*;

public class ThrowsExample {
   static void readFile() throws IOException {
      FileReader fr = new FileReader("data.txt");
      fr.read();
      fr.close();
   }

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

```
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Exception handled: " + e);
        }
    }
}
```

Custom (User-Defined) Exceptions

Java allows you to create your own exceptions by extending the Exception class. These are useful when you want to handle application-specific error conditions.

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

public class CustomExceptionExample {
    static void validateAge(int age) throws InvalidAgeException {
        if (age < 18)
            throw new InvalidAgeException("Age is below 18 - Not eligible to vote.");
        else
            System.out.println("Eligible to vote.");
    }

    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

Output:

Caught exception: Age is below 18 - Not eligible to vote.

Advantages of Exception Handling

1. **Improves Program Reliability:** Prevents abrupt program termination.
2. **Separates Error-Handling Code:** Keeps the main logic separate from error-handling logic.

3. **Propagates Errors:** Allows exceptions to be passed up the call stack.
4. **Group Error Types:** Enables specific handling for different types of exceptions.
5. **Ensures Resource Cleanup:** finally ensures resources are released even when exceptions occur.

Practical

1. Write a Java program that demonstrates the use of classes, objects, constructors, method overloading, inheritance, super, and abstract classes.

Aim

To Write a Java program that demonstrates the use of **classes**, **objects**, **constructors**, **constructor chaining**, **method overloading**, **inheritance**, **super**, **method overriding**, **abstract classes**, and **polymorphism**.

Procedure

1. Define an **abstract base class** Person with fields, a constructor, and an abstract method showDetails().
2. Create a concrete class Employee that **extends** Person. Implement showDetails(), provide **overloaded methods** calculateSalary(...) (demonstrating method overloading), and show a constructor that calls super(...).
3. Create a subclass Manager that **extends Employee** and **overrides** showDetails(); demonstrate super.method() usage inside the overridden method and constructor chaining.
4. In the main method of a public driver class InheritanceDemo, create objects of Employee and Manager, call overloaded methods and overridden methods, and display constructor invocation order and runtime polymorphism.
5. Compile and run to observe output demonstrating the required concepts.

Program

```
// InheritanceDemo.java
// Demonstrates: classes, objects, constructors, constructor chaining, method overloading,
// inheritance, use of super, abstract classes, method overriding and polymorphism.

abstract class Person {
    protected String name;
    protected int age;
```

```
    // Constructor for Person
    Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Person constructor called for: " + name);
    }

    // Abstract method - must be implemented by subclasses
    public abstract void showDetails();

    // Concrete helper method
    public void basicInfo() {
        System.out.println("Basic Info -> Name: " + name + ", Age: " + age);
    }
}

class Employee extends Person {
    protected String employeeId;
    protected double baseSalary;

    // Constructor chaining: calls Person constructor using super(...)
    Employee(String name, int age, String employeeId, double baseSalary) {
        super(name, age); // calls Person constructor
        this.employeeId = employeeId;
        this.baseSalary = baseSalary;
        System.out.println("Employee constructor called for ID: " + employeeId);
    }

    // Overloaded constructors (constructor overloading)
    Employee(String name, int age, String employeeId) {
        this(name, age, employeeId, 25000.0); // calls the other constructor in same class
    }

    // Implement abstract method from Person
    @Override
    public void showDetails() {
        // reuse parent helper method
        basicInfo();
        System.out.println("Role: Employee, Employee ID: " + employeeId);
    }

    // Method overloading: calculateSalary with different parameter lists

    // 1) No-arg: returns base salary
    public double calculateSalary() {
```

```
        System.out.println("calculateSalary() called");
        return baseSalary;
    }

    // 2) With bonus
    public double calculateSalary(double bonus) {
        System.out.println("calculateSalary(double bonus) called");
        return baseSalary + bonus;
    }

    // 3) With bonus and allowance
    public double calculateSalary(double bonus, double allowance) {
        System.out.println("calculateSalary(double bonus, double allowance) called");
        return baseSalary + bonus + allowance;
    }

    // 4) Using number of days worked to pro-rate salary (different signature)
    public double calculateSalary(int daysWorked, int totalWorkingDays) {
        System.out.println("calculateSalary(int daysWorked, int totalWorkingDays) called");
        if (totalWorkingDays == 0) return 0;
        return (baseSalary * daysWorked) / totalWorkingDays;
    }

    // A method that can be overridden
    public void work() {
        System.out.println(name + " (Employee) is doing general work.");
    }
}

class Manager extends Employee {
    private String department;

    // Manager constructor calls Employee constructor using super(...)
    Manager(String name, int age, String employeeId, double baseSalary, String department)
{
        super(name, age, employeeId, baseSalary); // Person <- Employee <- Manager
        this.department = department;
        System.out.println("Manager constructor called for department: " + department);
    }

    // Overloaded Manager constructor (demonstrates constructor chaining using this())
    Manager(String name, int age, String employeeId, String department) {
        this(name, age, employeeId, 50000.0, department); // default baseSalary
    }
```

```
    // Override showDetails() and use super.showDetails() to reuse parent behavior
    @Override
    public void showDetails() {
        super.showDetails(); // call Employee's showDetails which calls Person.basicInfo()
        System.out.println("Role: Manager, Department: " + department);
    }

    // Override work()
    @Override
    public void work() {
        // extend parent behavior
        super.work(); // call Employee.work()
        System.out.println(name + " (Manager) is supervising the " + department + " department.");
    }

    // Manager-specific method
    public void conductMeeting() {
        System.out.println(name + " is conducting a meeting in " + department + ".");
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        System.out.println("=== Creating Employee emp1 ===");
        Employee emp1 = new Employee("Raj", 28, "E1001", 30000.0);
        System.out.println();

        System.out.println("=== Showing emp1 details ===");
        emp1.showDetails();
        System.out.println();

        System.out.println("=== Calling overloaded calculateSalary on emp1 ===");
        System.out.println("Base salary: " + emp1.calculateSalary());
        System.out.println("Salary with bonus 2000: " + emp1.calculateSalary(2000.0));
        System.out.println("Salary with bonus 2000 and allowance 1500: " + emp1.calculateSalary(2000.0, 1500.0));
        System.out.println("Pro-rated (20/30 days): " + emp1.calculateSalary(20, 30));
        System.out.println();

        System.out.println("=== Creating Manager mgr1 (using default base salary constructor) ===");
        Manager mgr1 = new Manager("Anita", 35, "M2001", "Sales");
        System.out.println();
```

```
        System.out.println("=== Showing mgr1 details (overridden) ===");
        mgr1.showDetails();
        System.out.println();

        System.out.println("=== Demonstrating overridden work() and super.work() ===");
        mgr1.work();
        System.out.println();

        System.out.println("=== Polymorphism: Person reference to Manager object ===");
        Person p = new Manager("Sunil", 40, "M2002", 75000.0, "HR");
        p.showDetails(); // dynamic method dispatch -> Manager.showDetails()
        System.out.println();

        System.out.println("=== Result Summary ===");
        System.out.println("Objects created: emp1 (Employee), mgr1 (Manager), p (Person reference to Manager).");
    }
}
```

Sample Output

Creating Employee emp1
Person constructor called for: Raj
Employee constructor called for ID: E1001

Showing emp1 details
Basic Info -> Name: Raj, Age: 28
Role: Employee, Employee ID: E1001

Calling overloaded calculateSalary on emp1
calculateSalary() called
Base salary: 30000.0
calculateSalary(double bonus) called
Salary with bonus 2000: 32000.0
calculateSalary(double bonus, double allowance) called
Salary with bonus 2000 and allowance 1500: 33500.0
calculateSalary(int daysWorked, int totalWorkingDays) called
Pro-rated (20/30 days): 20000.0

Creating Manager mgr1 (using default base salary constructor)
Person constructor called for: Anita
Employee constructor called for ID: M2001
Manager constructor called for department: Sales

Showing mgr1 details (overridden)
Basic Info -> Name: Anita, Age: 35
Role: Employee, Employee ID: M2001
Role: Manager, Department: Sales

Demonstrating overridden work() and super.work()
Anita (Employee) is doing general work.
Anita (Manager) is supervising the Sales department.

Polymorphism: Person reference to Manager object
Person constructor called for: Sunil
Employee constructor called for ID: M2002
Manager constructor called for department: HR
Basic Info -> Name: Sunil, Age: 40
Role: Employee, Employee ID: M2002
Role: Manager, Department: HR

Explanation

- **Abstract class** Person demonstrates a template that cannot be instantiated directly and enforces implementation of showDetails() by subclasses.
- **Constructors**: Person → Employee → Manager show constructor invocation order. super(...) is used to call the parent constructor and properly initialize inherited fields.
- **Constructor chaining**: this(...) inside Employee and Manager demonstrates chaining among constructors.
- **Method overloading**: Employee.calculateSalary(...) shows multiple methods with the same name but different parameter lists.
- **Inheritance & overriding**: Manager overrides showDetails() and work() from its parent; super.showDetails() and super.work() reuse parent behavior and extend it.
- **Polymorphism / Dynamic Method Dispatch**: A Person reference p refers to a Manager object; p.showDetails() invokes the overridden Manager implementation at runtime, demonstrating dynamic method dispatch.

Result

Thus, a Java program was successfully written and executed to demonstrate the concepts of Classes, Objects, Constructors, Method Overloading, Inheritance, the use of super, and Abstract Classes.

2. Create a Java application that includes built–in and user–defined exception handling using try, catch, finally, throw, and throws.

Aim

To write a **Java application** that demonstrates the use of **built-in and user-defined exception handling** using the keywords **try**, **catch**, **finally**, **throw**, and **throws**.

Procedure

1. Start by defining a **user-defined exception class** by extending the Exception class.
2. Create a main class to demonstrate:

   - ❖ Built-in exceptions (like ArithmeticException or ArrayIndexOutOfBoundsException).
   - ❖ User-defined exceptions using the throw and throws keywords.

3. Use **try-catch blocks** to handle exceptions gracefully.
4. Include a **finally block** to execute cleanup code (executed regardless of whether an exception occurs).
5. Compile and run the program to observe different types of exception handling.

Program

```
// ExceptionHandlingDemo.java
// Demonstrates built-in and user-defined exception handling using try, catch, finally, throw, and throws

import java.util.Scanner;

// Step 1: Create a user-defined exception
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class ExceptionHandlingDemo {

    // Step 2: Method that uses throws and throw for user-defined exception
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age below 18 is not allowed for voting.");
        } else {
            System.out.println("You are eligible to vote!");
```

```
    }
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    // Demonstrate built-in exception handling
    try {
        System.out.println("=== BUILT-IN EXCEPTION HANDLING ===");
        System.out.print("Enter two numbers: ");
        int a = sc.nextInt();
        int b = sc.nextInt();

        // This may cause ArithmeticException
        int result = a / b;
        System.out.println("Result = " + result);

        // This may cause ArrayIndexOutOfBoundsException
        int[] numbers = {10, 20, 30};
        System.out.println("Accessing invalid array index: " + numbers[5]);
    } catch (ArithmeticException e) {
        System.out.println("Exception caught: Division by zero is not allowed!");
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception caught: Array index out of range!");
    } catch (Exception e) {
        System.out.println("General exception: " + e.getMessage());
    } finally {
        System.out.println("Finally block executed for built-in exceptions.\n");
    }

    // Demonstrate user-defined exception handling
    try {
        System.out.println("=== USER-DEFINED EXCEPTION HANDLING ===");
        System.out.print("Enter your age: ");
        int age = sc.nextInt();

        // Method call that may throw user-defined exception
        checkAge(age);

    } catch (InvalidAgeException e) {
        System.out.println("Custom Exception caught: " + e.getMessage());
    } finally {
        System.out.println("Finally block executed for user-defined exceptions.");
        sc.close(); // closing scanner resource
    }
```

```
        System.out.println("\nProgram executed successfully!");
    }
}
```

Sample Output:

Case 1: Built-in exception (Division by zero)
BUILT-IN EXCEPTION HANDLING
Enter two numbers: 10 0
Exception caught: Division by zero is not allowed!
Finally block executed for built-in exceptions.

USER-DEFINED EXCEPTION HANDLING
Enter your age: 16
Custom Exception caught: Age below 18 is not allowed for voting.
Finally block executed for user-defined exceptions.

Program executed successfully!

Case 2: No Exception Occurred
BUILT-IN EXCEPTION HANDLING
Enter two numbers: 20 5
Result = 4
Exception caught: Array index out of range!
Finally block executed for built-in exceptions.

USER-DEFINED EXCEPTION HANDLING
Enter your age: 25
You are eligible to vote!
Finally block executed for user-defined exceptions.

Program executed successfully!

Explanation

- **try block:** Contains the code that may throw exceptions.
- **catch block:** Catches specific exceptions and handles them gracefully.
- **finally block:** Executes regardless of whether an exception occurs (used for cleanup operations).
- **throw keyword:** Used to explicitly throw an exception object.
- **throws keyword:** Declares that a method may throw a particular exception.
- **Built-in exceptions** like ArithmeticException
- ArrayIndexOutOfBoundsException are handled.

- **User-defined exception (InvalidAgeException)** is thrown and handled to demonstrate custom exception handling.

Result

Thus, a **Java program** was successfully written and executed to demonstrate **exception handling** using **built-in** and **user-defined exceptions** with the keywords try, catch, finally, throw, and throws.

CHAPTER – 2

MULTITHREADING, JDBC, AND WEB CLIENT UI DEVELOPMENT

2.1 Java Thread Model

A **thread** in Java is the smallest unit of execution within a program. It is a **lightweight subprocess** that runs independently but shares the same memory space with other threads of the same process. This allows multiple tasks to execute **concurrently**, improving performance and efficiency.

Creating Threads in Java

There are **two main ways** to create a thread in Java:

1. **By Extending the Thread Class**
2. **By Implementing the Runnable Interface**

1. Creating a Thread by Extending the Thread Class

We can create a thread by creating a new class that **extends the Thread class**. Then, **override the run() method**, which defines the code that will execute in the new thread. Finally, create an object of this class and call the **start() method**, which automatically calls run() in a separate thread.

Example:

```
class PrintNumbers extends Thread {

    // run() method defines the thread's job
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Number: " + i);
            try {
                Thread.sleep(500); // pause for 0.5 second
            } catch (InterruptedException e) {
                System.out.println("Thread Interrupted");
            }
```

```
        }
    }
}
public class MainThreadExample {
    public static void main(String[] args) {
        PrintNumbers t1 = new PrintNumbers();
        t1.start();  // Starts the thread
    }
}
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

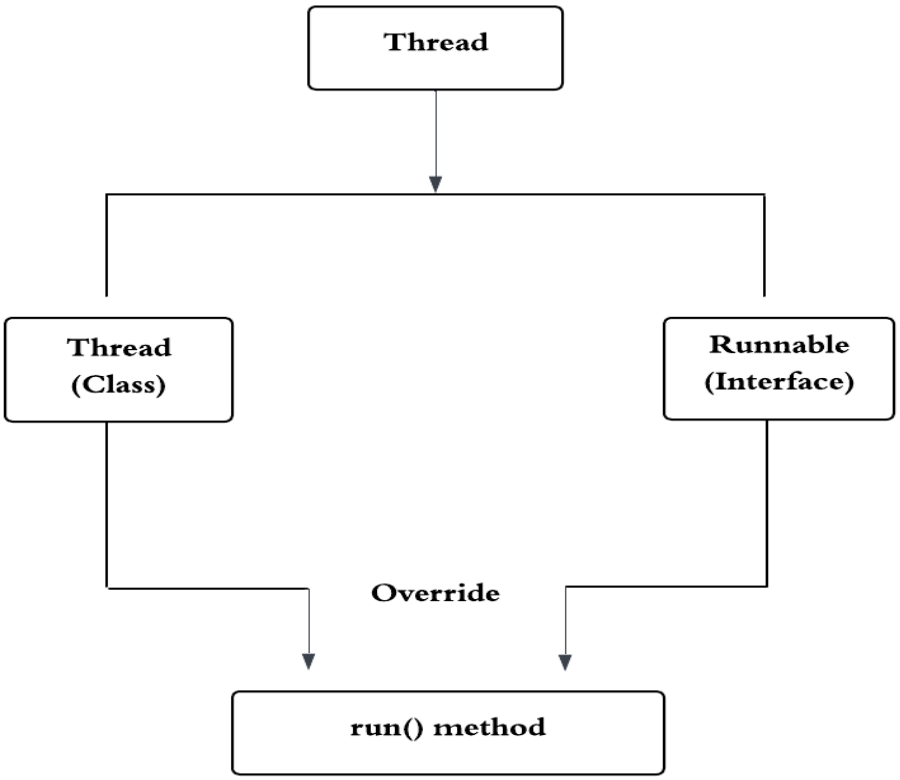


Fig. 2.1 Thread Creation.

2. Creating a Thread by Implementing the Runnable Interface

Alternatively, we can create a thread by implementing the **Runnable** interface.In this case, define the run() method inside your class, create a Thread object by passing the instance of your class to it, and call **start()**.

Example:

```
class MessagePrinter implements Runnable {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Printing message " + i);
            try {
                Thread.sleep(700);
            } catch (InterruptedException e) {
                System.out.println("Runnable thread interrupted");
            }
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        MessagePrinter mp = new MessagePrinter();
        Thread t1 = new Thread(mp);  // Pass Runnable to Thread
        t1.start();  // Start the thread
    }
}
```

Output:

```
Printing message 1
Printing message 2
Printing message 3
```

Life Cycle of a Thread in Java

A Java thread passes through several states during its lifetime:

1. New State

- When a **Thread object is created**, it is said to be in the **New** state.
- At this point, the thread has not yet started execution.
- The thread enters this state when you write:

- Thread t = new Thread();
- It remains here until the **start()** method is called.

Transition: New → Runnable (when start() is called)

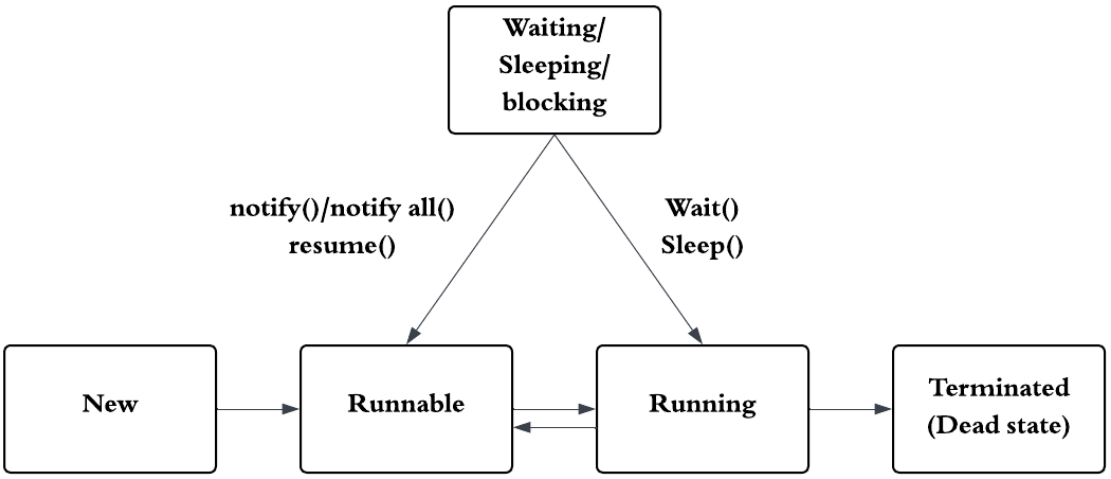


Fig. 2.2 Life cycle of Thread.

2. Runnable State

- After calling start(), the thread moves to the **Runnable** state.
- In this state, the thread is **ready to run** and waiting for CPU time.
- The Java thread scheduler decides which thread will move to the **Running** state next based on system policies (like priority or round-robin scheduling).

Transition: Runnable → Running (when CPU allocates execution time)

Can also return from:

- Waiting/Blocked → Runnable when notify(), notifyAll(), or resume() is called.

3. Running State

- When the thread gets CPU time, it moves into the **Running** state.
- Here, the **run()** method of the thread is executed.
- The thread remains in this state until:
  - It finishes execution, or
  - It is sent to **waiting** or **sleeping** state due to a method call like wait() or sleep().

Transition:

- Running → Waiting/Sleeping/Blocking (when thread temporarily stops execution)
- Running → Terminated (when thread completes execution)

4. Waiting / Sleeping / Blocking State

This state includes **three related sub-states** where a thread is temporarily **not runnable**.

- **Waiting:** The thread waits indefinitely for another thread to perform a particular action (like calling notify() or notifyAll()).
- **Sleeping:** The thread is paused for a fixed time using sleep(milliseconds).
- **Blocked:** The thread is waiting to access a resource currently used by another thread.

Transition:

- From Running - this state when wait() or sleep() is called.
- Returns to Runnable when the waiting period ends, or another thread invokes notify(), notifyAll(), or resume().

5. Terminated (Dead) State

- Once the **run()** method finishes execution, the thread enters the **Terminated** or **Dead** state.
- At this point, the thread's life is over it **cannot be restarted** by calling start() again.

Transition: Running → Terminated

Example Illustration

```
class ThreadLifeCycle extends Thread {
   public void run() {
      System.out.println("Thread is running...");
      try {
         Thread.sleep(1000); // Moves to sleeping state
         System.out.println("Thread woke up...");
      } catch (InterruptedException e) {
         System.out.println("Thread interrupted...");
      }
      System.out.println("Thread execution completed.");
   }

   public static void main(String[] args) {
      ThreadLifeCycle t1 = new ThreadLifeCycle();
      System.out.println("Thread state: New");
```

```
        t1.start();  // Thread moves to Runnable state
        System.out.println("Thread started...");
    }
}
```

Output:

```
Thread state: New
Thread started...
Thread is running...
Thread woke up...
Thread execution completed.
```

Concurrent Programming

Concurrent Programming in Java refers to the execution of **multiple tasks or processes simultaneously**, enabling better performance, responsiveness, and resource utilization.
In a single-threaded program, tasks are executed **sequentially** one after another. But in real-world applications, such as servers, games, or user interfaces, it's often necessary to perform multiple operations **at the same time**.

Java provides a **robust concurrency framework** that allows developers to write programs that can execute **independent parts concurrently**, either using **multithreading** or **asynchronous programming** mechanisms.

Concurrency

Concurrency means *dealing with multiple tasks at once*.It doesn't necessarily mean that all tasks are running at exactly the same time (that's parallelism). Instead, concurrency allows tasks to **progress independently** even if the CPU executes only one at a time, the system switches between them efficiently.

For example:

- A web browser may render a web page while downloading images.
- A music player may play songs while reading files from storage.

Java achieves concurrency mainly through **threads**.

Threads and Concurrency

A **thread** is the smallest unit of execution within a process. In concurrent programming, multiple threads can run **independently** within the same program, sharing the same memory space.

Threads allow you to:

- Perform **background tasks** while keeping the application responsive.
- Execute **multiple operations concurrently** (e.g., downloading files, updating UI, logging data).
- Utilize **multi-core processors** efficiently by running threads in parallel.

Creating Threads in Concurrent Programming

In Java, threads can be created in two main ways:

1. Extending the Thread Class

We can create a class that extends the Thread class and override the run() method, which defines the code that the thread executes.

Example:

```
class DownloadFile extends Thread {
    public void run() {
        System.out.println("Downloading file...");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            System.out.println("Download interrupted.");
        }
        System.out.println("Download completed!");
    }
}
public class ConcurrentExample1 {
    public static void main(String[] args) {
        DownloadFile t1 = new DownloadFile();
        t1.start(); // Starts a new thread
        System.out.println("Main thread continues...");
    }
}
```

Output:

```
Downloading file...
Main thread continues...
Download completed!
```

Here, the **main thread** and the **child thread (DownloadFile)** run **concurrently**.

2. Implementing the Runnable Interface

This is a more flexible way to create threads because it allows your class to extend another class (since Java supports only single inheritance).

Example:

```
class DataProcessor implements Runnable {
    public void run() {
        System.out.println("Processing data in background...");
    }
}

public class ConcurrentExample2 {
    public static void main(String[] args) {
        Thread t1 = new Thread(new DataProcessor());
        t1.start();
        System.out.println("Main thread running parallel to DataProcessor...");
    }
}
```

Output:

```
Main thread running parallel to DataProcessor...
Processing data in background...
```

Both tasks (main and data processing) execute concurrently.

The Executor Framework

- Java introduced the **Executor Framework** in java.util.concurrent package to simplify thread management.
- Instead of manually creating and starting threads, you can use **thread pools** to efficiently manage multiple concurrent tasks.

Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class Task implements Runnable {
    private String name;

    Task(String name) {
        this.name = name;
    }
```

```
    public void run() {
        System.out.println(name + " is running by " + Thread.currentThread().getName());
    }
}
public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 1; i <= 5; i++) {
            executor.execute(new Task("Task " + i));
        }
        executor.shutdown();
    }
}
```

Output (order may vary):

```
Task 1 is running by pool-1-thread-1
Task 2 is running by pool-1-thread-2
Task 3 is running by pool-1-thread-3
Task 4 is running by pool-1-thread-1
Task 5 is running by pool-1-thread-2
```

The Executor framework automatically manages thread creation, reuse, and shutdown making concurrent programming **more efficient and cleaner**.

Synchronization in Concurrent Programming

When multiple threads access **shared resources** (like variables or files), race conditions can occur. **Synchronization** ensures that only **one thread** accesses a shared resource at a time, preventing data inconsistency.

Example:

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

```
public class SyncExample {
   public static void main(String[] args) throws InterruptedException {
      Counter counter = new Counter();

      Thread t1 = new Thread(() -> {
         for (int i = 0; i < 1000; i++) counter.increment();
      });

      Thread t2 = new Thread(() -> {
         for (int i = 0; i < 1000; i++) counter.increment();
      });

      t1.start();
      t2.start();
      t1.join();
      t2.join();

      System.out.println("Final Count: " + counter.getCount());
   }
}
```

Output:

Final Count: 2000

Without synchronization, the output might be less than 2000 due to race conditions.

Advantages of Concurrent Programming

1. **Improved Performance:** Multiple tasks run simultaneously, leading to faster execution on multi-core systems.
2. **Resource Optimization:** Threads share memory and system resources efficiently.
3. **Responsive Applications:** UI and server applications remain responsive even during heavy computations.
4. **Simplified Problem Solving:** Complex tasks can be divided into smaller concurrent sub-tasks.

Challenges of Concurrency

- **Race Conditions:** Multiple threads accessing shared data simultaneously.
- **Deadlocks:** Threads waiting indefinitely for each other's resources.
- **Thread Interference:** Unpredictable behavior due to unsynchronized access.
- **Debugging Difficulty:** Concurrent programs are harder to trace and test.

2.2 Thread Priorities

In Java, every thread has a **priority** that helps the **thread scheduler** decide the order in which threads should be executed. A **thread scheduler** is part of the Java Virtual Machine (JVM) responsible for allocating CPU time to available threads.

Thread priorities are useful when you want certain tasks to execute **before or more often** than others. However, the actual scheduling behavior **depends on the operating system and JVM implementation**, so priorities **cannot guarantee** exact execution order.

When multiple threads are running concurrently, the JVM uses their **priority levels** as a hint to decide which thread to run next.

- Threads with **higher priority** are more likely to be given CPU time before lower-priority threads.
- However, **priority is not absolute** it only increases the chances of execution order, not the certainty.

For instance:

- A **high-priority thread** might handle important background tasks.
- A **low-priority thread** could perform logging or data backups.

Thread Priority Range and Constants

Java defines thread priorities using **integer values ranging from 1 to 10**. These are represented by **three built-in constants** of the Thread class:

Constant	Description	Value
Thread.MIN_PRIORITY	The lowest possible thread priority	1
Thread.NORM_PRIORITY	The default priority assigned to threads	5
Thread.MAX_PRIORITY	The highest possible thread priority	10

Default Thread Priority

By default, **all threads have a priority of 5** (Thread.NORM_PRIORITY), unless explicitly changed by the programmer.

The **main thread** the one that runs when the program starts also has a default priority of **5**.

Thread Priority Methods

The Thread class provides methods to **get** and **set** the priority of threads.

1. **getPriority()**: Returns the current priority of a thread.

Syntax: public final int getPriority();

2. **setPriority(int newPriority)**: Sets a new priority for a thread.

Syntax:

- public final void setPriority(int newPriority);
- Throws IllegalArgumentException if the given value is not between 1 and 10.

Example 1: Displaying Default Thread Priority

This example shows that the default thread priority is 5 for every thread.

```
class PriorityExample1 extends Thread {
   public void run() {
      System.out.println("Running Thread: " + Thread.currentThread().getName());
      System.out.println("Priority: " + Thread.currentThread().getPriority());
   }

   public static void main(String[] args) {
      PriorityExample1 t1 = new PriorityExample1();
      PriorityExample1 t2 = new PriorityExample1();
      PriorityExample1 t3 = new PriorityExample1();

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

Output:

```
Running Thread: Thread-0
Priority: 5
Running Thread: Thread-1
Priority: 5
Running Thread: Thread-2
Priority: 5
```

All threads have the default **NORM_PRIORITY (5)**.

Example 2: Setting Custom Thread Priorities

Here we manually assign different priorities to threads and observe the behavior.

```
class PriorityExample2 extends Thread {
    public void run() {
        System.out.println("Executing " + Thread.currentThread().getName() +
                           " with priority " + Thread.currentThread().getPriority());
    }

    public static void main(String[] args) {
        PriorityExample2 high = new PriorityExample2();
        PriorityExample2 normal = new PriorityExample2();
        PriorityExample2 low = new PriorityExample2();

        high.setPriority(Thread.MAX_PRIORITY);     // Priority 10
        normal.setPriority(Thread.NORM_PRIORITY);  // Priority 5
        low.setPriority(Thread.MIN_PRIORITY);      // Priority 1

        high.start();
        normal.start();
        low.start();
    }
}
```

Possible Output:

```
Executing Thread-0 with priority 10
Executing Thread-2 with priority 1
Executing Thread-1 with priority 5
```

The high-priority thread **may** execute first, but **order is not guaranteed** it depends on the OS thread scheduler.

Example 3: Demonstrating Priority Influence

This program demonstrates how threads with higher priorities often execute faster.

```
class PriorityDemo extends Thread {
    public PriorityDemo(String name) {
        super(name);
    }
    public void run() {
```

```
        for (int i = 1; i <= 3; i++) {
            System.out.println(getName() + " (Priority: " + getPriority() + ") → " + i);
        }
    }

    public static void main(String[] args) {
        PriorityDemo t1 = new PriorityDemo("LowPriorityThread");
        PriorityDemo t2 = new PriorityDemo("HighPriorityThread");
        t1.setPriority(2);
        t2.setPriority(9);

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

Possible Output:

HighPriorityThread (Priority: 9) → 1
HighPriorityThread (Priority: 9) → 2
LowPriorityThread (Priority: 2) → 1
HighPriorityThread (Priority: 9) → 3
LowPriorityThread (Priority: 2) → 2
LowPriorityThread (Priority: 2) → 3

The higher priority thread (**Priority 9**) tends to run more frequently, but actual order may vary.

Example 4: Getting Thread Priority Constants

You can fetch predefined priority constants from the Thread class directly.

```
public class PriorityConstants {
    public static void main(String[] args) {
        System.out.println("Minimum Priority: " + Thread.MIN_PRIORITY);
        System.out.println("Normal Priority: " + Thread.NORM_PRIORITY);
        System.out.println("Maximum Priority: " + Thread.MAX_PRIORITY);
    }
}
```

Output:

Minimum Priority: 1
Normal Priority: 5
Maximum Priority: 10

Key Points

Default Priority:

Every thread has a default priority of **5** (Thread.NORM_PRIORITY).

1. **Priority Range:** Thread priorities range between **1 (lowest)** and **10 (highest)**.
2. **Influence but not Guarantee:** Higher priority increases the chance of earlier execution but does not ensure it.
3. **Thread Scheduler:** The **JVM and OS scheduler** ultimately decide which thread runs next this behavior is **platform-dependent**.
4. **IllegalArgumentException:** Calling setPriority() with a value outside the range **1–10** will cause an exception.

Advantages of Using Thread Priority

- Allows you to indicate **relative importance** of threads.
- Helps **control scheduling** when multiple threads are competing for resources.
- Useful for **real-time or time-sensitive tasks** (e.g., animations, I/O tasks).

Limitations of Thread Priority

- The **execution order is not guaranteed** even if priorities differ.
- **Different operating systems** may treat thread priorities differently.
- Overusing thread priority can lead to **unpredictable performance** and **starvation** of lower-priority threads.

Creating and Managing Threads

In Java, multithreading allows a program to perform multiple tasks at the same time. A **thread** is the smallest independent unit of execution within a process. Each Java application runs at least one thread by default, known as the **main thread**, which starts automatically when the program begins. When we create additional threads, we enable concurrent execution, which improves performance, responsiveness, and efficiency of applications such as games, servers, and graphical user interfaces.

Creating and managing threads in Java is an essential part of concurrent programming. Java provides the **Thread class** and the **Runnable interface** to create and control threads. Managing threads involves controlling their execution flow, lifecycle, priority, synchronization, and termination.

Creating Threads in Java

There are two standard ways to create a thread in Java. One is by extending the **Thread class**, and the other is by implementing the **Runnable interface**. Both methods achieve the same goal but differ in design flexibility and inheritance use.

Creating a Thread by Extending the Thread Class

The first and simplest way to create a thread is by extending the Thread class. To do this, you define a class that inherits from Thread and override its run() method. The code inside the run() method represents the task that will execute when the thread starts. After defining the class, you create an instance of it and call the start() method. The start() method internally calls run() and begins a new thread of execution.

Syntax

```
class ClassName extends Thread {
    public void run() {
        // Code to be executed by the thread
    }

    public static void main(String[] args) {
        ClassName thread = new ClassName();
        thread.start();  // Starts the new thread
    }
}
```

Example

```
class MessageThread extends Thread {
    public void run() {
        System.out.println("Thread is running using Thread class.");
        for (int i = 1; i <= 3; i++) {
            System.out.println("Count: " + i);
            try {
                Thread.sleep(500);  // Pauses for half a second
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted.");
            }
        }
    }

    public static void main(String[] args) {
        MessageThread t1 = new MessageThread();
        t1.start();
```

```
        System.out.println("Main thread finished execution.");
    }
}
```

Explanation: In this example, the MessageThread class extends the Thread class and overrides the run() method. The start() method is used to begin the thread execution. The sleep() method temporarily pauses the thread, allowing other threads to execute. The main thread and the new thread run concurrently.

Creating a Thread by Implementing the Runnable Interface

The second method of creating a thread is by implementing the Runnable interface. This approach is preferred when your class already extends another class since Java does not support multiple inheritance. To use this method, you implement the run() method from the Runnable interface, create an object of your class, and pass it as an argument to a Thread object. Finally, you call the start() method on the Thread object to begin execution.

Syntax

```
class ClassName implements Runnable {
    public void run() {
        // Code to be executed by the thread
    }

    public static void main(String[] args) {
        ClassName obj = new ClassName();
        Thread thread = new Thread(obj);
        thread.start();
    }
}
```

Example

```
class TaskRunner implements Runnable {
    public void run() {
        System.out.println("Thread is running using Runnable interface.");
        for (int i = 1; i <= 3; i++) {
            System.out.println("Iteration: " + i);
            try {
                Thread.sleep(400);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted.");
            }
        }
    }
```

```
    public static void main(String[] args) {
        TaskRunner task = new TaskRunner();
        Thread t1 = new Thread(task);
        t1.start();
        System.out.println("Main method continues executing.");
    }
}
```

Explanation: Here, the class TaskRunner implements the Runnable interface and defines its own run() method. The object task is passed as a parameter to the Thread constructor, which creates a new thread. When the start() method is called, the thread begins execution independently from the main thread.

Managing Threads in Java

Creating a thread is only the first step. Managing threads involves controlling their behavior during execution. Java provides several methods in the Thread class that help manage threads effectively.

Starting a Thread

To start a thread, you use the start() method. This method invokes the run() method internally and allows concurrent execution. Calling run() directly will not create a new thread; it will simply execute as a normal method in the current thread.

Example

```
Thread t = new Thread(new TaskRunner());
t.start();  // Creates and starts a new thread
```

Putting a Thread to Sleep

The sleep() method is used to pause the execution of a thread temporarily for a specified time in milliseconds. It helps in controlling thread execution speed or simulating delay.

Syntax

```
Thread.sleep(milliseconds);
```

Example

```
try {
    Thread.sleep(1000); // Pauses thread for one second
} catch (InterruptedException e) {
    e.printStackTrace();
```

```
}
```

Joining Threads

The join() method is used when you want one thread to wait for the completion of another. This is useful when the execution of one thread depends on the result of another.

Example

```
class JoinExample extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }

    public static void main(String[] args) {
        JoinExample t1 = new JoinExample();
        JoinExample t2 = new JoinExample();
        t1.start();
        try {
            t1.join();  // Waits until t1 finishes
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t2.start();
    }
}
```

Explanation: In this example, the second thread t2 starts only after t1 has completed execution. The join() method ensures sequential control over thread execution.

Checking if a Thread is Alive

The isAlive() method checks whether a thread is still running or has finished execution. It returns a boolean value, true if the thread is still active, and false if it has terminated.

Syntax

```
thread.isAlive();
```

Example

```
class AliveCheck extends Thread {
   public void run() {
      System.out.println("Thread is running...");
   }

   public static void main(String[] args) {
      AliveCheck t1 = new AliveCheck();
      System.out.println("Before start: " + t1.isAlive());
      t1.start();
      System.out.println("After start: " + t1.isAlive());
   }
}
```

Stopping a Thread

Java previously used the stop() method to terminate threads, but it is now deprecated because it can leave resources in an inconsistent state. The recommended way to stop a thread is by using a flag variable inside the run() method.

Example

```
class StopExample extends Thread {

   private volatile boolean running = true;
   public void run() {
      while (running) {
         System.out.println("Thread is active...");
         try {
            Thread.sleep(500);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println("Thread stopped safely.");
   }
   public void stopThread() {
      running = false;
   }

   public static void main(String[] args) {
      StopExample t1 = new StopExample();
      t1.start();
      try {
```

```
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t1.stopThread();  // Stop thread safely
    }
}
```

Explanation: The thread continues to run while the variable running is true. When the flag is set to false, the loop ends, and the thread stops gracefully.

2.3 Thread Methods and Exceptions

Thread Methods

Threads are the core units of execution in Java that allow programs to perform multiple tasks concurrently. The Thread class in the java.lang package provides a set of **methods** that enable programmers to create, control, and coordinate threads effectively. These methods allow you to start threads, pause their execution, wait for them to finish, handle interruptions, and manage priorities. Understanding these methods is essential for writing efficient and safe multithreaded programs.

1. start() and run() Methods

In Java, the **start()** and **run()** methods are fundamental to thread execution. The run() method defines the code that a thread will execute once it starts running. However, calling run() directly behaves like a normal method call it executes in the current thread rather than creating a new one.

On the other hand, the **start()** method is used to begin the execution of a new thread. When start() is called, the Java Virtual Machine (JVM) allocates necessary resources, schedules the thread, and then automatically invokes its run() method in a separate thread of execution. This separation is what enables true concurrency.

Example:

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

public class Example {
```

```
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // starts a new thread
    }
}
```

2. sleep() Method

The **sleep()** method temporarily pauses the execution of the current thread for a specified amount of time, allowing other threads to execute during that period. It is a **static method** of the Thread class and can throw an InterruptedException if another thread interrupts the sleeping thread.The syntax of the method is Thread.sleep(long milliseconds), and an overloaded version also accepts nanoseconds. When a thread goes into sleep mode, it does not release any locks it holds. This method is commonly used for timing-based tasks, controlling execution speed, or simulating real-time delays.

Example:

```
try {
    Thread.sleep(1000); // sleeps for 1 second
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

3. join() Method

The **join()** method allows one thread to wait for the completion of another. When a thread calls join() on another thread, it pauses its own execution until the other thread has finished running. This is especially useful when you want to ensure that a background thread has completed its work before proceeding further in the program.

There are overloaded versions such as join(long millis) and join(long millis, int nanos) that allow you to specify how long the calling thread should wait before continuing execution, even if the other thread has not finished.

Example:

```
class ExampleJoin extends Thread {
    public void run() {
        System.out.println("Thread running...");
    }

    public static void main(String[] args) throws InterruptedException {
        ExampleJoin t1 = new ExampleJoin();
        t1.start();
```

```
        t1.join(); // waits for t1 to finish
        System.out.println("Thread has finished execution.");
    }
}
```

4. interrupt(), isInterrupted(), and interrupted() Methods

Java provides the **interrupt()** method to signal a thread that it should stop what it is doing and terminate gracefully. However, interrupting a thread does not forcibly stop it merely sets an internal interrupt flag that the thread can check periodically.

The **isInterrupted()** method returns whether a thread has been interrupted, while the static **interrupted()** method checks and **clears** the interrupt status of the currently executing thread. These methods enable cooperative interruption, meaning the thread itself decides when and how to respond to the interruption signal.

Example:

```
class InterruptDemo extends Thread {
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println(i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted!");
        }
    }
    public static void main(String[] args) {
        InterruptDemo t = new InterruptDemo();
        t.start();
        t.interrupt(); // send interrupt signal
    }
}
```

5. setName() and getName() Methods

Each thread in Java has a name, which can be retrieved or modified using the getName() and setName() methods. By default, threads are assigned names like Thread-0, Thread-1, etc. Naming threads helps identify them easily during debugging or logging in complex applications.

Example:

```
Thread t1 = new Thread();
t1.setName("WorkerThread");
System.out.println("Thread Name: " + t1.getName());
```

6. setPriority() and getPriority() Methods

Threads in Java have a priority level that helps the thread scheduler decide the order of execution. The priority can be set using the **setPriority()** method and retrieved using the **getPriority()** method.

Thread priorities range between **1 (MIN_PRIORITY)** and **10 (MAX_PRIORITY)**, with **5 (NORM_PRIORITY)** being the default. Although higher priority threads are more likely to run first, this behavior is platform-dependent and not guaranteed.

Example:

```
Thread t1 = new Thread();
t1.setPriority(Thread.MAX_PRIORITY);
System.out.println("Thread Priority: " + t1.getPriority());
```

7. setDaemon() and isDaemon() Methods

A **daemon thread** is a background thread that performs tasks such as garbage collection or background maintenance. The **setDaemon(true)** method marks a thread as a daemon thread, and this must be done before starting the thread. The **isDaemon()** method returns whether a thread is a daemon or not.

Daemon threads automatically terminate when all user threads (non-daemon threads) have finished execution, meaning they do not prevent the JVM from exiting.

Example:

```
Thread daemon = new Thread(() -> {
    while (true) {
        System.out.println("Daemon thread running...");
    }
});
daemon.setDaemon(true);
daemon.start();
```

8. yield() Method

The **yield()** method is a static method that causes the currently executing thread to temporarily pause and allow other threads of equal or higher priority to execute. It acts as a hint to the thread scheduler that the current thread is willing to yield its current use of the CPU.

However, there is no guarantee that the scheduler will honor this hint; therefore, yield() should not be used for controlling execution order in critical systems.

Example:

```
Thread.yield(); // Suggests scheduler to switch to another thread
```

9. isAlive() Method

The **isAlive()** method checks whether a thread has been started and has not yet finished executing. It returns true if the thread is still running and false if it has not started or has already terminated.

This method is often used to monitor thread status during program execution.

Example:

```
Thread t1 = new Thread(() -> System.out.println("Running..."));
System.out.println("Is alive before start: " + t1.isAlive());
t1.start();
System.out.println("Is alive after start: " + t1.isAlive());
```

10. getState() Method

The **getState()** method returns the current state of a thread, represented by the Thread.State enumeration. Possible states include **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING**, and **TERMINATED**. This method is primarily used for debugging and monitoring thread life cycles.

Example:

```
Thread t = new Thread();
System.out.println("Thread state: " + t.getState());
```

11. setUncaughtExceptionHandler() Method

If a thread terminates unexpectedly due to an exception that is not caught, the **setUncaughtExceptionHandler()** method allows you to handle that situation gracefully.

You can use it to log errors or perform cleanup operations when a thread encounters an unhandled exception.

Example:

```
Thread t = new Thread(() -> { throw new RuntimeException("Error!"); });
t.setUncaughtExceptionHandler((thread, ex) ->
    System.out.println("Exception in thread " + thread.getName() + ": " + ex));
t.start();
```

Exceptions

In Java, an **exception** is an event that disrupts the normal flow of a program's execution. When an exception occurs, the program terminates abnormally unless the exception is properly handled. This interruption happens because something unexpected has occurred—such as invalid user input, missing files, or system failures. Java provides a robust exception handling mechanism to ensure that the program can respond to these problems gracefully and continue running instead of crashing abruptly.

Why Exceptions Occur

Exceptions can occur for several reasons during the execution of a program. Some of the common causes include:

- A user entering invalid data, such as dividing a number by zero.
- Trying to access a file that does not exist on the system.
- A network connection being lost during communication.
- The Java Virtual Machine (JVM) running out of memory or other system resources.

Some exceptions arise due to **user errors**, some due to **programming mistakes**, and others because of **hardware or resource failures**. Java helps programmers anticipate and handle these situations effectively using its exception handling framework.

Categories of Exceptions in Java

Exceptions in Java are divided into three major categories:

1. **Checked Exceptions**
2. **Unchecked Exceptions**
3. **Errors**

Each category plays a unique role in handling different types of runtime issues.

Checked Exceptions

Checked exceptions are those that the Java compiler checks at compile time. This means if your code can potentially throw a checked exception, the compiler forces you to handle it using a try-catch block or declare it with the throws keyword. These exceptions usually occur due to external factors such as file access, database connection issues, or network errors.

Example of Checked Exception:

```
import java.io.File;
import java.io.FileReader;
import java.io.FileNotFoundException;

public class FileDemo {
  public static void main(String[] args) {
    try {
      File file = new File("E://data.txt");
      FileReader reader = new FileReader(file);
    } catch (FileNotFoundException e) {
      System.out.println("File not found: " + e.getMessage());
    }
  }
}
```

In this example, if the file **E://data.txt** does not exist, a FileNotFoundException is thrown. The compiler ensures that this situation is handled properly before the program runs.

Unchecked Exceptions

Unchecked exceptions, also called **runtime exceptions**, occur while the program is running. These exceptions are not checked during compilation because they usually occur due to logical programming errors such as dividing by zero or accessing an invalid array index. These exceptions are subclasses of RuntimeException.

Example of Unchecked Exception:

```
public class UncheckedDemo {
  public static void main(String[] args) {
    int[] numbers = {10, 20, 30};
    System.out.println(numbers[5]);
  }
}
```

In this example, the program tries to access the 6th element of the array, which doesn't exist. As a result, an ArrayIndexOutOfBoundsException is thrown at runtime.

Errors

Errors are serious problems that occur beyond the control of the programmer or user. These problems often indicate that the Java Virtual Machine is in a state from which recovery is impossible. Examples include running out of memory or system-level failures. Errors are subclasses of the Error class, and programs typically do not handle them because they represent conditions that are not meant to be recovered from.

Example of Error:

```
public class ErrorDemo {
  public static void main(String[] args) {
    recursiveMethod();
  }
  public static void recursiveMethod() {
    recursiveMethod();
  }
}
```

In this example, the recursive method keeps calling itself indefinitely, eventually leading to a **StackOverflowError**.

Java Exception Hierarchy

In Java, all exception classes are derived from the Throwable class. The Throwable class has two direct subclasses:

1. **Exception**
2. **Error**

Under the Exception class, there are two important subclasses:

- IOException (for checked exceptions)
- RuntimeException (for unchecked exceptions)

This hierarchy allows Java to classify exceptions clearly and handle them effectively.

Catching Exceptions Using Try and Catch

In Java, exceptions can be caught using a **try-catch** block. The code that might throw an exception is placed inside the try block, and the handling code goes inside the catch block.

Example:

```
public class ExceptionExample {
```

```
  public static void main(String[] args) {
    try {
      int[] arr = new int[3];
      System.out.println(arr[4]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception caught: " + e);
    }
    System.out.println("Program continues...");
  }
}
```

Output:

```
Exception caught: java.lang.ArrayIndexOutOfBoundsException: 4
Program continues...
```

Multiple Catch Blocks

Java allows multiple catch blocks to handle different types of exceptions separately. This helps programmers handle each type of problem in a specific way.

Example:

```
try {
  int a = 10 / 0;
  int[] arr = new int[3];
  System.out.println(arr[4]);
} catch (ArithmeticException e) {
  System.out.println("Arithmetic error: " + e);
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("Array index error: " + e);
}
```

In this example, the program first throws an ArithmeticException, which is caught by the first catch block.

The Finally Block

The **finally** block is used to execute important cleanup code such as closing files or releasing resources. It always executes, whether an exception occurs or not.

Example:

```
public class FinallyDemo {
```

```
  public static void main(String[] args) {
    try {
      int num = 10 / 0;
    } catch (ArithmeticException e) {
      System.out.println("Caught exception: " + e);
    } finally {
      System.out.println("This block always executes.");
    }
  }
}
```

Output:

Caught exception: java.lang.ArithmeticException: / by zero
This block always executes.

Try-with-Resources in Java

Starting from Java 7, a new feature called **try-with-resources** automatically closes resources such as files or network connections when the try block ends. The resource used must implement the AutoCloseable interface.

Example:

```
import java.io.FileReader;
import java.io.IOException;

public class TryWithResourcesDemo {
  public static void main(String[] args) {
    try (FileReader fr = new FileReader("E://file.txt")) {
      char[] a = new char[50];
      fr.read(a);
      for (char c : a) {
        System.out.print(c);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }}}
```

This approach automatically closes the FileReader object after use, eliminating the need for a finally block.

User-Defined Exceptions

Java also allows programmers to define their own exceptions by extending the Exception class. These are known as **user-defined exceptions**.

Example:

```
class InsufficientFundsException extends Exception {
  private double amount;
  public InsufficientFundsException(double amount) {
    this.amount = amount;
  }
  public double getAmount() {
    return amount;
  }
}

class BankAccount {
  private double balance;
  public void deposit(double amount) {
    balance += amount;
  }
  public void withdraw(double amount) throws InsufficientFundsException {
    if (amount > balance) {
      throw new InsufficientFundsException(amount - balance);
    }
    balance -= amount;
  }
}

public class BankDemo {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    account.deposit(500);
    try {
      account.withdraw(700);
    } catch (InsufficientFundsException e) {
      System.out.println("Sorry, you are short by $" + e.getAmount());
    }
  }
}
```

Output:

```
Sorry, you are short by $200.0
```

2.4 Inter-thread Communication

Inter-thread communication refers to the process through which multiple threads exchange information and coordinate their actions during execution. This is an essential part of multithreading, as threads often need to work together rather than operate completely independently. For example, one thread might produce data, while another consumes it. Java provides built-in mechanisms for thread communication using the methods **wait()**, **notify()**, and **notifyAll()**, which are defined in the **Object** class.

These methods help threads to **pause and resume execution** based on specific conditions, ensuring synchronization and preventing conflicts during data sharing.

Why Inter-Thread Communication is Needed

When multiple threads are working on a shared resource, they often need to communicate to coordinate their tasks. For instance:

- A **producer thread** might generate data and store it in a shared memory buffer.
- A **consumer thread** might retrieve data from the same buffer.

If both threads try to access the buffer simultaneously, problems such as data inconsistency may arise. To handle such scenarios, Java uses inter-thread communication methods to ensure that one thread waits while the other completes its operation.

Methods Used for Inter-Thread Communication

Java provides three key methods to achieve inter-thread communication. All three methods belong to the Object class and must be used within a synchronized context.

1. **wait()**
   - Causes the current thread to pause execution until another thread calls notify() or notifyAll() on the same object.
   - It releases the lock on the object temporarily while waiting.
2. **notify()**
   - Wakes up one thread that is waiting on the same object.
   - The awakened thread will not run immediately; it must reacquire the lock before continuing.

3. notifyAll()

- Wakes up **all threads** that are waiting on the object's monitor.
- One of them will eventually acquire the lock and proceed.

These methods can **only be called inside synchronized methods or synchronized blocks**; otherwise, Java throws an IllegalMonitorStateException.

Example

Let us consider an example where two threads one asking questions and another answering them communicate using wait() and notify().

```
class Chat {
  boolean flag = false;

  public synchronized void question(String msg) {
    if (flag) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println(msg);
    flag = true;
    notify();
  }

  public synchronized void answer(String msg) {
    if (!flag) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println(msg);
    flag = false;
    notify();
  }
}
class QuestionThread implements Runnable {
  Chat chat;
  String[] questions = { "Hi", "How are you?", "What are you doing?" };

  public QuestionThread(Chat chat) {
    this.chat = chat;
    new Thread(this, "QuestionThread").start();
  }
```

```
  public void run() {
    for (String q : questions) {
      chat.question(q);
    }
  }
}

class AnswerThread implements Runnable {
  Chat chat;
  String[] answers = { "Hello", "I'm fine, thank you!", "Just studying Java threads." };

  public AnswerThread(Chat chat) {
    this.chat = chat;
    new Thread(this, "AnswerThread").start();
  }

  public void run() {
    for (String a : answers) {
      chat.answer(a);
    }
  }
}

public class InterThreadDemo {
  public static void main(String[] args) {
    Chat chat = new Chat();
    new QuestionThread(chat);
    new AnswerThread(chat);
  }
}
```

Output:

```
Hi
Hello
How are you?
I'm fine, thank you!
What are you doing?
Just studying Java threads.
```

In this program:

- The QuestionThread and AnswerThread share the same Chat object.
- The wait() method pauses a thread until it is notified.
- The notify() method wakes up the other waiting thread.

- The flag variable ensures that threads execute alternately.

Another Example: Producer-Consumer Problem

A classic use of inter-thread communication is the **Producer-Consumer Problem**, where one thread (the producer) generates data, and another thread (the consumer) processes it.

```
class SharedBuffer {
  private int data;
  private boolean available = false;

  public synchronized void produce(int value) {
    while (available) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    data = value;
    available = true;
    System.out.println("Produced: " + data);
    notify();
  }

  public synchronized void consume() {
    while (!available) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println("Consumed: " + data);
    available = false;
    notify();
  }
}

class Producer implements Runnable {
  SharedBuffer buffer;
  public Producer(SharedBuffer buffer) {
    this.buffer = buffer;
    new Thread(this, "Producer").start();
  }
```

```
  public void run() {
    for (int i = 1; i <= 5; i++) {
      buffer.produce(i);
      try {
        Thread.sleep(500);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }}}

class Consumer implements Runnable {
  SharedBuffer buffer;
  public Consumer(SharedBuffer buffer) {
    this.buffer = buffer;
    new Thread(this, "Consumer").start();
  }
  public void run() {
    for (int i = 1; i <= 5; i++) {
      buffer.consume();
      try {
        Thread.sleep(800);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}

public class ProducerConsumerDemo {
  public static void main(String[] args) {
    SharedBuffer buffer = new SharedBuffer();
    new Producer(buffer);
    new Consumer(buffer);
  }
}
```

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
```

Consumed: 4
Produced: 5
Consumed: 5

In this example:

- The **producer** thread generates data and calls notify() after producing.
- The **consumer** waits for the producer to produce data using wait().
- The while loops ensure proper waiting conditions and prevent unexpected behavior if multiple threads are awakened.

Important Rules for Inter-Thread Communication

1. The methods wait(), notify(), and notifyAll() can only be called inside a **synchronized block**.
2. When a thread calls wait(), it temporarily releases the lock on the object.
3. The waiting thread must be awakened by another thread that calls notify() or notifyAll() on the same object.
4. If multiple threads are waiting, notify() wakes up one random thread, while notifyAll() wakes all of them.
5. The awakened thread must re-acquire the object's lock before resuming execution.

Advantages of Inter-Thread Communication

- Prevents **busy waiting** where a thread continuously checks for a condition.
- Improves **efficiency** by letting threads sleep until they are needed.
- Ensures **proper synchronization** when multiple threads share resources.
- Simplifies **coordination** between dependent threads.

2.5 Synchronization

Synchronization in Java is a mechanism that controls the access of multiple threads to shared resources. It ensures that only one thread can access a shared resource at any given time, preventing issues like **data inconsistency**, **race conditions**, and **thread interference**.

When multiple threads operate on the same object without synchronization, they may corrupt shared data because thread scheduling by the JVM and operating system is unpredictable. Synchronization ensures **atomicity (one-at-a-time execution)** and **visibility (updated data is visible to all threads)**.

Why Synchronization is Needed

Let's imagine two people (threads) updating the same bank account balance at the same time.If both withdraw money simultaneously without synchronization, they might both read the same balance before either writes the new one leading to an incorrect final balance.

Synchronization prevents such conflicts by **locking the shared resource** so only one thread can modify it at a time.

Types of Synchronization in Java

Java provides **two main types of synchronization**:

1. **Process Synchronization**
2. **Thread Synchronization**

1. Process Synchronization

Process Synchronization is used to coordinate the execution of multiple processes (or independent execution units) that share a resource such as a file, database, or memory location. The goal is to maintain **data integrity** and **proper order of execution**.

In Java, synchronization is achieved using **synchronized methods** or **synchronized blocks**.

Example: Bank Account System

In this program, two threads perform deposit and withdrawal operations on a shared bank account.

```
class BankAccount {
   private int balance = 1000; // Shared resource

   // Synchronized deposit method
   public synchronized void deposit(int amount) {
      balance += amount;
      System.out.println("Deposited: " + amount + ", Balance: " + balance);
   }

   // Synchronized withdraw method
   public synchronized void withdraw(int amount) {
      if (balance >= amount) {
         balance -= amount;
         System.out.println("Withdrawn: " + amount + ", Balance: " + balance);
      } else {
```

```
            System.out.println("Insufficient balance to withdraw: " + amount);
        }
    }

    public int getBalance() {
        return balance;
    }
}

public class ProcessSyncDemo {
    public static void main(String[] args) throws InterruptedException {
        BankAccount account = new BankAccount(); // Shared resource

        // Thread 1 deposits money
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                account.deposit(200);
                try { Thread.sleep(50); } catch (InterruptedException e) {}
            }
        });

        // Thread 2 withdraws money
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                account.withdraw(150);
                try { Thread.sleep(70); } catch (InterruptedException e) {}
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Final Balance: " + account.getBalance());
    }
}
```

Output:

```
Deposited: 200, Balance: 1200
Withdrawn: 150, Balance: 1050
Deposited: 200, Balance: 1250
Withdrawn: 150, Balance: 1100
Deposited: 200, Balance: 1300
Withdrawn: 150, Balance: 1150
```

Final Balance: 1150

Explanation:

- The deposit() and withdraw() methods are synchronized.
- Only one thread can modify the balance at a time.
- This prevents data corruption and maintains an accurate final balance.

2. Thread Synchronization

Thread Synchronization ensures that threads in the same process coordinate their work when accessing shared resources.

There are two types of thread synchronization:

1. **Mutual Exclusion**
2. **Cooperation (Inter-thread communication)**

A. Mutual Exclusion

Mutual exclusion ensures that only one thread executes a block of code (critical section) at a time. This prevents simultaneous modification of shared data.

Java provides **three ways** to achieve mutual exclusion:

1. **Synchronized Methods**
2. **Synchronized Blocks**
3. **Static Synchronization**

(i) Synchronized Method Example

```
class Counter {
   private int count = 0;

   // Synchronized method ensures atomic update
   public synchronized void increment() {
      count++;
   }

   public synchronized int getCount() {
      return count;
   }
}
public class SyncMethodDemo {
```

```
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> { for (int i = 0; i < 1000; i++) counter.increment(); });
        Thread t2 = new Thread(() -> { for (int i = 0; i < 1000; i++) counter.increment(); });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Counter: " + counter.getCount());
    }
}
```

Output:

Final Counter: 2000

Explanation: The synchronized keyword ensures that only one thread can execute increment() at a time, preventing race conditions.

(ii) Synchronized Block Example

Synchronized blocks give more control by allowing synchronization of only a part of a method.

```
class SharedData {
    private int data = 0;

    public void updateData() {
        synchronized (this) {
            data++;
            System.out.println(Thread.currentThread().getName() + " updated data to: " + data);
        }
    }
}

public class SyncBlockDemo {
    public static void main(String[] args) throws InterruptedException {
        SharedData shared = new SharedData();

        Thread t1 = new Thread(shared::updateData, "Thread-1");
        Thread t2 = new Thread(shared::updateData, "Thread-2");
        Thread t3 = new Thread(shared::updateData, "Thread-3");
```

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

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

Output:

Thread-1 updated data to: 1
Thread-2 updated data to: 2
Thread-3 updated data to: 3

Explanation: The synchronized block allows only one thread to execute the critical section, ensuring consistent data updates.

(iii) Static Synchronization Example

Static synchronization locks on the **class object** rather than an instance. This means that all objects of that class share the same lock.

```
class SharedPrinter {
    public static synchronized void printMessage(String msg) {
        System.out.println("Printing: " + msg);
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        System.out.println(msg + " printed successfully.");
    }
}

public class StaticSyncDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> SharedPrinter.printMessage("Document A"));
        Thread t2 = new Thread(() -> SharedPrinter.printMessage("Document B"));

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

Output:

Printing: Document A

Document A printed successfully.
Printing: Document B
Document B printed successfully.

Explanation: printMessage() is a static synchronized method, so only one thread at a time can execute it, even if they belong to different objects.

B. Cooperation (Inter-Thread Communication)

Cooperation allows threads to communicate with each other using the methods:

1. wait()
2. notify()
3. notifyAll()

These methods are defined in the **Object** class and can only be called inside synchronized blocks.

Example: Producer-Consumer Problem

```
class DataBuffer {
    private int data;
    private boolean available = false;

    public synchronized void produce(int value) {
        while (available) {
            try { wait(); } catch (InterruptedException e) {}
        }
        data = value;
        available = true;
        System.out.println("Produced: " + value);
        notify();
    }

    public synchronized void consume() {
        while (!available) {
            try { wait(); } catch (InterruptedException e) {}
        }
        System.out.println("Consumed: " + data);
        available = false;
        notify();
    }
}

public class ProducerConsumerDemo {
```

```
    public static void main(String[] args) {
        DataBuffer buffer = new DataBuffer();

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) buffer.produce(i);
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) buffer.consume();
        });

        producer.start();
        consumer.start();
    }
}
```

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

Explanation:

- The producer waits if the buffer is full.
- The consumer waits if the buffer is empty.
- The notify() method wakes the waiting thread once the state changes.

3. Volatile Keyword (Related to Synchronization)

The volatile keyword ensures **visibility** of changes made to a variable across threads. It tells the JVM that the variable is stored in main memory, not in a thread's local cache.

Example:

```
class TaskRunner {
    private volatile boolean running = true;
```

```
    public void start() {
        new Thread(() -> {
            while (running) {
                System.out.println("Running task...");
                try { Thread.sleep(200); } catch (InterruptedException e) {}
            }
            System.out.println("Stopped.");
        }).start();
    }

    public void stop() {
        running = false;
    }
}
public class VolatileDemo {
    public static void main(String[] args) throws InterruptedException {
        TaskRunner runner = new TaskRunner();
        runner.start();

        Thread.sleep(600);
        runner.stop();
    }
}
```

Output:

Running task...
Running task...
Running task...
Stopped.

Explanation: When running is marked volatile, its updated value is visible to all threads immediately, ensuring the task stops correctly.

2.6 JDBC Architecture and Drivers

JDBC Architecture

JDBC (Java Database Connectivity) is a Java API that allows applications to connect and interact with various relational databases such as **MySQL, Oracle, SQL Server, and PostgreSQL**. It provides a standard interface through which Java programs can perform database operations like connecting to the database, executing queries, updating records, and

processing results. JDBC essentially acts as a bridge between Java applications and database management systems, ensuring smooth communication between them.

Need for JDBC

Before JDBC was introduced, developers used the **ODBC (Open Database Connectivity)** API to communicate with databases. However, ODBC was based on the **C language**, making it **platform-dependent** and not fully compatible with Java's portability features. To overcome these limitations, **Sun Microsystems** introduced **JDBC**, a **pure Java-based API** that provides a simple, secure, and platform-independent way to connect to databases. JDBC allows Java programs to work seamlessly with any database without worrying about the underlying system architecture.

Why JDBC is Important

JDBC plays a vital role in Java application development as it provides a unified method to interact with databases. It enables Java developers to:

- Establish connections with databases.
- Execute SQL commands such as queries and updates.
- Retrieve and manipulate data.
- Handle exceptions related to database operations.

This makes JDBC an essential component of enterprise-level applications like **banking systems**, **e-commerce platforms**, **reservation systems**, and **online educational portals**, where dynamic data access and updates are crucial.

JDBC Architecture Overview

The JDBC architecture defines how a Java application communicates with a database. It consists of several layers and components that work together to establish a connection, send queries, and process the results. These components form the foundation of any database-driven Java application.

Components of JDBC Architecture

1. Application Layer: The application layer represents the Java application or servlet that interacts with the database. It sends SQL queries to the database through the JDBC API and displays the results to the user.

2. JDBC API: The JDBC API provides a set of **interfaces and classes** that define methods for database connectivity and query execution. Some of the important interfaces include **Driver**, **Connection**, **Statement**, **PreparedStatement**, **CallableStatement**, and **ResultSet**. Key classes include **DriverManager**, **Blob**, and **Clob**. Together, they simplify database operations such as inserting, updating, deleting, and retrieving data.

3. DriverManager: The DriverManager class acts as a mediator between the Java application and the database drivers. It loads and manages database-specific drivers and establishes a connection to the database using the appropriate driver.

4. JDBC Driver: JDBC drivers are responsible for converting Java API calls into database-specific queries. They enable communication between the Java program and the database by translating JDBC commands into the format the database understands.

5. Database Layer: This layer represents the actual **database system**, such as MySQL or Oracle, where data is stored and managed. It executes SQL commands received from the driver and sends results back to the application.

JDBC Processing Models

The JDBC architecture supports two main models for database access: **two-tier architecture** and **three-tier architecture**. Each model represents a different way in which Java applications interact with databases.

1. Two-Tier Architecture

In the **two-tier architecture**, a Java application communicates **directly** with the database using a JDBC driver. The client sends SQL queries through JDBC, and the database responds with results. This model is simple and suitable for standalone desktop applications or small-scale systems.

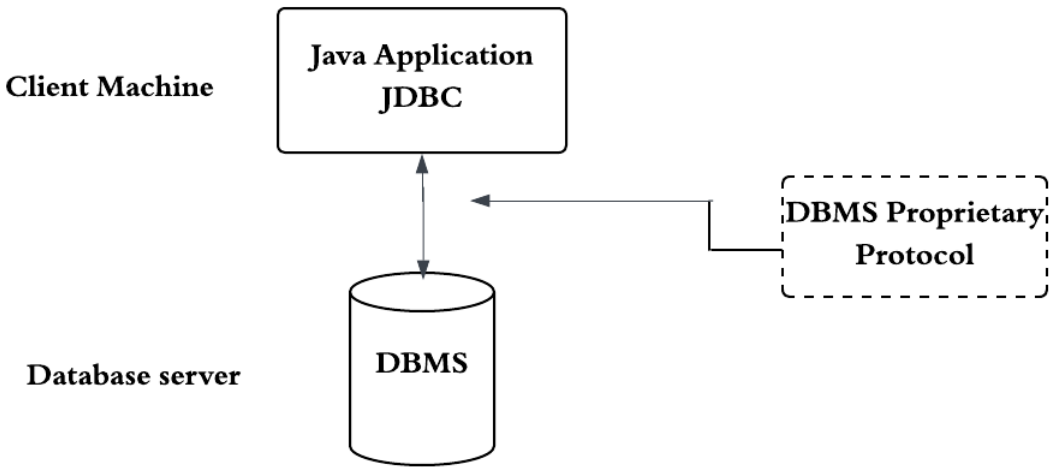


Fig. 2.3 Two-Tier Architecture.

Structure:

Client Application (Java) → JDBC Driver → Database

Example: A Java program that connects directly to a local MySQL database to retrieve employee details.

2. Three-Tier Architecture

The **three-tier architecture** adds an intermediate layer known as the **application server** or **middleware** between the client and the database. The client sends a request to the application server, which processes it and interacts with the database using JDBC. The database results are then passed back to the client through the server.

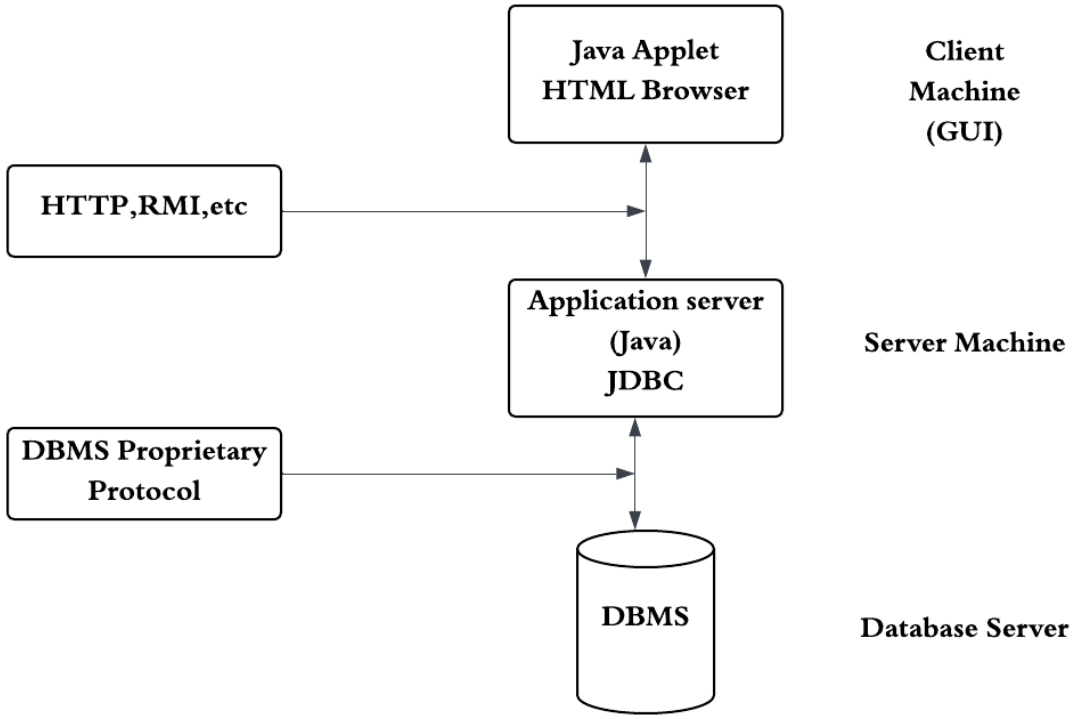


Fig. 2.4 Three-Tier Architecture.

Structure:

Client Application → Application Server → JDBC Driver → Database

This architecture is widely used in enterprise applications because it offers better **security**, **scalability**, and **maintainability**. The middle tier can also handle business logic, reducing the load on the client.

JDBC Drivers

JDBC drivers are essential components that handle the low-level communication between the Java program and the database. They translate Java JDBC calls into database-specific protocols. JDBC defines four types of drivers, each with its own method of database communication.

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

This driver converts JDBC calls into ODBC calls using native libraries. It is **partly written in C** and depends on ODBC, making it platform-dependent. It is now deprecated and no longer supported in modern Java versions.

2. Type-2 Driver (Native-API Driver)

This driver uses the database's native API (written in C or C++) to convert JDBC calls. While it offers better performance than Type-1, it still requires database-specific native libraries on the client machine.

3. Type-3 Driver (Network Protocol Driver)

The Type-3 driver communicates with the database through a **middleware server**. The middleware translates JDBC calls into database-specific calls, making this driver platform-independent and suitable for large enterprise systems.

4. Type-4 Driver (Thin Driver)

The Type-4 driver is a **pure Java driver** that converts JDBC calls directly into the database's native protocol. It requires no native libraries and provides the best performance. This driver is the **most commonly used** in modern applications.

JDBC Classes and Interfaces

The JDBC API provides several key classes and interfaces that simplify database operations:

Class/Interface	Description
DriverManager	Manages JDBC drivers and establishes database connections.

Connection	Represents a connection session with a specific database.
Statement	Used for executing SQL queries.
PreparedStatement	Used for precompiled SQL queries with parameters.
CallableStatement	Used to call stored procedures.
ResultSet	Holds data returned by a query and allows row-by-row navigation.
SQLException	Handles errors related to database access.

Table. 2.1 JDBC Classes and Interfaces.

Features of JDBC

The **JDBC 4.0** version introduced several improvements to simplify database connectivity and enhance performance.

1. **Automatic Driver Loading** – The driver is automatically loaded when using DriverManager.getConnection(), removing the need to call Class.forName().
2. **Enhanced Exception Handling** – New subclasses of SQLException were introduced for more precise error handling.
3. **XML Data Type Support** – The SQLXML interface allows applications to store and process XML data easily.
4. **Improved Connection Management** – Connection factories make it easier to manage multiple database connections.
5. **RowID Interface Support** – Provides support for databases that use ROWID as a unique identifier.
6. **National Character Set Support** – New data types like NCHAR and NVARCHAR were added for Unicode data.
7. **BLOB and CLOB Enhancements** – Methods such as createBlob() and createClob() simplify handling large binary and text data.

Advantages of JDBC Architecture

JDBC offers numerous advantages that make it an essential component in Java-based database programming:

1. **Platform Independence** : Because JDBC is written in Java, it runs on any platform with a Java Virtual Machine (JVM).

2. **Database Independence** : Java applications can connect to any database simply by changing the driver.
3. **Centralized Maintenance** : JDBC drivers manage all connections centrally, reducing client-side configuration.
4. **Enhanced Security** : JDBC provides secure database communication through controlled access.
5. **Comprehensive SQL Support** : It supports SQL queries, stored procedures, and transactions.
6. **XML and Large Object Handling** : JDBC supports XML data and large objects like BLOBs and CLOBs.
7. **Integration with Java EE** : JDBC works seamlessly with servlets, JSP, and frameworks like Hibernate and Spring.

Applications of JDBC

JDBC is used in nearly every Java application that interacts with databases. Some common real-world applications include:

- **Banking Systems** – Managing account transactions and customer records.
- **E-commerce Platforms** – Storing product information and handling orders.
- **Online Ticket Booking Systems** – Checking and updating seat availability dynamically.
- **Educational Portals** – Managing student records and exam results.
- **Government Portals** – Handling citizen data and service requests.

For example, when you book a train ticket online, JDBC helps the application connect to the database, check seat availability, reserve the ticket, and update the record in real time.

2.7 CRUD Operations

CRUD stands for Create Read Update and Delete. These four operations form the core of data management in most applications. Create refers to adding new records to storage. Read means retrieving existing records. Update covers modifying records that already exist. Delete removes records from storage. In Java these operations are commonly implemented with SQL statements when using a database or with collection and object manipulation when working in memory. Below you will find a clear explanation of each operation followed by code examples and sample output.

Create operation

The Create operation adds a new entity to storage. In Java you usually construct an object that models the entity and then persist it. When working with databases this is done with an SQL

INSERT statement or with an ORM method. When working in memory you add the object to a collection.

Example using an in memory list to add students

```
import java.util.ArrayList;
import java.util.List;

class Student {
    private int id;
    private String name;
    private String contact;

    public Student(int id, String name, String contact) {
        this.id = id;
        this.name = name;
        this.contact = contact;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public String getContact() { return contact; }

    public void setName(String name) { this.name = name; }
    public void setContact(String contact) { this.contact = contact; }
}

class StudentRepository {
    private final List<Student> list = new ArrayList<>();

    public void create(Student s) {
        list.add(s);
    }

    public List<Student> all() {
        return list;
    }
}

public class CreateDemo {
    public static void main(String[] args) {
        StudentRepository repo = new StudentRepository();
        repo.create(new Student(1, "Alice", "1234567890"));
        repo.create(new Student(2, "Bob", "0987654321"));
```

```
        for (Student s : repo.all()) {
            System.out.println(s.getId() + " " + s.getName() + " " + s.getContact());
        }
    }
}
```

Sample output

```
1 Alice 1234567890
2 Bob 0987654321
```

Read operation

The Read operation retrieves data. In Java you can perform SQL SELECT statements with JDBC or use repository methods that return objects. Filtering and pagination are common when working with large data sets.

Example retrieving a student by id from the in memory repository

```
class StudentRepository {
    // methods from previous example omitted for brevity

    public Student findById(int id) {
        for (Student s : list) {
            if (s.getId() == id) return s;
        }
        return null;
    }
}

public class ReadDemo {
    public static void main(String[] args) {
        StudentRepository repo = new StudentRepository();
        repo.create(new Student(1, "Alice", "1234567890"));

        Student found = repo.findById(1);
        if (found != null) {
            System.out.println("Found " + found.getName() + " contact " + found.getContact());
        } else {
            System.out.println("Student not found");
        }
    }
}
```

Sample output

Found Alice contact 1234567890

Update operation

Update changes an existing record. In a database this is done with an SQL UPDATE statement. When using an ORM you typically load the entity modify its fields and save it. When working with collections you find the object and modify its properties.

Example updating a student record

```
class StudentRepository {
   // earlier methods omitted

   public boolean update(int id, String newName, String newContact) {
      Student s = findById(id);
      if (s == null) return false;
      s.setName(newName);
      s.setContact(newContact);
      return true;
   }
}

public class UpdateDemo {
   public static void main(String[] args) {
      StudentRepository repo = new StudentRepository();
      repo.create(new Student(2, "Bob", "0987654321"));

      boolean updated = repo.update(2, "Bob Smith", "5555555555");
      System.out.println("Updated " + updated);

      Student s = repo.findById(2);
      System.out.println(s.getId() + " " + s.getName() + " " + s.getContact());
   }
}
```

Sample output

Updated true
2 Bob Smith 5555555555

Delete operation

Delete removes a record from storage. In many systems deletion is final and irreversible. In other systems soft deletes are used where a record is marked as inactive instead of being physically removed.

Example hard delete from the in memory list

```
class StudentRepository {
   // earlier methods omitted

   public boolean delete(int id) {
      for (int i = 0; i < list.size(); i++) {
         if (list.get(i).getId() == id) {
            list.remove(i);
            return true;
         }
      }
      return false;
   }
}

public class DeleteDemo {
   public static void main(String[] args) {
      StudentRepository repo = new StudentRepository();
      repo.create(new Student(1, "Alice", "1234567890"));
      repo.create(new Student(2, "Bob", "0987654321"));

      boolean deleted = repo.delete(1);
      System.out.println("Deleted " + deleted);

      for (Student s : repo.all()) {
         System.out.println(s.getId() + " " + s.getName());
      }
   }
}
```

Sample output

```
Deleted true
2 Bob
```

Soft delete and archival

Soft delete means marking a record as removed while keeping it in storage so it can be restored or audited later. A common pattern is to add a boolean field called active or deletedAt timestamp and filter queries accordingly. Data archival moves older records to a separate storage system to improve performance and meet retention requirements.

Student management example with a menu driven program

Below is a compact example that illustrates create read update and delete operations together using a simple console menu. The repository is kept in memory for clarity. In production you would replace repository methods with JDBC or an ORM.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class StudentApp {
   private static final List<Student> students = new ArrayList<>();

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      students.add(new Student(1, "Alice", "1234567890"));
      students.add(new Student(2, "Bob", "0987654321"));

      boolean running = true;
      while (running) {
         System.out.println("1 Add 2 Search 3 Update 4 Delete 5 Display 6 Exit");
         int choice = sc.nextInt();
         switch (choice) {
            case 1:
               System.out.println("Enter id name contact");
               students.add(new Student(sc.nextInt(), sc.next(), sc.next()));
               break;
            case 2:
               System.out.println("Enter id to search");
               Student f = findById(sc.nextInt());
              System.out.println(f == null ? "Not found" : f.getId() + " " + f.getName());
               break;
            case 3:
               System.out.println("Enter id newName newContact");
               int id = sc.nextInt(); String name = sc.next(); String contact = sc.next();
               Student s = findById(id);
                        if (s != null) { s.setName(name); s.setContact(contact);
                  System.out.println("Updated"); }
```

```
                else System.out.println("Not found");
                break;
            case 4:
                System.out.println("Enter id to delete");
                boolean removed = students.removeIf(st -> st.getId() == sc.nextInt());
                System.out.println(removed ? "Deleted" : "Not found");
                break;
            case 5:
                for (Student st : students) System.out.println(st.getId() + " " + st.getName() + " "
+ st.getContact());
                break;
            case 6:
                running = false;
                break;
            default:
                System.out.println("Invalid option");
        }
      }
      sc.close();
   }

   private static Student findById(int id) {
      for (Student s : students) if (s.getId() == id) return s;
      return null;
   }
}
```

A user interacting with the menu can add new students search by id update contact details delete a record and display all records. This example demonstrates how the four basic operations work in an application flow.

Input validation and edge cases

Always validate user input and handle edge cases. For create operations ensure unique ids and correct formats. For update and delete verify the record exists first. Handle invalid input with clear error messages and use try catch blocks around parsing and IO code. For persistence with a database manage transactions to keep data consistent and consider concurrency control when multiple clients may update the same records.

Connecting to Relational Databases

Connecting an application to a relational database is a fundamental task in most software systems. At its core it means establishing a reliable, secure, and performant channel through which the application can send SQL statements and receive results. The process involves driver software, connection strings or data sources, authentication and authorization,

transaction management, resource lifecycle handling, and operational concerns such as pooling, failover, and monitoring. Below is an in-depth explanation organized by topic so you can understand what happens at each stage and what to consider when building production-ready database connectivity.

Drivers and the client-side adapter

A database driver is the software component that translates your application's database API calls into the protocol the database server understands. In the Java world the most common example is a JDBC driver. Drivers can be pure-Java implementations or native libraries that call into vendor-specific C APIs. Choosing the right driver has an effect on portability, performance, and feature support. A modern pure-Java driver typically provides the best portability and is easier to deploy, while native drivers may offer optimizations for certain databases. Always obtain drivers from trusted sources and keep them updated to receive bug fixes and security patches.

Connection strings and configuration parameters

A connection string, DSN, or DataSource configuration tells the driver how to reach the database: the hostname or IP address, port, database name, user credentials, and optional parameters for timeouts, SSL, character set, and driver-specific tuning. Connection parameters control behavior such as socket timeouts, default schema, read-only mode, and whether to use SSL. Proper configuration is essential: defaults may be safe for development but not appropriate for production. Store credentials securely using environment variables, secret managers, or encrypted configuration rather than hard-coding them.

Authentication and authorization

Connecting is only the first step; the client must be authenticated and then authorized to perform requested operations. Authentication methods include username/password, Kerberos, TLS client certificates, and cloud IAM integration. After authentication the database enforces authorization through roles, grants, and privileges. Principle of least privilege is important: applications should connect with credentials that permit only the required operations. If multiple application components need different rights separate accounts or roles make auditing and containment easier.

Opening a connection: lifecycle and resource management

Creating a physical connection to a database often involves a network handshake, protocol negotiation, and resource allocation on the server. Because this operation can be relatively expensive, applications should avoid opening and closing connections per query. Instead applications should acquire connections from a managed pool and release them back when done. Whether you manage connections manually or via a framework, always ensure connections, result sets, and statements are closed in finally blocks or using structured constructs (try-with-resources in Java) to avoid resource leaks.

Connection pooling and why it matters

Connection pooling is a central performance and scalability technique. A pool maintains a set of open connections ready for use, allowing threads to borrow and return connections quickly. Pools can manage maximum size, minimum idle connections, validation queries, and eviction policies. Proper tuning prevents connection exhaustion and reduces latency. Oversized pools waste resources; undersized pools cause contention. Use battle-tested pool implementations like HikariCP, Apache DBCP, or the application server's built-in pool rather than writing your own.

Transactions and concurrency control

Transactions group multiple SQL statements into a single atomic unit: either all succeed or none do. Most relational databases use ACID semantics with isolation levels that trade off consistency and concurrency. Understanding isolation levels such as READ COMMITTED, REPEATABLE READ, and SERIALIZABLE is critical because they determine whether phenomena like dirty reads, non-repeatable reads, or phantom reads can occur. Applications should explicitly demarcate transactions, keep them short to reduce lock contention, and avoid user interaction while a transaction is open. For correctness under concurrent updates consider optimistic or pessimistic concurrency control patterns and use SQL features like SELECT ... FOR UPDATE or version columns where appropriate.

Prepared statements and parameterization

Prepared statements precompile SQL and allow parameter binding. They improve performance for repeated queries and, most importantly, prevent SQL injection by separating code from data. Always use parameterized queries rather than string concatenation to build SQL. Many drivers and ORMs support batching and statement caching that can further accelerate repeated operations.

Connection security and encryption

Production database connections should be encrypted in transit using TLS/SSL and should validate server certificates to prevent man-in-the-middle attacks. Use mutual TLS where possible for additional assurance. Avoid plaintext credentials on disk or logs. If the database is in the cloud, integrate with the provider's recommended secure connection methods and consider using managed identity or IAM roles instead of static credentials.

Error handling, retries, and idempotency

Network or transient database problems may cause operations to fail. Implement robust error handling that distinguishes transient errors (time-limited network blips, deadlock) from permanent ones (syntax errors, constraint violations). For transient failures consider retry policies with exponential backoff, but design operations to be idempotent or ensure that retries

do not produce incorrect duplicate effects. Use database-native retry guidance when available and be careful with retries inside transactions.

Performance considerations and tuning

Database access is often the bottleneck in applications. Reduce round-trips by using joins, appropriate indexing, and moving heavy logic into stored procedures or server-side queries where it makes sense. Use pagination for large result sets and streaming APIs or cursors if iterating over many rows. Monitor slow queries, add or adjust indexes carefully, and profile both query plans and application-level hotspots. Avoid fetching more columns or rows than needed and use connection pool monitoring to detect resource saturation.

Using ORMs and higher-level abstractions

Object-Relational Mapping frameworks such as Hibernate, JPA, or ActiveRecord simplify working with relational data by mapping tables to objects. ORMs reduce boilerplate but introduce their own considerations: lazy loading pitfalls, N+1 query problems, and caching semantics. Understand the SQL generated by your ORM and how its caching and session management interact with transactions. When performance or complex queries are critical, a mixed strategy using raw SQL or stored procedures for those paths can be beneficial.

Schema management, migrations, and versioning

Database schema evolves over time. Use a migration tool such as Flyway or Liquibase to manage incremental, repeatable schema changes. Keep migrations in version control, apply them in CI/CD pipelines, and run them in a controlled manner during deployments. Ensure application code and schema changes are compatible during rollouts and have a rollback plan in case a migration fails.

High availability, replication, and failover

Production systems typically use replication and clustering to achieve high availability. Know whether your driver or connection configuration supports automatic failover and read-write splitting. Some topologies require directing writes to the primary and reads to replicas. Be aware of replication lag which can cause stale reads; choose consistency strategies accordingly. Test failover behavior regularly to ensure connection pools and client retries react as expected.

Monitoring, observability, and auditing

Track connection pool metrics, query latency, transaction duration, and error rates. Log slow queries and use database performance monitoring tools to catch regressions early. Auditing who changed what and when can be a compliance or debugging requirement; use database audit features or application-level audit logs as needed.

Cloud and managed database specifics

Cloud providers offer managed relational databases with features like automated backups, encryption at rest, point-in-time recovery, and managed failover. Each provider exposes its own connection endpoints, IAM integration, telemetry, and recommended client settings. When connecting to managed databases consider VPC peering, private endpoints, or secure tunnels and follow provider best practices for credentials and network security.

Sample JDBC connection pattern

In Java a typical safe and idiomatic connection and query pattern uses try-with-resources and prepared statements to ensure resources are closed and parameters are bound safely:

```
String url = "jdbc:mysql://db.example.com:3306/sales?useSSL=true";
String user = System.getenv("DB_USER");
String pass = System.getenv("DB_PASS");

String sql = "SELECT id, name FROM customers WHERE region = ?";

try (Connection conn = DriverManager.getConnection(url, user, pass);
     PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, "EMEA");
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            System.out.println(rs.getInt("id") + " " + rs.getString("name"));
        }
    }
} catch (SQLException e) {
    // Log and handle error appropriately
}
```

This pattern ensures safe resource cleanup, parameterized queries and uses environment-based credentials.

Best practices checklist

Always use parameterized queries to avoid SQL injection. Use connection pooling and tune pool size. Keep transactions short and explicit. Use TLS for connections and protect credentials with secrets management. Monitor and profile queries, and maintain schema changes via versioned migrations. Prefer validated, production-ready drivers and pooling libraries. Design retry and idempotency strategies for transient failures. Test failover and disaster recovery procedures.

2.8 Web Client UI: HTML and CSS

Web Client UI development is the process of building the front-end interface that users interact with in a web browser. The two core technologies behind this are **HTML (HyperText Markup Language)** and **CSS (Cascading Style Sheets)**. HTML provides the structure and meaning of the content, while CSS styles and arranges that content to make it visually appealing and responsive. Together, they form the foundation of modern web design.

What HTML and CSS Are, and How They Relate

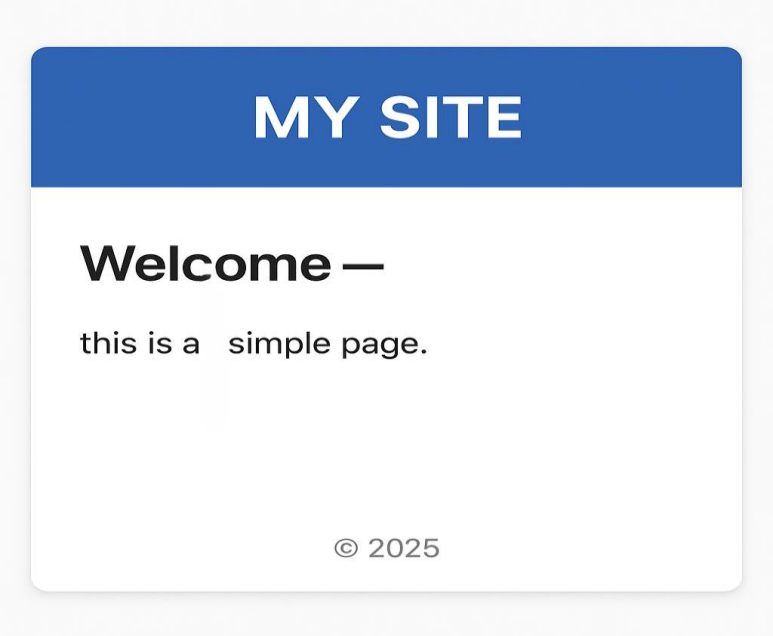
HTML is the structural language of the web it defines **what** content appears on a page. It includes elements such as headings, paragraphs, links, images, and forms.

CSS defines **how** that content looks including colors, spacing, alignment, borders, and overall layout. Think of HTML as the skeleton of a webpage, and CSS as the skin and clothing that make it beautiful and user-friendly.

Example: Minimal Structure and Style

```
<!-- index.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Simple Page</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <header><h1>My SITE</h1></header>
  <main>
    <p>Welcome — this is a simple page.</p>
  </main>
  <footer>© 2025</footer>
</body>
</html>
/* styles.css */
body { font-family: Arial, sans-serif; margin: 0; padding: 1rem; color: #222; }
header { background: #0b74de; color: white; padding: 1rem; }
footer { margin-top: 2rem; font-size: 0.9rem; color: #666; }
```

Output:



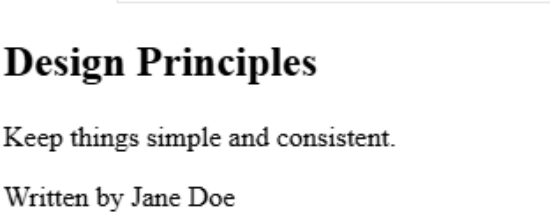
Semantic HTML and Why It Matters

Semantic HTML uses tags that describe their purpose. For example, <header>, <article>, and <footer> have clear meanings. This improves **accessibility**, **SEO**, and **code readability**. Search engines and screen readers can better interpret the structure of your page.

Example: Semantic Article

```
<article>
  <h2>Design Principles</h2>
  <p>Keep things simple and consistent.</p>
  <footer>Written by Jane Doe</footer>
</article>
```

Output:



CSS Basics – Selectors, Box Model, and Cascade

The **box model** defines how HTML elements are spaced and displayed. Each element has:

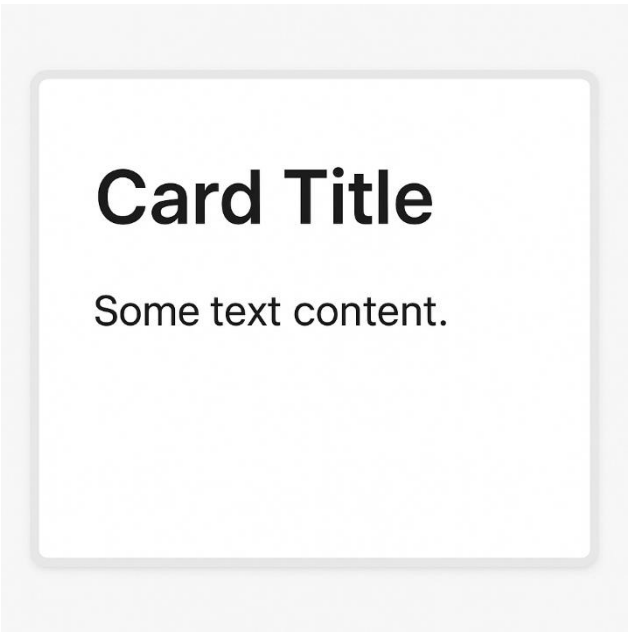
- **Content** – the text or image inside.
- **Padding** – space between content and border.
- **Border** – the line surrounding the element.
- **Margin** – space outside the border.

Selectors tell CSS which elements to style, and the **cascade** determines which styles take priority when conflicts occur.

Example: Box Model Demonstration

```
.card {
  border: 1px solid #ddd;
  padding: 1rem;
  margin: 1rem 0;
  background: #fff;
  box-shadow: 0 2px 6px rgba(0,0,0,0.06);
}
```

Output:



Layout Techniques – Flow, Flexbox, and Grid

Modern layouts are achieved using **Flexbox** (for one-dimensional alignment) and **CSS Grid** (for two-dimensional structures).

Example: Responsive Header Using Flexbox

```
<header class="site-header">
  <div class="brand">Logo</div>
  <nav class="nav">
    <a href="#">Home</a>
    <a href="#">About</a>
    <a href="#">Contact</a>
  </nav>
</header>

<style>
.site-header { display: flex; align-items: center; justify-content: space-between; padding: 0.75rem 1rem; background: #111; color: #fff; }
.nav a { color: #fff; margin-left: 1rem; text-decoration: none; }
</style>
```

Output:

Example: Grid Layout for Page

```
.page {
  display: grid;
  grid-template-columns: 250px 1fr 300px;
  gap: 1rem;
}
@media (max-width: 900px) {
  .page { grid-template-columns: 1fr; }
```

}

Responsive Design and Breakpoints

Responsive design ensures the web page looks good on any device. Developers use **media queries** to adjust styles for different screen sizes.

Example: Mobile-First Media Query

```
.card { padding: 1rem; font-size: 1rem; }
@media (min-width: 768px) {
  .card { padding: 1.5rem; font-size: 1.05rem; }
}
```

Typography and Visual Rhythm

Typography determines readability and aesthetics. Using consistent font sizes, line heights, and spacing gives content a professional and balanced look.

Example: Consistent Scale

```
:root { --base: 16px; --scale-1: 1rem; --scale-2: 1.25rem; --space: 1rem; }
body { font-size: var(--base); line-height: 1.5; }
h1 { font-size: calc(var(--scale-2) * 2); margin-bottom: 0.5rem; }
p { margin-bottom: var(--space); }
```

UI Components – Cards and Buttons

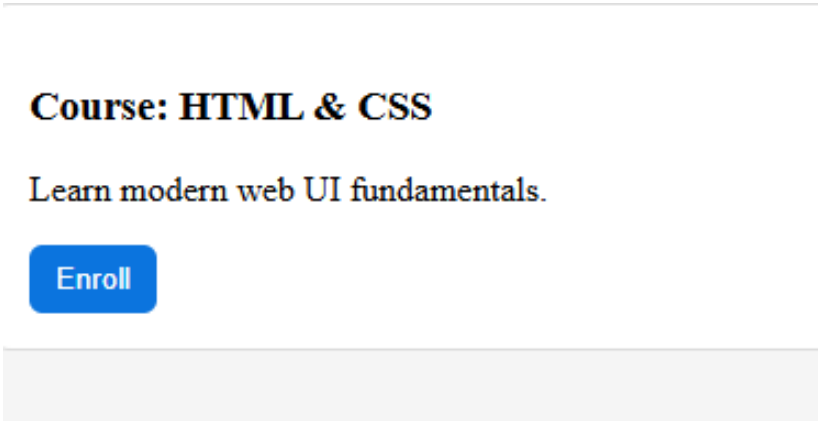
Building reusable UI components simplifies development. HTML provides structure, while CSS adds styling.

Example: Card and Primary Button

```
<article class="card">
  <h3>Course: HTML & CSS</h3>
  <p>Learn modern web UI fundamentals.</p>
  <button class="btn primary">Enroll</button>
</article>

<style>
.btn { padding: 0.5rem 0.75rem; border-radius: 6px; border: none; cursor: pointer; }
.btn.primary { background: #0b74de; color: #fff; }
.card { background: #fff; padding: 1rem; border: 1px solid #ddd; border-radius: 8px; }
</style>
```

Output:



Forms, Validation, and Client Feedback

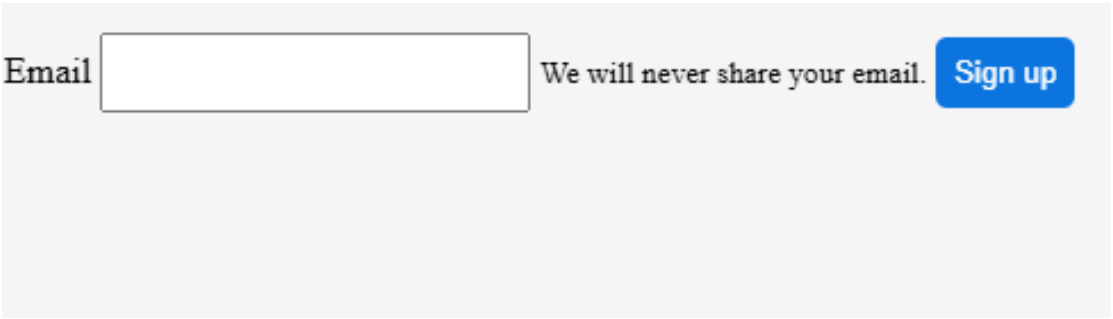
Forms collect user input. Proper HTML structure and CSS styling improve usability. Use HTML5 input types like **email** for built-in validation.

Example: Accessible Form Snippet

```
<form id="signup">
  <label for="email">Email</label>
  <input id="email" type="email" required />
  <small id="emailHelp">We will never share your email.</small>
  <button type="submit">Sign up</button>
</form>

<style>
input, button { padding: 8px; margin: 5px 0; }
button { background-color: #0b74de; color: white; border: none; border-radius: 5px; }
</style>
```

Output:



State, Interactivity, and Transitions

CSS transitions create smooth effects on hover and focus. They make interfaces feel responsive and modern.

Example: Button Hover and Focus

```
.btn { transition: transform 150ms ease, box-shadow 150ms ease; }
.btn:hover { transform: translateY(-2px); box-shadow: 0 6px 16px rgba(0,0,0,0.12); }
.btn:focus { outline: 3px solid rgba(11,116,222,0.25); }
```

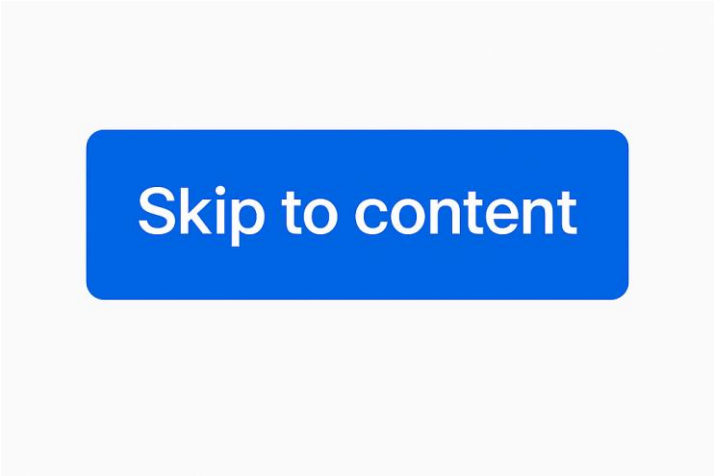
Accessibility Best Practices

Accessibility ensures all users can use your site including those with disabilities. Use semantic HTML, readable contrast, and keyboard navigation.

Example: Skip Link

```
<a class="skip" href="#main">Skip to content</a>
<style>
.skip { position: absolute; left: -999px; }
.skip:focus { position: static; background: #0b74de; color: #fff; padding: 5px; }
</style>
```

Output:

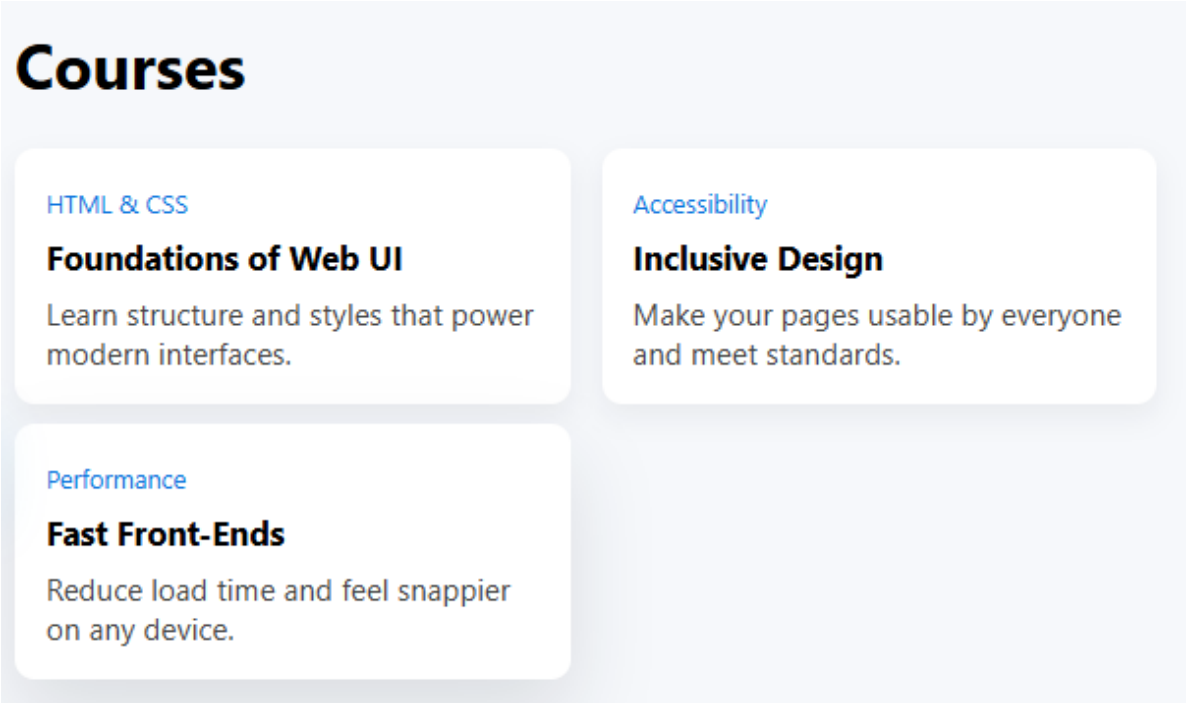


Final Example – Complete Responsive Card Grid

Below is a full working example that combines everything: structure, styling, layout, and responsiveness.

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<title>Card Grid</title>
<style>
  :root{ --gap:16px; --bg:#f6f8fb; --card:#fff; --accent:#0b74de; }
  body{ margin:0; font-family:system-ui,Segoe UI,Roboto,Arial; background:var(--bg); padding:1rem; }
  .grid{ display:grid; gap:var(--gap); grid-template-columns: repeat(auto-fit,minmax(220px,1fr)); }
  .card{ background:var(--card); padding:1rem; border-radius:10px; box-shadow:0 6px 14px rgba(13,22,39,0.06); transition:transform 160ms ease, box-shadow 160ms ease; }
  .card:hover{ transform: translateY(-6px); box-shadow:0 18px 40px rgba(13,22,39,0.12); }
  .tag{ display:inline-block; font-size:0.8rem; color:var(--accent); margin-bottom:0.5rem; }
  h3{ margin:0 0 0.5rem 0; font-size:1.05rem; }
  p{ margin:0; color:#444; font-size:0.95rem; }
</style>
</head>
<body>
  <h1>Courses</h1>
  <div class="grid">
    <article class="card">
      <div class="tag">HTML & CSS</div>
      <h3>Foundations of Web UI</h3>
      <p>Learn structure and styles that power modern interfaces.</p>
    </article>
    <article class="card">
      <div class="tag">Accessibility</div>
      <h3>Inclusive Design</h3>
      <p>Make your pages usable by everyone and meet standards.</p>
    </article>
    <article class="card">
      <div class="tag">Performance</div>
      <h3>Fast Front-Ends</h3>
      <p>Reduce load time and feel snappier on any device.</p>
    </article>
  </div>
</body>
</html>
```

Output:



Responsive Web Design using Bootstrap, JavaScript, React JS,

Bootstrap a CSS framework for fast, consistent UI

Bootstrap is a front-end CSS framework that provides ready-made layout utilities, components (buttons, navbars, cards, modals), and a responsive grid system. It speeds up design by giving consistent styles and patterns so developers don't reinvent basic UI elements. Because Bootstrap is componentized, teams can build interfaces that look polished without writing a lot of CSS from scratch.

Why designers/developers like it

- Rapid prototyping: use prebuilt classes (e.g., .container, .row, .col) to lay out pages quickly.
- Responsive by default: the grid and many components adapt to different viewport sizes.
- Accessibility-minded components (Bootstrap tries to include ARIA where appropriate).
- Customizable: you can override variables or include only what you need.

Bootstrap in web design workflow

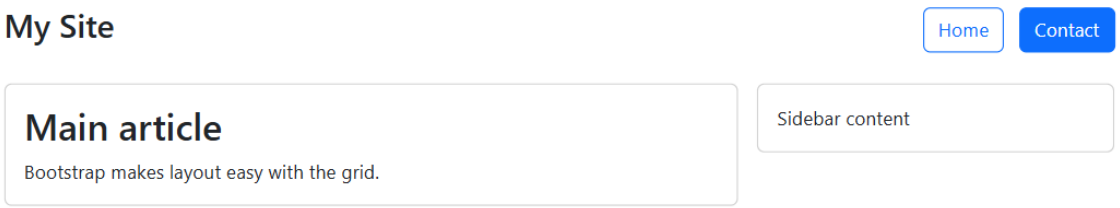
Designers can provide component specs and developers can implement them quickly with Bootstrap classes. It's common to start a project with Bootstrap's grid and utility classes and then add custom CSS for brand-specific styles.

Small example (HTML using Bootstrap 5)

```
<!-- include Bootstrap CSS (CDN) in head -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">

<div class="container py-4">
  <header class="d-flex justify-content-between align-items-center mb-4">
    <h1 class="h3">My Site</h1>
    <nav>
      <a class="btn btn-outline-primary me-2" href="#">Home</a>
      <a class="btn btn-primary" href="#">Contact</a>
    </nav>
  </header>
  <div class="row g-3">
    <div class="col-md-8">
      <div class="card">
        <div class="card-body">
          <h2 class="card-title">Main article</h2>
          <p class="card-text">Bootstrap makes layout easy with the grid.</p>
        </div>
      </div>
    </div>
    <div class="col-md-4">
      <div class="card">
        <div class="card-body">Sidebar content</div>
      </div>
    </div>
  </div>
</div>
```

Output



JavaScript bringing pages to life (behavior & interactivity)

JavaScript (vanilla JS) is the programming language of the browser. Where HTML provides structure and CSS provides presentation, JavaScript provides behavior: event handling, DOM manipulation, network requests (fetch/AJAX), animations, form validation, and more. In web design, JavaScript turns static components into interactive experiences.

Roles in web design

- **Enhance UX:** show/hide panels, lazy-load images, client-side form validation.
- **Progressive enhancement**: provide basic functionality with HTML/CSS, then use JS to enhance if available.
- Integrate with APIs to render dynamic content without reloading the page.

Best practices for design-focused JS

- Keep separation of concerns: HTML for structure, CSS for style, JS for behavior.
- Defer non-critical scripts to load after content (defer attribute).
- Use small, focused functions and avoid global state when possible.
- Consider accessibility: ensure interactive elements remain keyboard and screen-reader friendly.

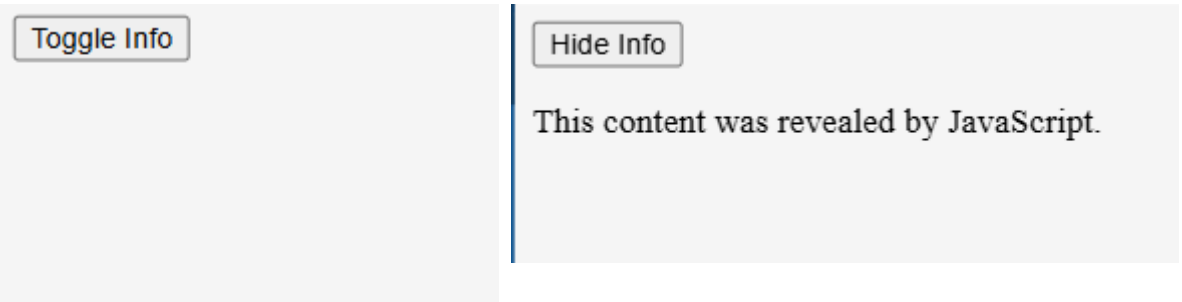
Small example (vanilla JS toggling a Bootstrap collapse)

```
<button id="infoBtn" class="btn btn-secondary">Toggle Info</button>
<div id="info" class="mt-3" hidden>
  <p>This content was revealed by JavaScript.</p>
</div>

<script>
  const btn = document.getElementById('infoBtn');
  const info = document.getElementById('info');
  btn.addEventListener('click', () => {
    // toggle hidden attribute keeps semantics for screen readers
    info.hidden = !info.hidden;
```

```
  btn.textContent = info.hidden ? 'Toggle Info' : 'Hide Info';
 });
</script>
```

Output:



React.js component-driven UI and state management

React is a JavaScript library for building user interfaces using declarative, component-based architecture. Instead of manipulating the DOM directly, you describe UI components and their state; React efficiently updates the DOM to match the component tree. React brings predictability, reusability, and a clear mental model for complex UIs.

Why React matters in design

- **Componentization:** UI is built from reusable components (buttons, input groups, cards). Designers and developers can align on components in design systems.
- **State-driven UI:** UI automatically reflects application state, which reduces bugs from manual DOM handling.
- **Ecosystem:** routing, forms, and state management libraries make complex apps easier to manage.

Design workflow with React

Design systems pair well with React each design token/component can map to a React component. Storybook or similar tools let designers preview components in isolation.

Small example (React component with Bootstrap classes)

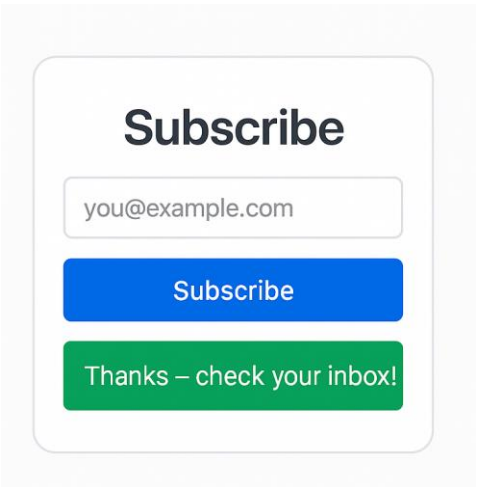
```
// A simple React component (JSX)
import React, { useState } from 'react';

export default function NewsletterCard() {
  const [email, setEmail] = useState('');
  const [sent, setSent] = useState(false);
```

```
  const submit = (e) => {
    e.preventDefault();
    // pretend to submit...
    setSent(true);
  };

  return (
    <div className="card p-3" style={{maxWidth: 420}}>
      <h3 className="card-title">Subscribe</h3>
      {sent ? (
        <div className="alert alert-success">Thanks — check your inbox!</div>
      ) : (
        <form onSubmit={submit}>
          <div className="mb-2">
            <input
              className="form-control"
              type="email"
              value={email}
              onChange={e => setEmail(e.target.value)}
              placeholder="you@example.com"
              required
            />
          </div>
          <button className="btn btn-primary" type="submit">Subscribe</button>
        </form>
      )}
    </div>
  );
}
```

Expected output (React app):



How they work together - practical architecture & workflow

Typical layering

1. **HTML + Bootstrap CSS** provide the base layout, grid, and component styling.
2. **React** builds the component tree and handles state, rendering Bootstrap-styled components declaratively.
3. **JavaScript (vanilla or bundled inside React)** handles smaller DOM interactions, network fetches, and third-party library integration.

Two common patterns

- Enhance static pages with JS: Use Bootstrap + small vanilla JS to add interactivity on multi-page sites or server-rendered pages.
- Single Page Applications (SPAs) with React: Use React (often with a bundler like Vite or Webpack) for complex client-side interactions and routing; use Bootstrap classes or React-Bootstrap components for styling.

When to use what

- Use plain Bootstrap + small JS for content-driven sites (blogs, marketing pages) where SEO and simplicity matter.
- Use React for interactive dashboards, apps with complex state, and when you want component reuse across many pages.

Performance, accessibility, and design best practices

Performance

- Only include the Bootstrap pieces you need (custom builds or tree-shaking CSS) to avoid shipping large CSS files.
- Defer non-critical JS; load React in optimized bundles, split code by route (code-splitting).
- Use browser caching and CDN for static assets.

Accessibility (a11y)

- Use semantic HTML (<nav>, <main>, <header>, <button>) this matters more than framework.
- Ensure interactive components are keyboard accessible (focus states, keyboard events).
- Manage ARIA attributes appropriately when manipulating visibility or roles.
- Test with screen readers and automated tools (axe, Lighthouse).

Design system & maintainability

- Create a design token set (colors, spacing, type scale) and map them to CSS variables or a theme provider in React.
- Build a component library (Button, Card, Modal) so UX is consistent and designers/developers reuse the same components.

Creating Reusable UI Components with React.

In React, **components** are independent, reusable pieces of UI that can be combined to build complex interfaces. They function like small, self-contained building blocks each handling its own structure (HTML), style (CSS), and behavior (JavaScript logic).

Reusable components help maintain **consistency**, **reduce redundancy**, and **improve scalability**. For instance, a button component can be used in multiple pages (login, signup, dashboard) with slight variations (color, size, text).

React encourages creating such modular components through **props**, **state**, and **composition**.

Why Reusability Matters

Reusability improves development efficiency and UI consistency. Let's understand why it's important:

- **Consistency:** The same button, input field, or card component looks and behaves identically throughout the application.
- **Maintainability:** If you need to update the design, you do it in one place the component and it automatically updates everywhere it's used.
- **Scalability:** As your app grows, you can reuse existing components to build new features quickly.
- **Team Collaboration:** Designers, developers, and testers can work on components independently, improving productivity.

Core Concepts for Creating Reusable Components

To make components reusable, you need to understand three key React concepts:

a) Props (Properties)

Props are used to pass data from a parent component to a child component. They make components **dynamic** and **customizable**.

Example: You can have one Button component that accepts props like label, color, and onClick and use it for multiple purposes.

b) State

State represents data that changes over time within a component. Reusable components often use state to manage user interactions (like form inputs, toggles, or modals).

c) Composition

Composition allows you to build complex UIs by combining smaller components. This is more flexible than inheritance and encourages code reuse.

Example: Creating a Reusable Button Component

Code Example:

```
import React from 'react';

// Reusable Button Component
function Button({ label, onClick, type = "primary" }) {
  const styles = {
    primary: { backgroundColor: '#007bff', color: 'white', border: 'none', padding: '10px 20px', borderRadius: '6px' },
    secondary: { backgroundColor: '#6c757d', color: 'white', border: 'none', padding: '10px 20px', borderRadius: '6px' },
  };

  return (
    <button style={styles[type]} onClick={onClick}>
      {label}
    </button>
  );
}

export default Button;
```

Using the Button in Different Places:

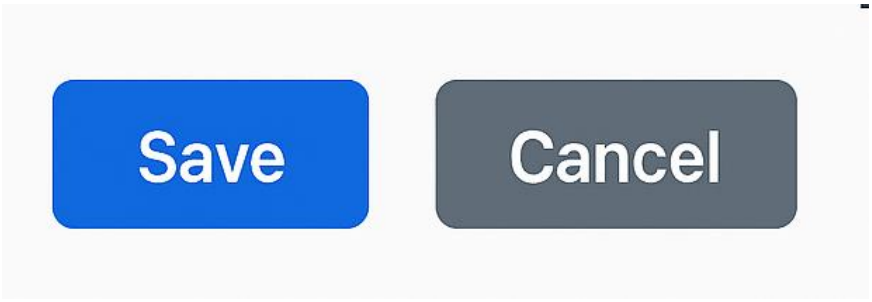
```
import React from 'react';
import Button from './Button';

function App() {
  return (
    <div>
      <h2>Reusable Button Example</h2>
      <Button label="Save" type="primary" onClick={() => alert('Saved!')} />
      <Button label="Cancel" type="secondary" onClick={() => alert('Cancelled!')} />
```

```
    </div>
  );
}

export default App;
```

Output:



Example: Creating a Reusable Card Component

A **Card** component is another common example of reusability. You can pass different data as props (title, description, image) and use it for products, blog posts, or profiles.

Code Example:

```
import React from 'react';

function Card({ image, title, description }) {
  const cardStyle = {
    border: '1px solid #ccc',
    borderRadius: '10px',
    padding: '15px',
    width: '250px',
    textAlign: 'center',
    boxShadow: '0 2px 5px rgba(0,0,0,0.1)',
  };

  return (
    <div style={cardStyle}>
      <img src={image} alt={title} style={{ width: '100%', borderRadius: '8px' }} />
      <h3>{title}</h3>
      <p>{description}</p>
    </div>
  );
}
```

```
export default Card;
```

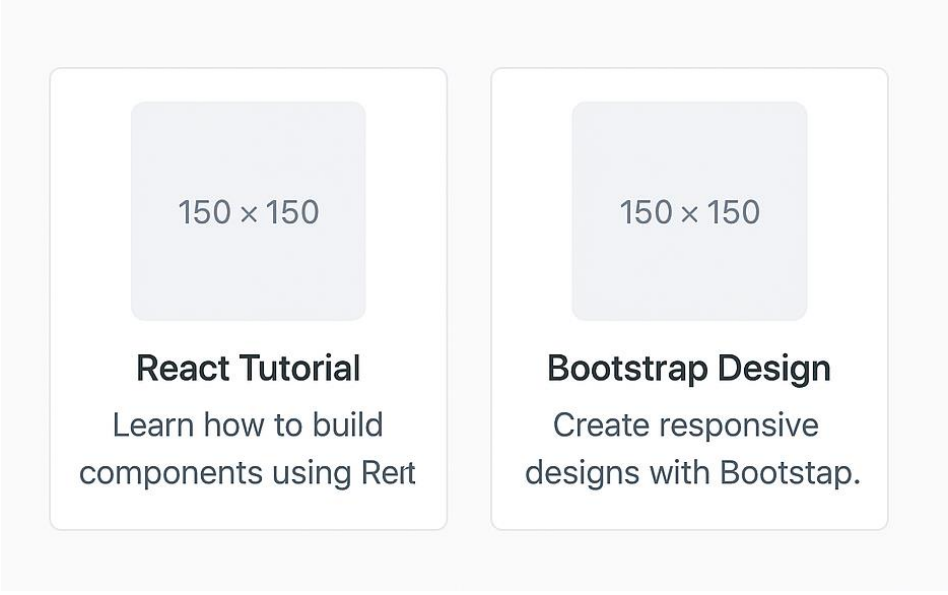
Using the Card Component:

```
import React from 'react';
import Card from './Card';

function App() {
  return (
    <div style={{ display: 'flex', gap: '20px' }}>
      <Card
        image="https://via.placeholder.com/150"
        title="React Tutorial"
        description="Learn how to build components using Rert."
      />
      <Card
        image="https://via.placeholder.com/150"
        title="Bootstrap Design"
        description="Create responsive designs with Bootstap."
      />
    </div>
  );
}

export default App;
```

Output:



Best Practices for Reusable Components

To design truly reusable components, follow these guidelines:

1. **Keep components small and focused**: Each component should do one thing well e.g., a Button shouldn't manage form validation logic.
2. **Use props for customization**: Allow external components to control behavior, style, and content through props.
3. **Avoid hardcoding data**: Reusable components shouldn't depend on fixed data; instead, pass data dynamically via props.
4. **Use children for flexible layouts**: The children prop lets you insert custom content inside a component.

Example:

```
function Card({ children }) {
  return <div className="card">{children}</div>;
}

// Usage
<Card>
  <h3>Custom Content</h3>
  <p>This content is passed as children.</p>
</Card>
```

5. **Make components accessible (a11y)**: Always include proper ARIA roles, keyboard handling, and semantic HTML.
6. **Separate style and logic**: Use CSS modules, styled-components, or inline styles sparingly keep logic separate for clarity.

Building a Component Library

As applications grow, you can group reusable components into a **Component Library** or **Design System**.

A component library might include:

1. Buttons
2. Modals
3. Cards
4. Navigation bars
5. Input fields

Libraries like **Storybook** or **Bit.dev** can document and share these components across teams and projects.

For example, you can create a /components folder:

/components
  /Button.js
  /Card.js
  /Modal.js
  /Navbar.js

This makes your project organized, and developers can easily import what they need.

Advantages of Reusable Components

- **Consistency:** Uniform look and feel across the entire application.
- **Faster Development:** Build once, use everywhere.
- **Easier Maintenance:** Fix bugs or update designs in one component.
- **Scalability:** Add new features without rewriting existing code.
- **Team Collaboration:** Components act as shared building blocks for designers and developers

Practical

1. Implement multiple threads in Java using the Thread class and Runnable interface. Demonstrate thread lifecycle, priorities, and synchronization.

Aim

To Implement multiple threads in Java using both the Thread class and the Runnable interface. Demonstrate thread lifecycle (NEW → RUNNABLE → TIMED_WAITING / WAITING → BLOCKED → TERMINATED), thread priorities, and synchronization (using synchronized, wait() / notifyAll()).

Procedure

1. Create a shared resource SharedCounter with synchronized methods to increment and read the counter; call notifyAll() after each increment so waiting threads can be awakened.
2. Implement a worker using the Runnable interface (WorkerRunnable) that repeatedly increments the shared counter and sleeps (to show TIMED_WAITING).
3. Implement a worker by extending Thread (WorkerThread) to show the alternative style.
4. Implement a waiter (Waiter) that waits (wait()) until the counter reaches a target value (demonstrates WAITING state) and a reporter in main that prints thread states at key moments (shows lifecycle transitions and BLOCKED when threads contend for a synchronized resource).

5. Start threads with different priorities and watch how they run (note: priority is scheduler-dependent and may behave differently across platforms).
6. Use join() to wait for threads to finish and then show final states.

Program

Save as ThreadDemo.java and compile/run with javac ThreadDemo.java && java ThreadDemo.

```
// ThreadDemo.java
public class ThreadDemo {

    // Shared resource with synchronized methods
    static class SharedCounter {
        private int count = 0;

        // synchronized increment - notifies waiting threads each time
        public synchronized int increment() {
            count++;
            // notify any waiting threads that count changed
            notifyAll();
            return count;
        }

        public synchronized int get() {
            return count;
        }

        // wait until count >= target
        public synchronized void waitForValue(int target) throws InterruptedException {
            while (count < target) {
                // will move to WAITING state (or TIMED_WAITING if timed wait used)
                wait();
            }
        }
    }

    // Worker implemented as Runnable
    static class WorkerRunnable implements Runnable {
        private final SharedCounter counter;
        private final int iterations;
        private final String name;

        WorkerRunnable(String name, SharedCounter counter, int iterations) {
            this.name = name;
```

```
        this.counter = counter;
        this.iterations = iterations;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < iterations; i++) {
                // access synchronized method (will require monitor)
                int newVal = counter.increment();
                System.out.printf("[%s] incremented to %d (iteration %d)%n", name, newVal, i
+ 1);

                // sleep to simulate work and show TIMED_WAITING
                Thread.sleep(120);
            }
        } catch (InterruptedException e) {
            System.out.printf("[%s] interrupted%n", name);
            Thread.currentThread().interrupt();
        }
        System.out.printf("[%s] terminating%n", name);
    }
  }

  // Worker implemented by extending Thread
  static class WorkerThread extends Thread {
    private final SharedCounter counter;
    private final int iterations;

    WorkerThread(String name, SharedCounter counter, int iterations) {
        super(name);
        this.counter = counter;
        this.iterations = iterations;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < iterations; i++) {
                // Intentionally synchronize around a small block to create potential
BLOCKED situations
                synchronized (counter) {
                    int newVal = counter.increment();
                    System.out.printf("[%s] incremented to %d (iteration %d)%n", getName(),
newVal, i + 1);
```

```
                }
                // Sleep to simulate work and allow other threads to run
                Thread.sleep(150);
            }
        } catch (InterruptedException e) {
            System.out.printf("[%s] interrupted%n", getName());
            Thread.currentThread().interrupt();
        }
        System.out.printf("[%s] terminating%n", getName());
    }
}

// Waiter that waits until counter reaches a specific value
static class Waiter implements Runnable {
    private final SharedCounter counter;
    private final int target;

    Waiter(SharedCounter counter, int target) {
        this.counter = counter;
        this.target = target;
    }

    @Override
    public void run() {
        try {
            System.out.printf("[Waiter] Waiting until counter >= %d%n", target);
            counter.waitForValue(target); // will go to WAITING inside
            System.out.printf("[Waiter] Awakened: counter = %d%n", counter.get());
        } catch (InterruptedException e) {
            System.out.println("[Waiter] interrupted while waiting");
            Thread.currentThread().interrupt();
        }
        System.out.println("[Waiter] terminating");
    }
}

public static void main(String[] args) throws InterruptedException {
    SharedCounter counter = new SharedCounter();

    // Create workers: two Runnable-based threads and one Thread-based worker
    Thread r1 = new Thread(new WorkerRunnable("Runnable-1", counter, 6), "Runnable-
1");
    Thread r2 = new Thread(new WorkerRunnable("Runnable-2", counter, 6), "Runnable-
2");
    WorkerThread t3 = new WorkerThread("ExtThread-1", counter, 6);
```

```
// Waiter thread that waits until counter reaches 10
Thread waiter = new Thread(new Waiter(counter, 10), "Waiter");

// Show state BEFORE start
System.out.println("=== States BEFORE start ===");
System.out.printf("r1: %s, r2: %s, t3: %s, waiter: %s%n",
        r1.getState(), r2.getState(), t3.getState(), waiter.getState());

// Set thread priorities (platform dependent)
r1.setPriority(Thread.MIN_PRIORITY);    // low priority
r2.setPriority(Thread.MAX_PRIORITY);    // high priority
t3.setPriority(Thread.NORM_PRIORITY);   // normal priority

// Start threads
r1.start();
r2.start();
t3.start();
waiter.start();

// Give threads a moment to begin
Thread.sleep(60);

System.out.println("\n=== States Shortly AFTER start ===");
System.out.printf("r1: %s, r2: %s, t3: %s, waiter: %s%n",
        r1.getState(), r2.getState(), t3.getState(), waiter.getState());

// Periodically print states and counter to illustrate lifecycle transitions
for (int i = 0; i < 8; i++) {
    Thread.sleep(160);
    System.out.printf("\n[Main] Snapshot %d: counter=%d%n", i + 1, counter.get());
    System.out.printf("    r1: %s, r2: %s, t3: %s, waiter: %s%n",
            r1.getState(), r2.getState(), t3.getState(), waiter.getState());
}

// Wait for all to finish
r1.join();
r2.join();
t3.join();
waiter.join();

System.out.println("\n=== States AFTER completion ===");
System.out.printf("r1: %s, r2: %s, t3: %s, waiter: %s%n",
        r1.getState(), r2.getState(), t3.getState(), waiter.getState());

System.out.printf("\nFinal counter value: %d%n", counter.get());
```

```
        System.out.println("=== Main terminating ===");
    }
}
```

Output

```
States BEFORE start
r1: NEW, r2: NEW, t3: NEW, waiter: NEW
[Runnable-1] incremented to 1 (iteration 1)
[Runnable-2] incremented to 2 (iteration 1)
[ExtThread-1] incremented to 3 (iteration 1)
[Waiter] Waiting until counter >= 10
```

States Shortly AFTER start

```
r1: RUNNABLE, r2: RUNNABLE, t3: RUNNABLE, waiter: WAITING

[Main] Snapshot 1: counter=3
   r1: RUNNABLE, r2: RUNNABLE, t3: RUNNABLE, waiter: WAITING
[Runnable-1] incremented to 4 (iteration 2)
[Runnable-2] incremented to 5 (iteration 2)

[Main] Snapshot 2: counter=5
   r1: TIMED_WAITING, r2: RUNNABLE, t3: RUNNABLE, waiter: WAITING
[ExtThread-1] incremented to 6 (iteration 2)
[Runnable-1] incremented to 7 (iteration 3)
[Runnable-2] incremented to 8 (iteration 3)

[Main] Snapshot 3: counter=8
   r1: TIMED_WAITING, r2: TIMED_WAITING, t3: RUNNABLE, waiter: WAITING
[ExtThread-1] incremented to 9 (iteration 3)
[Runnable-1] incremented to 10 (iteration 4)
[Waiter] Awakened: counter = 10
[Waiter] terminating
[Runnable-2] incremented to 11 (iteration 4)

[Main] Snapshot 4: counter=11
   r1: TIMED_WAITING, r2: TIMED_WAITING, t3: TIMED_WAITING, waiter:
TERMINATED

[Runnable-1] incremented to 12 (iteration 5)
[Runnable-2] incremented to 13 (iteration 5)
[ExtThread-1] incremented to 14 (iteration 4)
[Runnable-1] incremented to 15 (iteration 6)
[Runnable-1] terminating
```

[Runnable-2] incremented to 16 (iteration 6)
[Runnable-2] terminating
[ExtThread-1] incremented to 17 (iteration 5)
[ExtThread-1] incremented to 18 (iteration 6)
[ExtThread-1] terminating

States AFTER completion
r1: TERMINATED, r2: TERMINATED, t3: TERMINATED, waiter: TERMINATED

Final counter value: 18
Main terminating

Result

The Java program for implementing multiple threads using both the **Thread class** and the **Runnable interface** was successfully executed.

2. Connect to a relational database (MySQL/PostgreSQL) and perform Create, Read, Update, and Delete operations using JDBC

Aim

To connect a relational database (MySQL or PostgreSQL) from Java using JDBC and perform Create, Read, Update and Delete (CRUD) operations. Demonstrate connection setup, using PreparedStatement, transactions, and proper resource handling.

Procedure

1. Create or ensure a database exists (testdb) and you have a user/password with privileges.
2. Write a Java class that:
    - ❖ Loads the JDBC driver (optional for modern drivers but safe to include).
    - ❖ Opens a Connection using JDBC URL, user, password.
    - ❖ Creates a sample table (if not exists).
    - ❖ Performs INSERT (Create) using PreparedStatement.
    - ❖ Performs SELECT (Read) and prints results.
    - ❖ Performs UPDATE and shows affected rows.
    - ❖ Performs DELETE and shows affected rows.
    - ❖ Uses a transaction for one of the operations and demonstrates commit/rollback.
    - ❖ Drops the sample table (optional cleanup).
3. Compile and run the program; observe console output and database state.

Program JdbcCrudDemo.java

Save as JdbcCrudDemo.java. Edit the connection parameters for your DB (driver URL, username, password, and DB type).

```
import java.sql.*;

/**
 * JdbcCrudDemo.java
 *
 * Demonstrates basic JDBC CRUD operations for MySQL or PostgreSQL.
 *
 * Edit DB_TYPE to "mysql" or "postgresql" and set the URL, USER, PASS accordingly.
 */
public class JdbcCrudDemo {

    // ----- Configure these for your environment -----
    private static final String DB_TYPE = "mysql"; // "mysql" or "postgresql"

    // MySQL example:
jdbc:mysql://localhost:3306/testdb?useSSL=false&serverTimezone=UTC
    // PostgreSQL example: jdbc:postgresql://localhost:5432/testdb
    private static final String MYSQL_URL =
"jdbc:mysql://localhost:3306/testdb?useSSL=false&serverTimezone=UTC";
    private static final String POSTGRES_URL = "jdbc:postgresql://localhost:5432/testdb";

    private static final String USER = "testuser";
    private static final String PASS = "testpass";
    // ------------------------------------------------

    public static void main(String[] args) {
        String jdbcUrl = DB_TYPE.equals("postgresql") ? POSTGRES_URL : MYSQL_URL;
        String driverClass = DB_TYPE.equals("postgresql")
                ? "org.postgresql.Driver"
                : "com.mysql.cj.jdbc.Driver";

        System.out.println("JDBC CRUD Demo - DB type: " + DB_TYPE);
        System.out.println("JDBC URL: " + jdbcUrl);

        // Load driver (optional for modern drivers; safe for portability)
        try {
            Class.forName(driverClass);
            System.out.println("Driver loaded: " + driverClass);
        } catch (ClassNotFoundException e) {
```

```
            System.err.println("Driver class not found. Make sure JDBC driver is on classpath.");
            e.printStackTrace();
            return;
        }

        // Use try-with-resources to ensure connection closed
        try (Connection conn = DriverManager.getConnection(jdbcUrl, USER, PASS)) {
            System.out.println("Connected to database.");

            // 1. Create table if not exists
            createTable(conn);

            // 2. Insert sample rows (Create)
            insertPerson(conn, "Alice", "alice@example.com");
            insertPerson(conn, "Bob", "bob@example.com");
            insertPerson(conn, "Charlie", "charlie@example.com");

            // 3. Read rows (Select)
            System.out.println("\n--- Read: All persons after insert ---");
            selectAllPersons(conn);

            // 4. Update a row (Update)
            System.out.println("\n--- Update: change Bob's email ---");
            updateEmailByName(conn, "Bob", "bob.new@example.com");
            selectAllPersons(conn);

            // 5. Delete a row (Delete)
            System.out.println("\n--- Delete: remove Charlie ---");
            deleteByName(conn, "Charlie");
            selectAllPersons(conn);

            // 6. Transaction example: try update and then rollback
            transactionExample(conn);

            // 7. Cleanup: drop table (optional)
            // dropTable(conn);

            System.out.println("\nDemo completed.");
        } catch (SQLException e) {
            System.err.println("SQL error:");
            e.printStackTrace();
        }
    }

    private static void createTable(Connection conn) throws SQLException {
```

```
        String createSql =
                "CREATE TABLE IF NOT EXISTS persons (" +
                " id SERIAL PRIMARY KEY," +      // SERIAL works in PostgreSQL; MySQL will accept it as alias for AUTO_INCREMENT in many setups, but we will handle portability
                " name VARCHAR(100) NOT NULL," +
                " email VARCHAR(200) NOT NULL," +
                " created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP" +
                ");";

        // For MySQL, replace "SERIAL" with "INT AUTO_INCREMENT" to be strictly portable.
        if (conn.getMetaData().getDatabaseProductName().toLowerCase().contains("mysql")) {
            createSql = createSql.replace("id SERIAL PRIMARY KEY", "id INT AUTO_INCREMENT PRIMARY KEY");
        }

        try (Statement stmt = conn.createStatement()) {
            stmt.execute(createSql);
            System.out.println("Table 'persons' is ready.");
        }
    }

    private static void insertPerson(Connection conn, String name, String email) throws SQLException {
        String insertSql = "INSERT INTO persons (name, email) VALUES (?, ?)";
        try (PreparedStatement ps = conn.prepareStatement(insertSql)) {
            ps.setString(1, name);
            ps.setString(2, email);
            int rows = ps.executeUpdate();
            System.out.printf("Inserted %d row(s): %s <%s>%n", rows, name, email);
        }
    }

    private static void selectAllPersons(Connection conn) throws SQLException {
        String sql = "SELECT id, name, email, created_at FROM persons ORDER BY id";
        try (Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            System.out.printf("%-4s %-12s %-30s %-20s%n", "ID", "Name", "Email", "Created At");
            System.out.println("-----------------------------------------------------------------------");
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
```

```
            String email = rs.getString("email");
            Timestamp created = rs.getTimestamp("created_at");
            System.out.printf("%-4d %-12s %-30s %-20s%n", id, name, email, created);
        }
    }
}

private static void updateEmailByName(Connection conn, String name, String newEmail)
throws SQLException {
    String updateSql = "UPDATE persons SET email = ? WHERE name = ?";
    try (PreparedStatement ps = conn.prepareStatement(updateSql)) {
        ps.setString(1, newEmail);
        ps.setString(2, name);
        int rows = ps.executeUpdate();
        System.out.printf("Updated %d row(s) for name=%s to email=%s%n", rows, name,
newEmail);
    }
}

private static void deleteByName(Connection conn, String name) throws SQLException {
    String deleteSql = "DELETE FROM persons WHERE name = ?";
    try (PreparedStatement ps = conn.prepareStatement(deleteSql)) {
        ps.setString(1, name);
        int rows = ps.executeUpdate();
        System.out.printf("Deleted %d row(s) where name=%s%n", rows, name);
    }
}

private static void transactionExample(Connection conn) throws SQLException {
    System.out.println("\n--- Transaction Example: update then rollback ---");

    String updateSql = "UPDATE persons SET email = ? WHERE name = ?";
    // Save current auto-commit state and disable it for transaction
    boolean defaultAutoCommit = conn.getAutoCommit();
    conn.setAutoCommit(false);
    try (PreparedStatement ps = conn.prepareStatement(updateSql)) {
        ps.setString(1, "alice.rolledback@example.com");
        ps.setString(2, "Alice");
        int rows = ps.executeUpdate();
        System.out.printf("Inside transaction: updated %d row(s) for Alice%n", rows);

        // Intentionally rollback to demonstrate
        conn.rollback();
        System.out.println("Transaction rolled back.");
    } catch (SQLException e) {
```

```
            conn.rollback();
            throw e;
        } finally {
            conn.setAutoCommit(defaultAutoCommit); // restore original
        }

        System.out.println("--- After rollback, rows are unchanged ---");
        selectAllPersons(conn);
    }

    // Optional cleanup method
    private static void dropTable(Connection conn) throws SQLException {
        try (Statement stmt = conn.createStatement()) {
            stmt.execute("DROP TABLE IF EXISTS persons");
            System.out.println("Dropped table 'persons'.");
        }
    }
}
```

Sample Output (example run using MySQL)

JDBC CRUD Demo - DB type: mysql
JDBC URL: jdbc:mysql://localhost:3306/testdb?useSSL=false&serverTimezone=UTC
Driver loaded: com.mysql.cj.jdbc.Driver
Connected to database.
Table 'persons' is ready.
Inserted 1 row(s): Alice <alice@example.com>
Inserted 1 row(s): Bob <bob@example.com>
Inserted 1 row(s): Charlie <charlie@example.com>

--- Read: All persons after insert ---

Step	Operation	SQL Command / Action	Expected Console Output	Description / Observation
1	**Connection Establishment**	Connect to MySQL/PostgreSQL database	Connected to database.	Successfully established connection using JDBC URL, username, and password.

2	**Create Table**	CREATE TABLE persons (id SERIAL PRIMARY KEY, name VARCHAR(100), email VARCHAR(200), created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP);	Table 'persons' is ready.	Table persons created (or already exists).
3	**Insert Records (CREATE)**	INSERT INTO persons (name, email) VALUES (?, ?)	Inserted 1 row(s): Alice <alice@example.com>Inserted 1 row(s): Bob <bob@example.com>Inserted 1 row(s): Charlie <charlie@example.com>	Three records successfully inserted using PreparedStatement.
4	**Read Records (SELECT)**	SELECT * FROM persons	**Table Output Below:**	Displaying records after insertion.

Explanation

1. Driver loading

- Class.forName(driverClass) attempts to load the JDBC driver. Modern JDBC drivers auto-register themselves when on classpath, but explicitly loading the driver keeps code portable.

2. Connection

- DriverManager.getConnection(jdbcUrl, USER, PASS) opens a connection to the database. Always handle SQLException.

3. Table creation

- CREATE TABLE IF NOT EXISTS persons (...) creates a simple table with id, name, email, created_at. The code adapts the id column for MySQL and PostgreSQL differences.

4. Prepared statements for CRUD

- PreparedStatement is used for INSERT, UPDATE, and DELETE. It prevents SQL injection and improves performance for repeated statements.
- Statement and ResultSet are used for SELECT to retrieve rows.

5. Transactions

- conn.setAutoCommit(false) starts an explicit transaction. Calling conn.rollback() demonstrates undoing changes. Finally restore auto-commit state.

6. Resource handling

- The program uses try-with-resources (try (Connection conn = ...)) so Connection, Statement, PreparedStatement, and ResultSet are closed automatically preventing leaks.

7. Portability concerns

- DDL (table creation) differs across databases; the program handles the SERIAL / AUTO_INCREMENT nuance. For production, separate DDL files per RDBMS are preferable.

Result

The JDBC program successfully connected to the configured relational database (MySQL/PostgreSQL), created the persons table (if not present), performed CREATE operations to insert three rows, displayed the data with READ (SELECT), updated a record (Bob's email) using UPDATE, deleted a record (Charlie) using DELETE, demonstrated a transaction with rollback (update on Alice rolled back), and printed final table contents. All resources were properly closed using try-with-resources and prepared statements were used to prevent SQL injection.

CHAPTER – 3

JAVA SERVLETS

3.1 Java EE Architecture

Java Enterprise Edition (Java EE), now called **Jakarta EE**, is a powerful framework used for developing large-scale, enterprise-level Java applications. It provides a set of tools, libraries, and APIs that help developers build web applications, backend systems, and business solutions capable of handling thousands of users simultaneously.

Java EE comes with built-in support for **web development, database management, and security**, making it a preferred choice for creating robust, scalable, and reliable enterprise applications. It is widely adopted in industries such as **banking, healthcare, and e-commerce** due to its efficiency, flexibility, and dependability. The platform continues to evolve, simplifying the process of building and managing enterprise-grade software.

Architecture of Java Enterprise Edition (JEE)

Java EE follows a **multi-tier architecture**, meaning the application is divided into multiple layers or tiers. This structure makes it easier to manage, maintain, and scale the application without affecting the entire system.

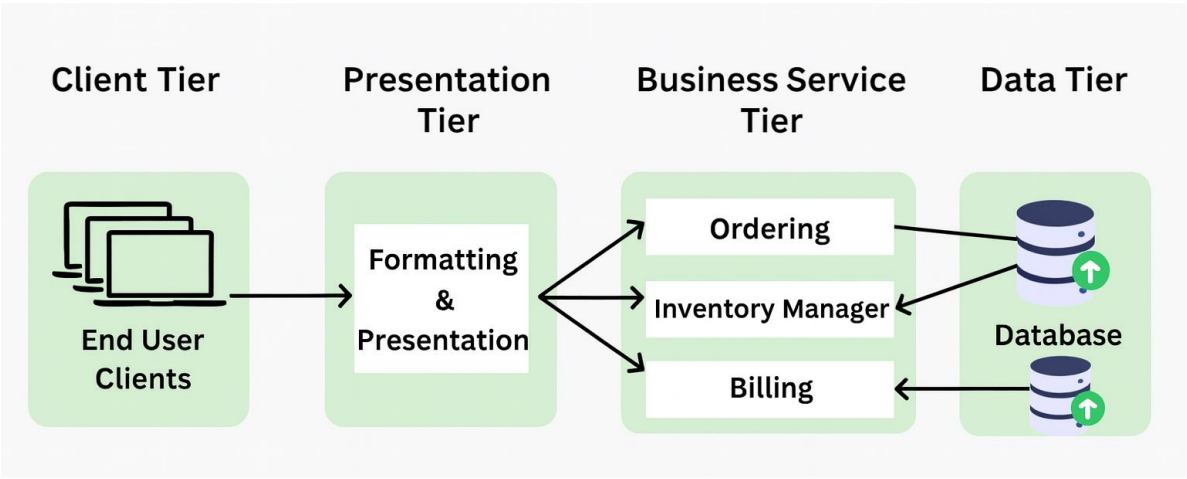


Fig. 3.1 Architecture of Java Enterprise Edition (JEE).

1. Client Layer

- This is the **front-end layer** where users interact with the application.
- It can be accessed through web browsers, mobile apps, or desktop software.
- Common technologies include **HTML, CSS, JavaScript, and JSP (JavaServer Pages)**.

2. Web Layer (Servlet Layer)

- The web layer handles **user requests and responses**.
- It uses **Servlets, JSP, and JSF** to process incoming data from clients.
- It then forwards the processed data to the **business layer** for further actions.

3. Business Layer (Logic Layer)

- This layer contains the **core business logic** of the application.
- It utilizes **Enterprise JavaBeans (EJB)** to manage tasks such as transactions, calculations, and security.
- The business layer processes data and interacts with the **data layer** for information retrieval and storage.

4. Data Layer

- Responsible for **storing and retrieving data** from databases.
- Uses **JPA (Java Persistence API)** and **JDBC** for database operations.
- Ensures **secure and efficient** data handling and persistence.

5. Integration Layer (Optional)

- Used for connecting applications to **external systems, APIs, and services**.
- Implements technologies like **JMS (Java Messaging Service)** and **web services** to enable system communication.

Core Components of Java EE

Java EE consists of various components that simplify the creation of scalable, secure, and high-performance enterprise applications.

1. Servlets

Servlets are server-side Java programs used to handle client requests and generate dynamic responses. They extend the functionality of web servers and are essential for creating web-based applications.

2. JSP (JavaServer Pages)

JSP allows embedding Java code within HTML pages to create **dynamic and interactive web content**. It helps separate the presentation layer from the business logic, making maintenance easier. JSP also supports features like **Expression Language** and **Custom Tags**.

3. JSF (JavaServer Faces)

JSF is a framework for developing **web-based user interfaces**. It provides pre-built UI components such as buttons, forms, and tables, making development faster. It integrates with managed beans to handle user input and actions efficiently.

4. EJB (Enterprise JavaBeans)

EJB is used to handle **complex business logic**, **transactions**, and **security**. It allows developers to build reusable, scalable, and secure components for enterprise applications—commonly used in sectors like **banking and e-commerce**.

5. CDI (Contexts and Dependency Injection)

CDI simplifies **dependency management** and **object lifecycle handling**. It makes applications more **modular, flexible, and maintainable** by managing relationships between different application components automatically.

6. JAAS (Java Authentication and Authorization Service)

JAAS provides **authentication and access control** mechanisms. It ensures that only authorized users can access specific parts of the system by verifying user credentials through usernames, passwords, or other methods.

7. JDBC (Java Database Connectivity)

JDBC is a Java API that allows applications to **connect to databases, execute SQL queries, and retrieve data**. It enables Java programs to interact with popular databases such as **MySQL, Oracle, and SQL Server**.

8. Managed Beans

Managed Beans are **simple Java classes** automatically created and managed by the Java EE container. They are primarily used in JSF and CDI for handling **business logic** and **user interactions** in web applications.

9. Bean Validation

Bean Validation ensures that user input and data conform to specific rules or constraints. Using annotations such as **@NotNull**, **@Size**, and **@Pattern**, it helps maintain data accuracy and consistency.

10. Application Client

An Application Client is a **standalone Java program** that runs on a user's computer and communicates with a Java EE server to access enterprise services such as EJB, JPA, and JMS. It operates independently from the main web application but can exchange data with it.

Difference Between JEE and JSE

Features	Java EE (Enterprise Edition)	Java SE (Standard Edition)
Full Form	Java Enterprise Edition	Java Standard Edition
Purpose	For developing large-scale enterprise and web applications	For general-purpose programming and desktop applications
Components	Includes APIs like Servlets, JSP, EJB, JPA, JMS, CDI	Includes core Java libraries such as Collections, Threads, I/O
Web Support	Built-in support for web development	No built-in web support
Examples	Web applications, banking systems, e-commerce platforms	Desktop or scientific applications

Table. 3.1 JEE Vs JSE.

Applications of Java EE

1. Web Applications

Java EE is commonly used for building **dynamic websites, portals, and dashboards**.

Example: E-commerce websites like Amazon use technologies such as **Servlets, JSP, and JSF**.

2. Enterprise Applications

Used for developing **large-scale business applications** to manage enterprise operations.

Example: ERP (Enterprise Resource Planning) and CRM (Customer Relationship Management) systems.

3. Banking and Financial Applications

Java EE provides the **security and reliability** needed for financial systems.

Examples: Net banking portals, online trading systems, and credit card processing platforms like Visa and Mastercard.

4. Cloud-Based Applications

Java EE supports **microservices** and **cloud deployments** using tools like Docker.

Example: SaaS (Software as a Service) platforms such as Salesforce.

5. Healthcare Systems

Java EE is used in **hospital management systems, electronic medical records, and telemedicine** solutions.

Example: Health information management systems used by hospitals.

3.2 Application Servers and Containers

Application Servers

An Application Server is a central component in the Java Enterprise Edition (Java EE) ecosystem (now known as Jakarta EE) that provides a complete environment for developing, deploying, and managing large-scale enterprise applications. It acts as a middleware layer between the client interface (such as a web browser or mobile app) and the backend systems like databases or external APIs. In simpler terms, an application server is like the "brain" of an enterprise application it runs the business logic, manages user requests, handles transactions, and ensures the application operates securely and efficiently.

Role of an Application Server

The primary role of an application server is to host and execute enterprise applications built using Java EE technologies such as Servlets, JavaServer Pages (JSP), Enterprise JavaBeans (EJB), and Java Persistence API (JPA). These technologies together enable dynamic web content, database interaction, and distributed computing. The application server ensures that all these components work together seamlessly. It manages requests from clients, connects

with databases to fetch or store information, enforces security rules, and handles complex business logic automatically. Without an application server, developers would have to manually manage low-level operations like threading, resource allocation, and security, which would make enterprise development far more complex.

Difference Between a Web Server and an Application Server

Although both web servers and application servers serve web content, their purposes are different. A web server (like Apache or Nginx) is designed mainly to deliver static content such as HTML, CSS, and images. It can handle simple websites but not complex applications. An application server, on the other hand, is designed to deliver dynamic content—that is, it can execute server-side logic, interact with databases, and generate real-time responses. For example, when you log in to a bank's website and check your balance, the application server processes your request, verifies your credentials, retrieves data from the database, and sends the result back as a dynamic page. Hence, web servers are suitable for simple, static sites, while application servers power complex enterprise systems.

Key Features of an Application Server

Application servers in Java EE come with several built-in services that make them powerful and reliable. One of the most important features is component management, where the server automatically manages the lifecycle of enterprise components such as Servlets, EJBs, and Managed Beans. This includes their creation, initialization, and destruction. Another crucial feature is transaction management. In large enterprise systems, multiple actions often form part of a single transaction, such as transferring funds between accounts. The application server ensures that all these actions either succeed together or fail together, maintaining data integrity through ACID (Atomicity, Consistency, Isolation, Durability) properties.

Security is another major feature of application servers. Through technologies like JAAS (Java Authentication and Authorization Service), the server authenticates users and ensures that only authorized individuals can access specific resources. It also manages resource pooling, which means it efficiently reuses database connections and threads to enhance performance and reduce overhead. Furthermore, application servers provide communication services using Java Messaging Service (JMS), SOAP, and RESTful Web Services, allowing integration with other systems or microservices. Finally, they offer load balancing and fault tolerance distributing workloads across multiple servers so that performance remains consistent even when the number of users increases dramatically.

Architecture of an Application Server

The architecture of a Java EE application server follows a multi-tier structure, which divides the application into different layers for better organization and scalability. The client tier is the topmost layer where users interact through web browsers, desktop software, or mobile

applications. The web tier handles client requests and responses using technologies such as Servlets and JSP. The business tier contains the actual business logic, implemented through components like Enterprise JavaBeans (EJBs) or CDI-managed beans. Finally, the data tier interacts directly with databases using JPA or JDBC to perform operations like data retrieval and storage. Each of these tiers communicates with the others through well-defined interfaces, ensuring modularity, scalability, and maintainability.

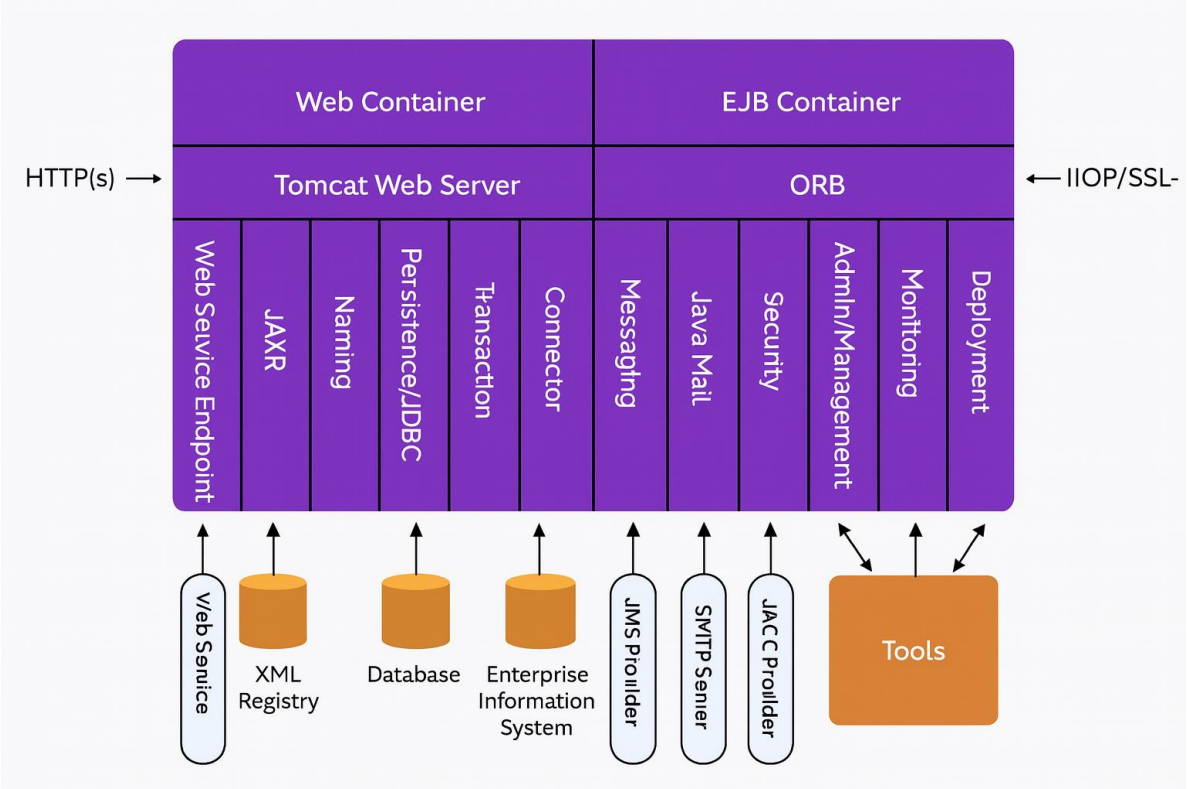


Fig. 3.2 Architecture of Application Server.

Popular Java EE Application Servers

There are several popular application servers available for Java EE (Jakarta EE), both open-source and commercial. GlassFish, developed initially by Sun Microsystems and now maintained by the Eclipse Foundation, is the official reference implementation of Jakarta EE. It fully supports all EE specifications and is often used for testing and learning purposes. WildFly (previously known as JBoss) is another widely used open-source application server developed by Red Hat. It is known for its high performance, lightweight modular design, and ability to handle large enterprise deployments.

Another lightweight option is Apache TomEE, an enhanced version of Apache Tomcat that adds full Java EE features like EJB, JPA, and JAX-RS, making it ideal for smaller enterprise or RESTful web applications. Commercial options include IBM WebSphere and Oracle

WebLogic, both of which are powerful, feature-rich servers used in industries that require high security and transaction reliability. WebSphere is often chosen by large financial institutions, while WebLogic is popular for enterprise resource planning (ERP) and customer relationship management (CRM) systems. A newer server, Payara, is derived from GlassFish and provides cloud-ready features suitable for modern microservices and container-based applications.

Example of How an Application Server Works

Consider a real-world example of an online shopping application built using Java EE. When a user opens the shopping site and logs in, the web layer of the application server (using Servlets or JSF) handles the request. The server then calls components in the business layer, such as EJBs, to validate the login credentials and retrieve user details. If the user adds items to the cart, the application server manages the session and stores this information securely. When the user proceeds to checkout, the server coordinates multiple actions verifying the payment, updating inventory, and confirming the order all within a single transaction to maintain consistency. If any part of the process fails (for example, a payment issue), the entire transaction is rolled back automatically. Throughout this process, the application server ensures proper resource management, transaction control, and data security without any manual coding by the developer.

Advantages of Using an Application Server

Application servers provide several advantages that make them indispensable for enterprise development. They greatly simplify development by managing low-level system details such as resource allocation, threading, and transactions. Developers can therefore focus on implementing business logic rather than infrastructure. They also offer portability, meaning a Java EE application can run on any compliant server without modification. Their built-in security features ensure safe data handling and access control, while their scalability and load balancing capabilities enable applications to serve thousands of users simultaneously without performance degradation. Moreover, the modular design of Java EE components allows for reusability and easy maintenance, making enterprise systems more cost-effective and efficient over time.

Real-World Applications of Application Servers

Application servers are used across a wide range of industries. In the banking sector, they manage secure financial transactions and customer data for online banking systems, such as those built on WebLogic or WebSphere. In e-commerce, they power online shopping portals like Amazon or Flipkart, handling product catalogs, payments, and user sessions efficiently using WildFly or TomEE. The healthcare industry uses application servers such as Payara or GlassFish to manage electronic medical records, patient data, and hospital operations securely. Government and public sector organizations also rely on application servers to

provide stable, scalable platforms for citizen services such as tax filing, identity management, and e-governance systems.

Containers

The word **container** is used in two closely related ways in enterprise Java conversations, and it helps to keep them distinct. In **Java EE / Jakarta EE**, a container is a runtime environment provided by the application server that manages a specific kind of component (for example a Web container for Servlets/JSP/JSF or an EJB container for Enterprise JavaBeans). That container supplies services such as lifecycle management, dependency injection, transaction coordination, security, JNDI lookups and remote connectivity so developers can write business logic without implementing those low-level services themselves.

In modern infrastructure and DevOps, **containerization** refers to packaging an application and everything it needs (libraries, configuration, binaries) into a lightweight, isolated runtime unit that runs on the host OS kernel. These OS-level containers (Docker, Podman, etc.) are designed for portability, fast startup, and consistent behavior across environments.

Java EE "containers" what they provide and why they matter

In the Java EE model, containers are logical environments inside the application server that host Java components and provide configurable, enterprise-grade services. When you assemble an application into Java EE modules and deploy them, each module is deployed into a matching container that handles the routine concerns for you. For example, a **Web container** instantiates and manages Servlets, maps URLs to those Servlets, manages HTTP sessions, and integrates with security realms. An **EJB container** instantiates EJBs, intercepts method calls to provide transactions and concurrency control, mediates security checks, and supplies remote invocation mechanics so a client can call an EJB as if it were local.

Because the container supplies services such as **transaction management**, **security enforcement**, **JNDI lookups**, **resource pooling (database connections, threads)** and **lifecycle callbacks**, developers focus on business logic. Containers also let you configure behavior per environment: the same enterprise bean might run with stronger DB privileges in production and more restricted access in a test environment, simply by changing container configuration rather than code. In short, Java EE containers are the runtime scaffolding that makes component-based enterprise programming practical and portable.

Example (Java EE): you deploy an EJB that implements order processing. The EJB container will automatically start/stop instances, manage transactions around business methods (so calls either commit or rollback as a unit), and apply security rules before letting a user invoke the method,

OS-level containers (containerization)

Containerization packages the application and its dependencies into an image that can be run as an isolated instance on any compatible host. The image contains only what the app needs (libraries, config, runtime) and shares the host operating system kernel with other containers. Because there is no full guest OS per container, containers are much lighter and faster to start than virtual machines.

Containerization is widely used for **microservices** (each microservice runs in its own container), **CI/CD pipelines** (build once, test and deploy the same image everywhere), **cloud-native** apps (portable across cloud providers), **dev/test environments** (reproducible), and **modernizing legacy apps** (packaging old apps so they run consistently on new infra).

Example (OS containers): you package a Java web application with a specific JDK and needed native libraries into a Docker image. That image runs the same on your laptop, a CI server, or a Kubernetes cluster eliminating "works on my machine" problems.

How OS containers work the main building blocks

At a high level, an OS container is nothing mystical: it is an isolated process (or set of processes) running on the host that has its own view of certain resources (filesystem, network interfaces, process IDs, etc.) and enforced resource limits. The container is launched from an **image** (a layered, read-only filesystem plus metadata and a command to run). The container runtime creates an isolated environment and runs the image's command as a process in that environment.

At a technical level this isolation and efficiency are achieved by several kernel features and supporting components:

- **Namespaces**: kernel feature that gives a process its own isolated view of global system resources. There are namespaces for process IDs (PID), networking (NET), mount points (MNT), interprocess communication (IPC), UTS (hostname), and user IDs (USER). Namespaces make processes inside a container see their own PID tree, their own network interfaces, and their own mount points so they appear isolated from the rest of the system.
- **cgroups (control groups)**: kernel facility that limits and accounts for resource usage (CPU, memory, disk I/O, network) for a group of processes. cgroups ensure a container cannot consume all host resources and enable fine-grained resource quotas.
- **Union/overlay filesystems**: container images are built as layered filesystems (each Dockerfile instruction adds a layer). Union filesystems (OverlayFS, AUFS) present these layers as one coherent filesystem to the container. Layering makes images compact and makes builds and caching efficient.
- **Container runtime and toolchain**: tools like Docker, containerd, and CRI-O coordinate image creation, storage, and launching. They interface with the kernel (namespaces, cgroups) to set up the container environment. The Open Container

Initiative (OCI) defines standard formats for images and runtimes so different tools interoperate.

- **Image registries**: images are pushed to and pulled from registries (Docker Hub, private registries). The registry stores image layers and metadata so environments can fetch the exact artifact to run.
- **Network model**: container runtimes create virtual network interfaces and bridge them to the host network or provide overlay/SDN networking (flannel, Calico) in distributed clusters. This gives containers IP addresses and controlled connectivity.

So when you docker run an image, the runtime: (1) pulls the image layers, (2) creates a new writable layer for that container, (3) sets up namespaces and cgroups, (4) mounts the overlay filesystem, (5) sets up networking, and (6) executes the image's entrypoint process in that isolated environment.

Container image lifecycle and how you operate them

Container usage typically follows a predictable lifecycle: **build**, **store**, **distribute**, **run**, **monitor**, and **update**. You author a Dockerfile (or similar) that describes how to build the image which base image to use, what files to copy, and which command to run. The image is built into layered artifacts and pushed to a registry. At deployment time the runtime pulls the image to the target host and instantiates one or many containers from that image. In production you don't run containers manually for scale; you use an orchestrator (Kubernetes, Docker Swarm, or similar) that schedules containers across many hosts, performs health checks, restarts failed containers, and scales replicas up or down.

Example workflow: Developers build an image for a Java microservice, push it to an internal registry as myorg/orders:1.2.0. The CI pipeline runs tests using that image. Kubernetes pulls myorg/orders:1.2.0 and creates pods, which the cluster exposes through a Service and an Ingress for external traffic.

How Java EE containers relate to OS containers synergy and differences

Although they share the same English word, **Java EE containers** and **OS containers** operate at different levels. Java EE containers are software runtime components inside an application server that provide services to Java components. OS containers are operating-system level process isolation units that package and run whole applications (including application servers). In modern deployments you commonly run a Java EE application server (GlassFish, WildFly, Payara, etc.) inside an OS container. That combines the enterprise services of the Java EE container with the portability and deployment benefits of OS containers.

Running a Java EE app inside OS containers makes it easier to scale the server or migrate between environments, while the Java EE container continues to provide transaction, security, and lifecycle services inside each container instance.

Practical example: deploying a Jakarta EE app in containers

Suppose you have a Jakarta EE application packaged as a WAR or EAR that expects an application server. A typical containerized deployment would start from a base image that contains a JRE/JDK and the application server (for example an OpenJDK + WildFly image). Your Dockerfile copies the WAR into the server's deploy directory and configures container startup to launch the application server. The final image is pushed to a registry and run in production, where an orchestrator runs multiple replicas, loads balances HTTP traffic across instances, and manages rolling updates with zero downtime.

This approach lets you test locally with the exact same image used in production and scale by simply increasing replica counts in the orchestrator.

Security, observability and operational considerations

Containers provide isolation but not perfect security by themselves: because they share the host kernel, a kernel vulnerability can allow container escape. Best practices include running containers with least privilege (non-root user inside the container), minimizing image size and installed packages, using signed and scanned images, applying vulnerabilities patches, and using kernel hardening and container security policies (AppArmor, SELinux). Observability is also crucial: instrument containers for logging, metrics and tracing so you can monitor performance, resource usage (via cgroups), and application behavior. Use health checks and readiness probes so orchestrators can manage lifecycle reliably.

Orchestration and scaling Kubernetes and clustering

At small scale you run single containers, but enterprise workloads need orchestration. Kubernetes is the de-facto orchestrator: it groups containers into pods, schedules them on nodes, manages desired state (number of replicas), handles service discovery, and supports rolling updates and self-healing. When you deploy Jakarta EE applications in production, you typically run multiple containerized server instances behind a load balancer and let Kubernetes manage scaling, updates, and failure recovery.

3.3 Servlet API and Lifecycle

Servlet API

Servlets are Java programs that run on a **Java-enabled web server or application server**. They are used to handle and process client requests coming from a web browser (or any HTTP client), generate appropriate responses, and send these responses back to the client through the web server. In simpler terms, servlets help developers create **dynamic web applications** using Java.

When building Java web applications, developers use **Servlets** as the foundation. To create a servlet, we rely on the **Servlet API**, which contains all the necessary classes and interfaces required to develop, deploy, and manage servlet-based applications. The Servlet API is divided into two main packages:

1. javax.servlet
2. javax.servlet.http

javax.servlet Package

The **javax.servlet** package provides interfaces and classes that support **generic, protocol-independent servlets**. This means servlets developed using this package can be used with various communication protocols, not just HTTP.

This package defines the **contracts** between a servlet class and the **runtime environment** provided by the servlet container (the part of an application server that runs and manages servlets). It contains classes and interfaces that handle servlet lifecycle management, event notifications, and input/output operations.

Important Classes in javax.servlet Package

- **GenericServlet:** This is an abstract class used to create generic, protocol-independent servlets. It implements most of the servlet lifecycle methods and provides a foundation for custom servlet development.
- **ServletContextEvent:** Represents events related to the lifecycle of the servlet context, such as initialization or destruction of the web application.
- **ServletContextAttributeEvent:** Used to generate notifications when attributes in the servlet context are added, removed, or replaced.
- **ServletInputStream:** Provides an input stream for reading binary data sent by the client.
- **ServletOutputStream:** Provides an output stream for sending binary data to the client.
- **ServletRequestEvent:** Indicates lifecycle events for a servlet request.
- **ServletRequestAttributeEvent:** Generates notifications when request attributes are modified during a request's lifecycle.
- **ServletRequestWrapper / ServletResponseWrapper:** These classes provide standard implementations of request and response interfaces. Developers can extend them to modify or adapt the behavior of requests and responses.

Important Interfaces in javax.servlet Package

- **Servlet:** The main interface that all servlets must implement. It defines essential methods such as init(), service(), and destroy() that control the servlet lifecycle. Developers typically create servlets by extending GenericServlet or HttpServlet, which implement this interface.

- **ServletConfig:** Defines a configuration object created by the servlet container during servlet initialization. It passes initialization parameters to the servlet.
- **ServletContext:** Represents the web application context and provides methods for communication between the servlet and the container. It allows access to application-level parameters defined in web.xml.
- **RequestDispatcher:** Provides a mechanism to forward requests or include responses from another servlet, JSP, or HTML file within the same application.
- **Filter, FilterChain, FilterConfig:** Used to implement filters that preprocess requests or post-process responses. Filters can perform operations such as logging, authentication, or compression before or after a servlet is executed.
- **ServletContextListener / ServletContextAttributeListener:** Receive notifications when the servlet context or its attributes change.
- **ServletRequest, ServletResponse:** Define the structure for request and response objects used by servlets to receive client information and send results back.
- **ServletRequestListener / ServletRequestAttributeListener:** Allow monitoring of request creation, destruction, or attribute modifications during a servlet's operation.

Exceptions in javax.servlet Package

- **ServletException:** A general exception class thrown by a servlet when it encounters an error while processing a request.
- **UnavailableException:** Indicates that a servlet or filter is temporarily or permanently unavailable, preventing it from handling further requests.

javax.servlet.http Package

The **javax.servlet.http** package provides interfaces and classes specific to the **HTTP protocol**, which is the most common protocol used in web development. It defines how a servlet interacts with clients over HTTP, handling requests and generating appropriate responses.

This package builds upon the generic javax.servlet framework and adds classes and interfaces that are specific to **HTTP requests, responses, sessions, and cookies**.

Important Classes in javax.servlet.http Package

- **HttpServlet:** An abstract class that extends GenericServlet and provides a framework for creating HTTP-specific servlets. Developers subclass this class and override methods such as doGet(), doPost(), doPut(), and doDelete() to handle different HTTP request types.
- **Cookie:** Represents small pieces of data sent from a server to a client's browser and stored locally. Cookies are used for session tracking and personalization.
- **HttpSessionEvent / HttpSessionBindingEvent:** Represent events related to session creation, modification, and invalidation.

- **HttpServletRequestWrapper / HttpServletResponseWrapper:** Provide adaptable implementations of the request and response interfaces that can be extended to customize behavior.

Important Interfaces in javax.servlet.http Package

- **HttpServletRequest:** Provides methods to access client request information, such as headers, parameters, cookies, and session details. It extends the ServletRequest interface.
- **HttpServletResponse:** Defines methods for sending HTTP responses back to the client, including setting response headers, status codes, and output content.
- **HttpSession:** Provides an interface to identify a user session across multiple requests and store session-specific data.
- **HttpSessionListener/HttpSessionAttributeListener/HttpSessionActivationListener:** Allow developers to receive notifications about session creation, destruction, and changes to session attributes.
- **HttpSessionBindingListener:** Notifies an object when it is bound to or unbound from a session.

GenericServlet Class

The **GenericServlet** class is part of the javax.servlet package. It is an **abstract, protocol-independent base class** that implements most of the methods from the Servlet and ServletConfig interfaces, and also implements Serializable.

Developers extend GenericServlet when creating protocol-neutral servlets. The only method that must be implemented is the **service()** method, which defines how the servlet processes client requests. The GenericServlet class simplifies servlet creation by providing default implementations for lifecycle methods such as init() and destroy().

Example:

```
public class MyGenericServlet extends GenericServlet {
   public void service(ServletRequest req, ServletResponse res)
         throws ServletException, IOException {
      res.getWriter().println("Hello from Generic Servlet!");
   }
}
```

In this example, the developer only overrides the service() method to handle client requests.

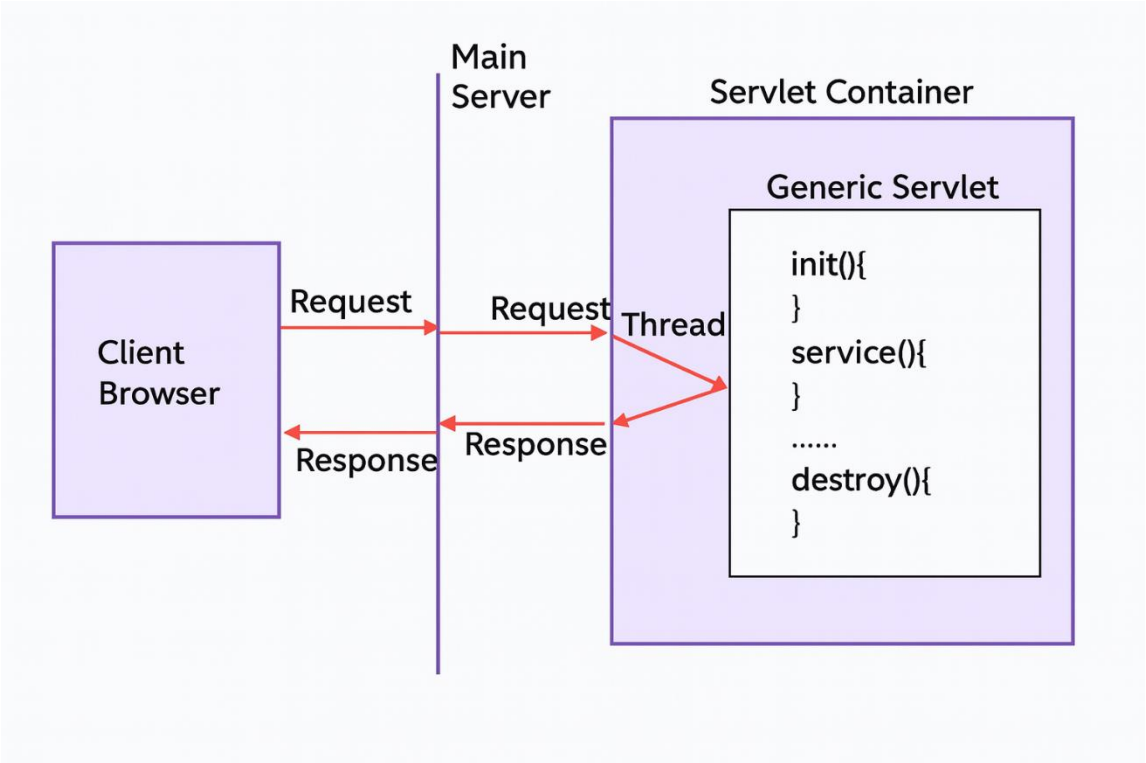


Fig. 3.3 GenericServlet Class.

HttpServlet Class

The **HttpServlet** class is defined in the javax.servlet.http package. It extends the GenericServlet class and provides **HTTP protocol-specific functionality**. Developers subclass HttpServlet to create servlets that handle web requests sent via the HTTP protocol.

An HTTP servlet overrides one or more of the following methods depending on which HTTP methods it supports:

- **doGet()** – Handles HTTP GET requests, typically used for retrieving data or displaying information.
- **doPost()** – Handles HTTP POST requests, used for submitting form data or updating server resources.
- **doPut()** – Handles HTTP PUT requests, used for updating resources on the server.
- **doDelete()** – Handles HTTP DELETE requests, used for deleting resources.

Additional lifecycle methods like **init()** and **destroy()** are used for resource management, while **getServletInfo()** can provide metadata about the servlet, such as author or version details.

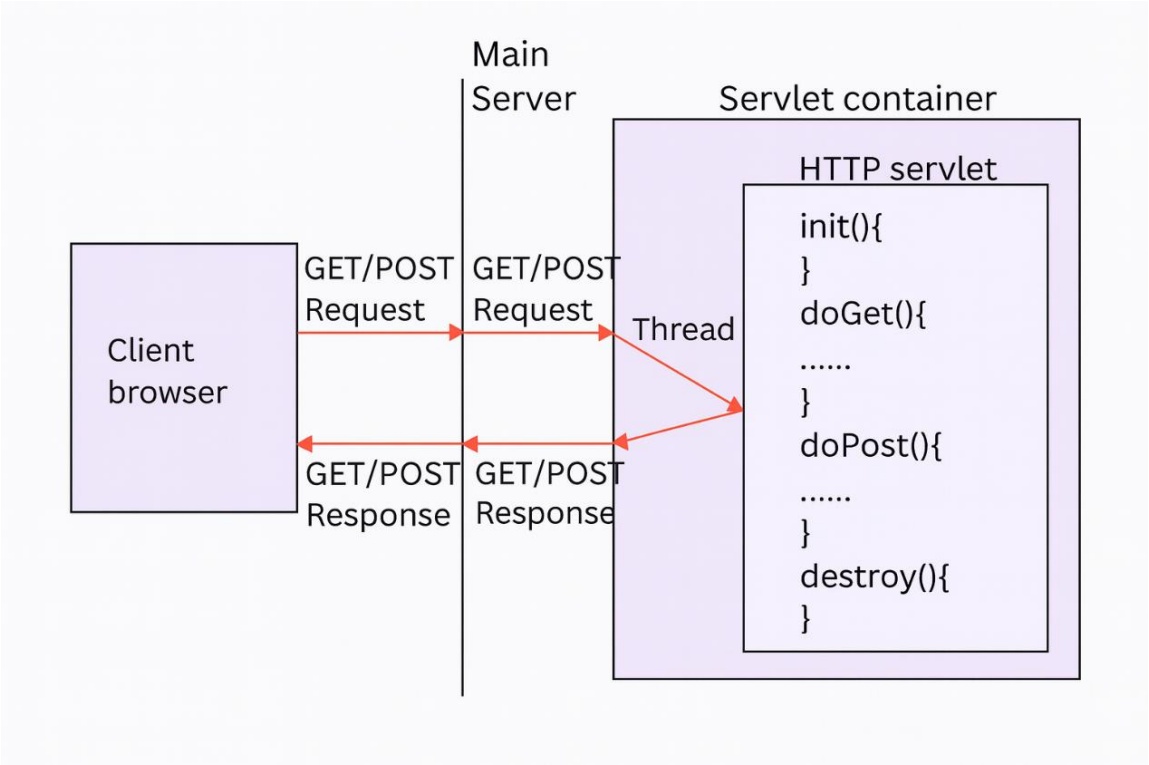


Fig. 3.4 HttpServlet Class

Example:

```
public class MyHttpServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      response.getWriter().println("Hello from HTTP Servlet!");
   }
}
```

Here, the doGet() method is overridden to handle HTTP GET requests and generate a response.

Lifecycle

A **Servlet Life Cycle** defines the complete process from a servlet's **creation to destruction**. It includes how a servlet is **loaded, initialized, executed**, and finally **terminated** by the container. Every servlet passes through specific stages managed by the **Servlet Container** (such as Apache Tomcat, GlassFish, or WildFly), which handles all lifecycle events automatically.

The Servlet API provides specific **lifecycle methods** that a developer can override to perform initialization, request handling, and cleanup tasks. Understanding this cycle is essential for developing efficient and resource-safe web applications.

A typical Servlet Life Cycle begins when an HTTP request from a client (such as a web browser) reaches the web server. The web server then delegates this request to the Servlet Container, which is responsible for managing all servlet-related operations.

When the container receives the request, it first checks whether the corresponding servlet class is already loaded into memory. If it is not, the container loads and instantiates the servlet before invoking its service() method. This method is the core of the servlet, responsible for handling client requests and generating appropriate responses.

The servlet container efficiently manages multiple client requests by creating a separate thread for each request. Each thread executes the service() method of the same servlet instance concurrently. This multithreaded approach allows the servlet to process many requests at the same time, ensuring high performance and responsiveness.

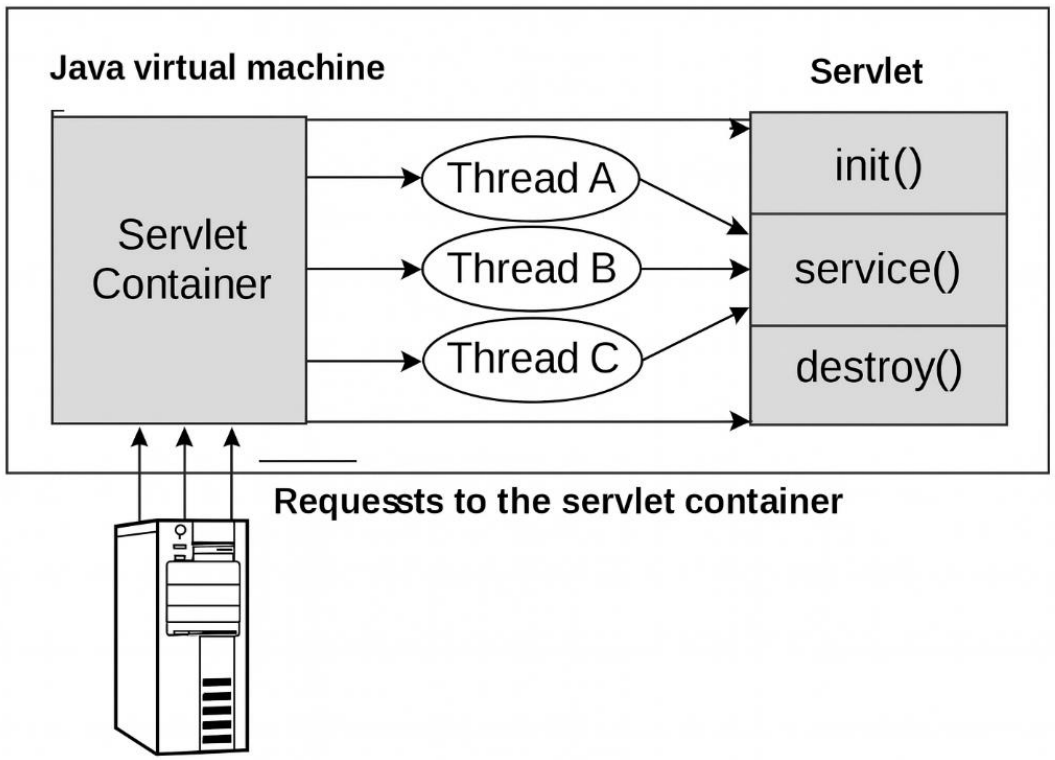


Fig. 3.5 Typical servlet life-cycle scenario.

Main Stages of Servlet Life Cycle

The life cycle of a servlet consists of **four major stages**:

1. **Loading the Servlet**
2. **Initializing the Servlet**
3. **Handling Client Requests**
4. **Destroying the Servlet**

These stages are managed by the **Servlet Container**, which ensures that the servlet runs efficiently, serving multiple client requests concurrently.

1. Loading a Servlet

The first step in the life cycle is **loading and instantiation**. When the web application starts, or when the servlet is requested for the first time, the servlet container performs the following tasks:

- **Class Loading:** The servlet container loads the servlet class into memory using the class loader.
- **Instantiation:** The container creates an instance of the servlet by invoking its **no-argument constructor**.

A servlet can be loaded in two ways:

- **Eager Loading:** The servlet is loaded during server startup if configured with the <load-on-startup> element in web.xml.
- **Lazy Loading:** The servlet is loaded only when it receives the first client request.

At this stage, the servlet object is created but not yet initialized.

2. Initializing a Servlet

After the servlet is instantiated, the container calls its **init()** method to perform initialization. This method is invoked **only once** in the entire lifecycle of the servlet.

Purpose:

The init() method is used to perform **one-time setup tasks** such as:

- Establishing database connections
- Loading configuration data
- Initializing resources needed throughout the servlet's lifetime

Syntax:

```
public void init() throws ServletException {
   // Initialization code (executed once)
}
```

Key Points:

- Called once after servlet creation.
- Can throw ServletException in case of initialization failure.
- Initialization should not be done in the constructor because constructors cannot throw checked exceptions.

Example:

```
@Override
public void init() throws ServletException {
   System.out.println("Servlet initialized successfully.");
}
```

After initialization, the servlet is ready to handle incoming requests.

3. Handling Client Requests

Once the servlet is initialized, it begins processing **client requests**. This stage continues until the servlet is removed from memory.

Request and Response Objects

Each client request is represented by a ServletRequest object, and the corresponding response is represented by a ServletResponse object.

For HTTP-based applications, these are specialized as:

- HttpServletRequest
- HttpServletResponse

The servlet container creates these objects for every request and passes them to the servlet's **service()** method.

The service() Method

The service() method is the **core of the servlet**. It receives the request, processes it, and sends a response back to the client.

Syntax:

```
public void service(ServletRequest req, ServletResponse res)
      throws ServletException, IOException {
   // Request handling logic
}
```

In the case of an **HTTP servlet**, the service() method automatically determines the type of request (GET, POST, PUT, DELETE) and delegates it to the corresponding method:

- doGet() → Handles HTTP GET requests.
- doPost() → Handles HTTP POST requests.
- doPut() → Handles HTTP PUT requests.
- doDelete() → Handles HTTP DELETE requests.

Each request is typically handled in a **separate thread**, allowing multiple clients to interact with the servlet concurrently.

Example:

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println("<h3>Welcome! This is a GET request handled by Servlet.</h3>");
}
```

4. Destroying a Servlet

The final stage occurs when the servlet container decides to remove the servlet from memory usually during application shutdown or when resources must be freed.

The container first waits for all active request threads to finish and then calls the **destroy()** method.

Purpose:

To release or close resources acquired during initialization, such as:

- Database connections
- File handles
- Network sockets
- Cached objects or threads

Syntax:

```
public void destroy() {
   // Cleanup code
}
```

After executing this method, the servlet instance becomes **eligible for garbage collection** by the JVM.

Example:

```
@Override
public void destroy() {
   System.out.println("Servlet is being destroyed. Resources released.");
}
```

3.4 Handling Client Requests and Responses

When developing web applications in Java, one of the most crucial concepts is understanding how to handle **client requests** and **server responses**. The communication between a client (such as a web browser, mobile app, or REST API consumer) and a server takes place using the **HTTP protocol**. This process typically involves the client sending a request message, the server interpreting it, performing necessary operations, and then generating a structured response.

In Java, this process is managed by the **Servlet API**, which provides a standard mechanism for creating web applications. A servlet acts as the **controller** that handles all the incoming client requests, processes them using application logic, and returns responses in the form of HTML pages, JSON data, XML, or any other suitable format.

Frameworks like **Spring Boot** are built on top of the servlet architecture. They simplify the request–response mechanism through annotations and abstractions, but underneath, the same servlet principles apply.

The Role of the Servlet Container

Before understanding how requests and responses are handled, it is important to understand the **Servlet Container** (also known as the Web Container). The servlet container is a part of the Java Web Server (such as Apache Tomcat, Jetty, or GlassFish) that is responsible for:

1. Managing the lifecycle of servlets (initialization, execution, and destruction).
2. Mapping incoming client requests to appropriate servlets based on configuration.
3. Creating and managing objects like HttpServletRequest and HttpServletResponse.

4. Handling multithreading by assigning each request to a new or existing thread.

Whenever a client sends an HTTP request to a server, the servlet container performs the following sequence:

- It receives the request from the web server.
- It identifies which servlet should handle the request based on the URL pattern defined in the web.xml deployment descriptor or servlet annotations.
- It creates instances of HttpServletRequest and HttpServletResponse objects.
- It passes these objects to the appropriate method of the servlet, such as doGet() or doPost().
- After the servlet finishes processing, the container sends the response back to the client.

This structure ensures that developers can focus on **application logic**, while the container manages complex networking, threading, and protocol details.

The HttpServletRequest Object

The HttpServletRequest object represents the client's request. It is created automatically by the servlet container and passed to the servlet when a request arrives. This object encapsulates all the data sent from the client and provides several methods to access it.

Some of the most important data that can be retrieved from this object include:

- **HTTP Method:** Using request.getMethod() you can determine whether the request is a GET, POST, PUT, or DELETE request.
- **Request Parameters:** Parameters passed via URL query strings or HTML form data can be retrieved using request.getParameter("paramName"). For example, if the URL is /hello?name=John, then request.getParameter("name") returns John.
- **Headers:** HTTP headers contain meta-information about the request. You can access them with request.getHeader("User-Agent") or request.getHeaderNames().
- **Body Content:** In POST or PUT requests, clients often send large data or JSON payloads. The body can be read using request.getReader() for text or request.getInputStream() for binary content.
- **Cookies and Sessions:** The getCookies() and getSession() methods help manage user state and track interactions across multiple requests.
- **Client Information:** Details such as the client's IP address or hostname can be accessed with getRemoteAddr() and getRemoteHost().

Thus, the HttpServletRequest object serves as a **gateway to all incoming data** from the client.

Processing the Request in the Servlet

Once the servlet receives the HttpServletRequest and HttpServletResponse objects, it begins processing the request. A servlet extends the HttpServlet class and overrides one or more of its lifecycle methods to handle specific types of HTTP requests.

The most commonly used methods include:

- **doGet():** Handles HTTP GET requests, which are generally used to retrieve data or display resources.
- **doPost():** Handles POST requests, typically used to send data to the server, such as form submissions or JSON objects.
- **doPut():** Used to update existing data on the server.
- **doDelete():** Used to remove resources.

When the servlet container receives a request, it determines which HTTP method is used and calls the corresponding method on the servlet. Inside these methods, developers write logic to perform the required actions, such as database queries, file handling, data validation, or computation.

For example, if a user submits a form with their name, the servlet can extract the name parameter and use it to display a personalized message or store it in a database.

The HttpServletResponse Object

After processing the client request, the servlet prepares an appropriate response using the HttpServletResponse object. This object allows the servlet to control every aspect of the HTTP response that will be sent back to the client.

The main functionalities of the HttpServletResponse object include:

- **Setting the Status Code:** Using response.setStatus(HttpServletResponse.SC_OK) or response.sendError(HttpServletResponse.SC_NOT_FOUND) to inform the client about the result of their request.
- **Setting Headers:** Response headers provide additional metadata about the response. For instance, response.setHeader("Cache-Control", "no-cache") prevents caching, while response.setContentType("application/json") specifies that the response contains JSON data.
- **Writing Content:** To send content to the client, the servlet uses a PrintWriter for text-based data or a ServletOutputStream for binary data. For example, response.getWriter().println("<h1>Hello User</h1>") sends HTML content.
- **Redirection:** If the servlet needs to redirect the client to another resource, it can call response.sendRedirect("/home").
- **Cookies and Sessions:** The servlet can add cookies to the response using response.addCookie() for managing user sessions or preferences.

In short, HttpServletResponse allows complete control over the output sent to the client, including status, headers, and content.

Example: A Complete Request-Response Flow

Here is an example that shows the complete process of handling a client request and generating a response using servlets.

```
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet("/greet")
public class GreetingServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      String name = request.getParameter("name");
      if (name == null || name.isEmpty()) {
         name = "Guest";
      }
      PrintWriter out = response.getWriter();
      out.println("<html><body>");
      out.println("<h2>Welcome, " + name + "!</h2>");
      out.println("</body></html>");
   }
}
```

In this example, when a client visits http://localhost:8080/greet?name=Prithiga, the servlet receives the request, extracts the “name” parameter, and sends back an HTML response containing the greeting. The servlet container automatically handles the creation and destruction of the request and response objects.

Handling Requests and Responses in Spring Boot

Spring Boot builds upon the Servlet API to make development simpler and faster. Instead of manually working with HttpServletRequest and HttpServletResponse, developers use **annotations** that tell Spring Boot how to handle requests.

For example, a simple controller in Spring Boot looks like this:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello(@RequestParam String name) {
        return "Hello, " + name + "!";
    }
}
```

When the client sends a request like /hello?name=Priya, Spring Boot automatically maps it to the sayHello() method, retrieves the "name" parameter, and returns the response as plain text.

Spring Boot also provides features like automatic JSON conversion, exception handling, and response status management using annotations like @ResponseStatus and @ExceptionHandler. This allows developers to focus on business logic rather than low-level HTTP details.

Managing Sessions and Cookies

Sessions and cookies are essential for maintaining user state in web applications. A **session** allows the server to store information for a user between multiple requests. For example, once a user logs in, their session can store their username or user ID. This is achieved using the HttpSession object.

Cookies are small text files stored on the client's machine. They can store information such as preferences or authentication tokens. In servlets, cookies are created and added to responses using the Cookie class and retrieved using the getCookies() method of HttpServletRequest.

Using both sessions and cookies properly enables smooth and personalized user experiences while ensuring secure communication between clients and servers.

Error Handling and HTTP Status Codes

When a servlet processes a request, various situations can arise such as missing resources or invalid input. Proper error handling is crucial. The servlet can send error codes and messages using methods like response.sendError(404, "Page Not Found").

Spring Boot simplifies error handling through global exception management. Developers can define a centralized class annotated with @ControllerAdvice that catches exceptions across the application and returns appropriate HTTP status codes and messages.

Security Considerations

Security is an integral part of handling client requests. Every piece of data received from the client must be validated to prevent attacks like cross-site scripting (XSS) or SQL injection. Cookies should be marked as secure and HTTP-only to prevent unauthorized access. Developers should also ensure all communication takes place over HTTPS. Frameworks like Spring Security further enhance application protection through authentication and authorization mechanisms.

Working with Databases through Servlets

Servlets are a vital component of Java web applications, enabling dynamic web pages that interact with databases. When a user interacts with a web interface, such as filling out a form or requesting data, the servlet acts as a bridge between the client and the database. To achieve this connection, Java uses **JDBC (Java Database Connectivity)**, a standard API that provides methods to connect to a database, execute SQL queries, and retrieve or manipulate results.

By integrating JDBC with servlets, web applications can handle operations such as user authentication, registration, data retrieval, updating records, and generating reports dynamically, making the web application both powerful and interactive.

Setting up the Environment

Before starting database programming with servlets, it is important to set up the required environment properly. This includes a **database system**, such as MySQL or PostgreSQL, a **JDBC driver**, and a **web server** such as Apache Tomcat.

If you are using **PostgreSQL**, you need to install the PostgreSQL JDBC driver, usually available as a JAR file named postgresql.jar. This file must be added to the WEB-INF/lib directory of your web application so that the servlet container can access it.

When using **Maven** or **Gradle**, the dependency can be managed directly in your project configuration file, making setup easier.

Database Preparation

To demonstrate the connection, let us assume we are using a PostgreSQL database named mobile_store. Inside this database, we create a table named mobiles to store mobile phone details such as model name, brand, and price.

```
CREATE TABLE mobiles (
    id SERIAL PRIMARY KEY,
    brand VARCHAR(50),
    model VARCHAR(50),
    price NUMERIC(10,2)
);

INSERT INTO mobiles (brand, model, price) VALUES
('Apple', 'iPhone 15', 89900.00),
('Samsung', 'Galaxy S24', 74999.00),
('OnePlus', 'OnePlus 12', 57999.00);
```

With this setup, we can now create servlets that will interact with this table to either **insert new records** or **fetch existing ones**.

Steps for JDBC Connection in Servlets

Working with databases using servlets involves a sequence of well-defined steps.

1. Importing the Required Packages

All database operations in Java are supported by the java.sql package. Therefore, we begin by importing the required classes such as Connection, DriverManager, Statement, PreparedStatement, and ResultSet.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

2. Registering the JDBC Driver

The next step is to register the JDBC driver that allows Java to communicate with the database. The driver class is loaded into memory using the Class.forName() method. This step tells the JVM to use the appropriate driver to handle database connections.

```
try {
    Class.forName("org.postgresql.Driver");
    System.out.println("PostgreSQL Driver Registered Successfully!");
} catch (ClassNotFoundException e) {
    System.out.println("Unable to load Driver class.");
    e.printStackTrace();
}
```

3. Establishing a Database Connection

Once the driver is registered, a connection is established to the database using the DriverManager.getConnection() method. This method requires the **database URL**, **username**, and **password**.

```
String URL = "jdbc:postgresql://localhost:5432/mobile_store";
String USER = "postgres";
String PASSWORD = "admin";
Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
```

If the credentials and URL are correct, a Connection object is returned, representing an active link to the database.

4. Creating a Statement Object

After the connection is established, a Statement or PreparedStatement object is created to execute SQL queries. A Statement is used for static queries, while a PreparedStatement is preferred for parameterized queries because it prevents **SQL injection** and improves performance.

```
Statement stmt = conn.createStatement();
```

5. Executing SQL Queries

Using the created statement, we can now execute SQL queries. For example, to retrieve all records from the mobiles table, we can use:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM mobiles");
```

For data manipulation operations such as **INSERT**, **UPDATE**, or **DELETE**, the executeUpdate() method is used.

```
String sql = "INSERT INTO mobiles (brand, model, price) VALUES ('Redmi', 'Note 13', 18999)";
int rows = stmt.executeUpdate(sql);
```

If the query modifies rows successfully, executeUpdate() returns the number of affected rows.

6. Processing the Result

If the query returns data, it can be accessed through a ResultSet object. The ResultSet acts as a cursor that moves through each record in the result set.

```
while (rs.next()) {
   System.out.println(rs.getString("brand") + " - " + rs.getString("model") + " : " + rs.getDouble("price"));
}
```

This loop reads all records from the mobiles table and prints their details.

7. Closing the Connection

It is important to close the database resources after usage to prevent memory leaks. The connection, statement, and result set objects must be closed in a finally block.

```
rs.close();
stmt.close();
conn.close();
```

Example 1: Inserting Data into Database

Below is a servlet that inserts data submitted from an HTML form into a PostgreSQL database.

HTML Form (insert.html):

```
<!DOCTYPE html>
<html>
<head>
   <title>Insert Mobile Data</title>
</head>
<body>
   <h2>Enter Mobile Details</h2>
   <form action="insertData" method="post">
      Brand: <input type="text" name="brand"><br><br>
      Model: <input type="text" name="model"><br><br>
      Price: <input type="text" name="price"><br><br>
      <input type="submit" value="Insert">
   </form>
</body>
</html>
```

Servlet Code (InsertDataServlet.java):

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
```

```
import java.sql.SQLException;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet("/insertData")
public class InsertDataServlet extends HttpServlet {
    private static final String URL = "jdbc:postgresql://localhost:5432/mobile_store";
    private static final String USER = "postgres";
    private static final String PASSWORD = "admin";

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String brand = request.getParameter("brand");
        String model = request.getParameter("model");
        String price = request.getParameter("price");

        try {
            Class.forName("org.postgresql.Driver");
            Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
            String sql = "INSERT INTO mobiles (brand, model, price) VALUES (?, ?, ?)";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, brand);
            pstmt.setString(2, model);
            pstmt.setDouble(3, Double.parseDouble(price));
            int rows = pstmt.executeUpdate();

            if (rows > 0) {
                response.getWriter().println("<h3>Data inserted successfully!</h3>");
            } else {
                response.getWriter().println("<h3>Insertion failed.</h3>");
            }

            pstmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
            response.getWriter().println("Error: " + e.getMessage());
        }
    }
}
```

In this servlet, when the user submits the form, the servlet receives the form data, establishes a connection to the database, and inserts a new record using a prepared statement.

Example 2: Retrieving Data from Database

Now, let us create another servlet that fetches all records from the database and displays them dynamically as HTML.

HTML Form (home.html):

```
<!DOCTYPE html>
<html>
<head>
<title>Home Page</title>
</head>
<body>
   <form action="fetch" method="get">
      <input type="submit" value="Fetch Mobile Details">
   </form>
</body>
</html>
```

Servlet Code (FetchDataServlet.java):

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet("/fetch")
public class FetchDataServlet extends HttpServlet {
   private static final String URL = "jdbc:postgresql://localhost:5432/mobile_store";
   private static final String USER = "postgres";
   private static final String PASSWORD = "admin";

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
        try {
            Class.forName("org.postgresql.Driver");
            Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM mobiles");

            out.println("<html><body>");
            out.println("<h2>Mobile Phone Details</h2>");
            out.println("<table
border='1'><tr><th>ID</th><th>Brand</th><th>Model</th><th>Price</th></tr>");

            while (rs.next()) {
                out.println("<tr><td>" + rs.getInt("id") + "</td>");
                out.println("<td>" + rs.getString("brand") + "</td>");
                out.println("<td>" + rs.getString("model") + "</td>");
                out.println("<td>" + rs.getDouble("price") + "</td></tr>");
            }

            out.println("</table></body></html>");

            rs.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            out.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

When the user clicks the “Fetch Mobile Details” button, this servlet retrieves all records from the database and displays them as a dynamic HTML table.

Exception Handling and Resource Management

Database operations are prone to exceptions such as incorrect SQL syntax, invalid credentials, or missing tables. Hence, it is recommended to use try-catch-finally blocks to manage these exceptions effectively. Additionally, database connections, statements, and result sets must be closed after use to prevent **memory leaks**. Java 7 introduced the **try-with-resources** feature that automatically closes resources, simplifying this process.

Advantages of Using JDBC with Servlets

Using JDBC within servlets provides several advantages:

1. It allows real-time interaction between the user interface and the backend database.
2. It ensures platform independence since JDBC supports multiple databases like MySQL, Oracle, and PostgreSQL.
3. It allows easy integration with frameworks like JSP and Spring MVC.
4. It supports transaction management and batch processing for high-performance applications.

3.5 Servlet Advanced Concepts: Request Dispatcher

The RequestDispatcher interface provides a simple but powerful mechanism for servlet collaboration. It allows one servlet or other web resource to delegate processing to another resource or to include the output of another resource inside the response. The two core operations are forward and include. Forward hands off the entire request and response to another target resource so that the target produces the final response. Include invokes the target resource and merges its output into the response that the calling resource is building. RequestDispatcher therefore supports modular design and reuse of dynamic or static resources such as servlets, JSP pages and HTML files.

How to obtain a RequestDispatcher object

There are three common ways to obtain a RequestDispatcher object. The first is to ask the servlet request for a dispatcher by path. This path may be relative to the current servlet or start with a slash to indicate the context root. The second is to ask the servlet context for a dispatcher by path. The third is to ask the servlet context for a named dispatcher where the name is the name used when the servlet was registered. In code these three approaches look like this.

```
// From the request using a context relative path
RequestDispatcher rd1 = request.getRequestDispatcher("/welcome");

// From the servlet context using a path
RequestDispatcher rd2 = getServletContext().getRequestDispatcher("/welcome");

// From the servlet context using a servlet name
RequestDispatcher rd3 = getServletContext().getNamedDispatcher("welcomeServlet");
```

Choosing between these options depends on how you deploy and map your servlets. A path based lookup is the most common. A named lookup is useful when you want to decouple the caller from the actual URL mapping.

Forward method and its semantics

The forward method transfers control completely to the target resource. When a forward is invoked the servlet container stops writing any content from the calling servlet to the response buffer and calls the target resource with the same request and response objects. The HTTP status and headers are not committed before calling forward. If the response has already been committed to the client, calling forward will throw an IllegalStateException. Any request attributes set on the request are visible to the target. Because forward changes which resource writes the final response, the client URL remains the same unless the target itself issues a redirect.

A typical use of forward is to validate user input in one servlet and then forward to a JSP for rendering.

```
// In a validation servlet
if (valid) {
   request.setAttribute("user", userObject);
   RequestDispatcher rd = request.getRequestDispatcher("/WEB-INF/views/profile.jsp");
   rd.forward(request, response); // profile.jsp renders the page using request attributes
} else {
   RequestDispatcher rd = request.getRequestDispatcher("/error.html");
   rd.forward(request, response);
}
```

Note that forwarding to a path under WEB INF is a common pattern to prevent direct browser access to JSPs that should only be used for rendering.

Include method and its semantics

The include method invokes the target resource and inserts the target output into the response stream at the exact point where include is called. The calling servlet remains responsible for the overall response. The target resource can read request parameters and attributes and it may write content to the response. After include returns, the calling servlet can continue writing more content to the response. Include sets dispatcher type to INCLUDE and also exposes special request attributes so the included target can discover the include context.

A common use of include is to assemble pages from reusable components such as headers, footers and navigation fragments.

```
// In a servlet building a page
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>");
RequestDispatcher header = request.getRequestDispatcher("/common/header.jsp");
header.include(request, response); // header output appears here
```

```
out.println("<div class='main'>");
// main content
out.println("</div>");
RequestDispatcher footer = request.getRequestDispatcher("/common/footer.jsp");
footer.include(request, response); // footer output appended
out.println("</body></html>");
```

Include is also used when a servlet wants to reuse the output of another servlet or JSP without giving up control of the response.

Relative and absolute paths and mapping rules

When you obtain a RequestDispatcher using request.getRequestDispatcher with a path that does not start with a slash it is treated as relative to the current request path. When the path starts with a slash it is interpreted relative to the current context root. Calling getServletContext().getRequestDispatcher requires a path that starts with a slash and is always relative to the context root. These rules affect how you structure your application and where you place JSPs and static resources.

Request attributes versus parameters and scope implications

RequestDispatcher forwards and includes share the same HttpServletRequest object. That means request attributes are a safe mechanism to pass data between the original servlet and the target resource. Parameters from the original HTTP request remain available to the target, but attributes provide a convenient programmatic channel for objects such as domain models, error messages or form data. For example:

```
request.setAttribute("errors", errorList);
RequestDispatcher rd = request.getRequestDispatcher("/login.jsp");
rd.forward(request, response);
```

On the other hand session attributes are stored at session scope and persist across multiple requests. Use request attributes for short lived data that belongs to a single request processing cycle.

Error handling and common exceptions

Forward must be invoked before the response is committed. If the servlet or container has already flushed the response headers to the client, calling forward will throw IllegalStateException. Include may be used after partial output, but it is good practice to control when content is committed by setting response buffer size or by avoiding direct flush calls. All dispatch operations can throw ServletException or IOException, so typical code wraps calls in try catch and either logs or forwards to an error page.

```
try {
    RequestDispatcher rd = request.getRequestDispatcher("/target");
    rd.forward(request, response);
} catch (ServletException | IOException e) {
    log("Dispatch failed", e);
    response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
```

Practical example with login and messages

Below is a concise login example using both include and forward semantics. The login servlet validates credentials and forwards to a welcome servlet on success. On failure it writes an inline message and then includes the original form so the user can retry.

```
// LoginServlet doPost
String name = request.getParameter("userName");
String password = request.getParameter("userPass");

if ("admin".equalsIgnoreCase(name) && "admin".equals(password)) {
    request.setAttribute("userName", name);
    RequestDispatcher rd = request.getRequestDispatcher("/welcome");
    rd.forward(request, response);
} else {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<p>Sorry incorrect user name or password please try again</p>");
    RequestDispatcher rd = request.getRequestDispatcher("/index.html");
    rd.include(request, response);
}
```

The welcome servlet reads the request attribute or parameter and produces the final greeting.

```
// WelcomeServlet doGet or doPost
String name = (String) request.getAttribute("userName");
if (name == null) {
    name = request.getParameter("userName");
}
response.getWriter().println("Welcome " + (name == null ? "Guest" : name));
```

RequestDispatcher related request attributes and error dispatch attributes

The servlet specification defines a set of request attribute names that are automatically set during forward and include operations and during error dispatch. These attributes allow the target to inspect the original request context. For forward the container sets attributes such as those for the original request URI, original context path and original servlet path. For include

the container sets attributes that indicate the include request URI and include servlet path. During error dispatch the container makes the exception, the status code and the original request URI available under standardized attribute names. These attribute names are available as constants documented in the servlet API for use by filters and error pages.

Examples of typical attribute names you may inspect include names that represent original request URI or error exception. Reading those attributes lets an error page display which resource caused the problem or log a full diagnostic.

When to prefer forward versus redirect

Forward keeps processing on the server and the client URL remains unchanged. Use forward to render views after server side processing or to chain processing between components. Redirect uses response.sendRedirect and sends a 302 status to the client asking the browser to issue a new request to a different URL. Redirect is required if you want the browser address bar to change or if you want to implement the Post redirect Get pattern to avoid duplicate form submission. Use redirect when the new location is in another application or when you want the client to see the new URL.

Security and design considerations

When forwarding to JSPs or other resources avoid forwarding to resources that the client could access directly if you want to enforce server side control. Placing view JSPs under WEB INF prevents direct browser access while allowing forwards from servlets. Avoid exposing sensitive data by not forwarding request attributes to untrusted components. Validate input and sanitize any data rendered by included resources to prevent cross site scripting attacks.

3.6 Session Management

Session management means keeping track of a user across multiple requests so the application can remember who the user is and what they were doing. Web traffic is inherently stateless which means each HTTP request stands alone. Session management restores continuity by associating a unique session with a client and storing data on the server side tied to that session. The servlet API exposes this capability through the HttpSession object which is the central mechanism for storing and retrieving user specific data across several page visits.

A session is a server side data container created when a user first interacts with the application. The servlet container generates a unique session identifier and typically provides it to the client in a cookie. On each subsequent request the container reads the identifier and loads the corresponding HttpSession object. Sessions are important because they allow user authentication to persist across pages, they let the server store temporary preferences and shopping cart contents, and they reduce friction by avoiding repeated logins or repeated data

entry. Sessions are also a boundary for applying security rules and for implementing logout or automatic expiration.

How to obtain and use HttpSession

The servlet API makes session handling simple. Call request.getSession to obtain the current session or to create a new one if none exists. Calling request.getSession(false) returns null if there is no existing session. Calling request.getSession(true) or request.getSession with no argument forces creation when necessary.

Example of creating a session and setting attributes:

```
HttpSession session = request.getSession(); // creates if absent
session.setAttribute("username", "alice");
session.setAttribute("cartSize", Integer.valueOf(3));
```

Example of reading and removing attributes:

```
HttpSession session = request.getSession(false); // do not create
if (session != null) {
   String username = (String) session.getAttribute("username");
   session.removeAttribute("cartSize");
}
```

Example of invalidating a session:

```
HttpSession session = request.getSession(false);
if (session != null) {
   session.invalidate(); // destroys session and all attributes
}
```

Example of changing session timeout:

```
HttpSession session = request.getSession();
session.setMaxInactiveInterval(30 * 60); // thirty minutes in seconds
```

These examples show how attributes provide an easy way to pass Java objects between servlets and JSP pages during the life of a session.

Common session tracking mechanisms

Cookies are the standard way to convey the session identifier from client to server. The container issues a cookie named JSESSIONID by default that contains the session id. If the client returns that cookie on subsequent requests, the container maps the id to the server side session object.

When cookies are not available there are two fallback options. One option is URL rewriting where the container appends the session id to links generated by response.encodeURL or response.encodeRedirectURL. The other option is hidden form fields where the id is embedded in forms and posted back. URL rewriting works for standard hyperlinks and form submissions but it exposes the session id in the URL which has security implications. Hidden form fields only work for form submissions and therefore are not a general replacement for cookies.

Example of URL rewriting:

```
String url = response.encodeURL("profile");
out.println("<a href=\"" + url + "\">View Profile</a>");
```

A typical login example with secure session handling

When implementing login it is best practice to avoid session fixation attacks by invalidating any existing session and creating a new one after authentication. Store the minimal required data and avoid putting large objects or sensitive data that does not need to remain in memory.

Login servlet example that protects against session fixation:

```
// after verifying credentials
HttpSession oldSession = request.getSession(false);
if (oldSession != null) {
    oldSession.invalidate();
}
HttpSession session = request.getSession(true); // new session
session.setAttribute("username", username);
session.setMaxInactiveInterval(20 * 60); // 20 minutes
response.sendRedirect("dashboard");
```

This pattern ensures the session id used before authentication cannot be reused by an attacker.

Using sessions for CSRF protection

Sessions are commonly used to store a CSRF token that a page renders into forms as a hidden field. When the form is submitted the server compares the token in the request with the token held in the HttpSession. If they match the request is valid.

Example token generation and validation:

```
// token generation when rendering form
HttpSession session = request.getSession();
String token = UUID.randomUUID().toString();
session.setAttribute("csrfToken", token);
```

```
out.println("<input type='hidden' name='csrfToken' value='" + token + "'/>");

// token validation in form handler
String tokenFromForm = request.getParameter("csrfToken");
String tokenInSession = (String) request.getSession(false).getAttribute("csrfToken");
if (tokenInSession == null || !tokenInSession.equals(tokenFromForm)) {
   response.sendError(HttpServletResponse.SC_FORBIDDEN, "Possible CSRF detected");
}
```

Storing the token in session keeps the secret server side while the form provides the value for comparison.

Session security considerations

Mark session cookies with HttpOnly and Secure flags to reduce risk. HttpOnly prevents client side scripts from reading the cookie, which mitigates some cross site scripting attacks. Secure ensures the cookie is sent only over HTTPS. Many containers allow enabling these flags in configuration, or you can set a cookie manually.

Example setting a cookie manually with security flags:

```
Cookie cookie = new Cookie("JSESSIONID", session.getId());
cookie.setHttpOnly(true);
cookie.setSecure(true);
cookie.setPath(request.getContextPath());
response.addCookie(cookie);
```

Always use HTTPS for authenticated sessions and avoid exposing session ids in URLs whenever possible. Regenerate session ids on privilege changes such as login, and clear sensitive data from session attributes before invalidation.

Session size, scalability and distributed environments

Sessions are stored on the server and can consume memory if they hold large objects or many concurrent users exist. Avoid placing big objects in session. For scalable applications deployed to multiple nodes in a cluster you have several choices. You can use sticky sessions so a user is always routed to the same server node. You can use session replication where the container replicates session data to other nodes. You can externalize sessions into a shared store such as an in memory datastore like Redis. Externalizing sessions is commonly used for cloud or container deployments because it decouples session lifetime from any single application instance and supports horizontal scaling.

Example of session lifecycle in a servlet scenario

When a user visits the site for the first time and calls request.getSession, the container generates an id and creates a new HttpSession. The servlet populates attributes with user specific values. As the user navigates the site the container locates the same session using the session id from cookie or URL rewriting. If the user is inactive longer than the configured timeout the container invalidates the session automatically and fires sessionDestroyed events for listeners. If the application explicitly calls session.invalidate the session is immediately removed and any subsequent request will have no associated session unless a new one is created.

Create sessions only when necessary. Use request.getSession(false) when simply checking for authentication to avoid creating many unused sessions. Store primitives and small value objects rather than large data. Invalidate sessions on logout and after major account changes. Regenerate session id on login to prevent fixation attacks. Set cookie flags HttpOnly and Secure and always use HTTPS for authenticated endpoints. Consider an external session store for distributed architectures.

3.7 Cookies and Http Session Interface

Cookies

In web applications, **cookies** are one of the most important techniques for maintaining state between client and server. The HTTP protocol itself is stateless every request sent from a client (like a browser) is treated as a completely new one by the server. To make a web application remember a user's information across multiple requests, **cookies** are used.

A cookie is a **small piece of textual information** (a key–value pair) stored by the browser on behalf of the server. When the client makes a request for the first time, the server sends a cookie back with the response. This cookie is stored in the client's browser. On every subsequent request to the same domain, the browser automatically includes the cookie in the request headers. This allows the server to recognize the returning client and retrieve stored information about the session, preferences, or user data.

How Cookies Work

When a client (browser) first sends a request to a web server, the server can attach a cookie to the HTTP response. The browser then stores this cookie locally.

For example, the HTTP response header might look like this:

Set-Cookie: userId=101; Max-Age=3600; Path=/; HttpOnly

On subsequent requests to the same server, the browser sends:

Cookie: userId=101

This way, the server identifies that this request belongs to the same user who previously logged in or performed a specific action.

If the cookie's ID matches an existing record, the server treats it as an **old user session**. If no matching cookie is found, the request is treated as **new**.

The Cookie Class in Java

In Java Servlets, cookies are handled using the Cookie class which is part of the javax.servlet.http package. Creating, sending, and reading cookies in Java is simple because the servlet container automatically manages cookie headers in HTTP requests and responses.

Creating a cookie:

```
Cookie cookie = new Cookie("user", "JohnDoe");
response.addCookie(cookie);
```

Retrieving cookies:

```
Cookie[] cookies = request.getCookies();
for (Cookie c : cookies) {
   if (c.getName().equals("user")) {
      String value = c.getValue();
   }
}
```

The servlet API provides methods to set, get, and manage cookies conveniently.

Important Methods in the Cookie Class

Method	Description
getName()	Returns the name of the cookie
getValue()	Returns the cookie value
setValue(String value)	Changes the value of the cookie
getMaxAge()	Returns the maximum age in seconds
setMaxAge(int seconds)	Defines how long the cookie should be stored

getPath()	Specifies the URL path to which the cookie applies
setPath(String path)	Sets the valid path for the cookie
getDomain()	Returns the domain for which the cookie is valid
setSecure(boolean flag)	Ensures the cookie is sent only over HTTPS
setHttpOnly(boolean flag)	Prevents client-side scripts from reading the cookie
clone()	Returns a copy of the cookie object

Table. 3.2 Important Methods in the Cookie Class.

These methods make cookies flexible for various purposes like authentication, personalization, and session persistence.

Example 1: Passing Data Between Servlets Using Cookies

Let's take a simple example of passing data (the institute name) from one servlet to another using cookies.

Step 1: HTML Form

```
<!DOCTYPE html>
<html>
<head>
   <title>Institute Name</title>
</head>
<body>
   <form action="servlet1" method="POST">
      <h2>Enter your institute's name:</h2>
      <input type="text" name="name">
      <input type="submit" value="Go!">
   </form>
</body>
</html>
```

When the user submits the form, the request goes to Servlet1.

Step 2: Servlet1 – Creating and Sending a Cookie

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.IOException;
import java.io.PrintWriter;
```

```
public class Servlet1 extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      String name = request.getParameter("name");
      out.println("<h1>Hello, welcome to " + name + "</h1>");
      out.println("<h2><a href='servlet2'>Go to Servlet 2</a></h2>");

      // Create cookie
      Cookie c = new Cookie("institute_name", name);
      response.addCookie(c);

      out.close();
   }
}
```

This servlet collects the name from the form, prints a greeting, and adds a cookie named "institute_name" to the response. Now the client's browser stores this cookie.

Step 3: Servlet2 – Reading the Cookie

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.IOException;
import java.io.PrintWriter;

public class Servlet2 extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      Cookie[] cookies = request.getCookies();
      boolean found = false;
      String name = "";

      if (cookies != null) {
         for (Cookie c : cookies) {
            if (c.getName().equals("institute_name")) {
               found = true;
               name = c.getValue();
```

```
            }
        }
    }

    if (found) {
        out.println("<h1>Welcome back, " + name + "!</h1>");
        out.println("<h2>Thank you for visiting again.</h2>");
    } else {
        out.println("<h1>New user detected. Please go to the home page and enter your institute name.</h1>");
    }

    out.close();
  }
}
```

Output Description

When you submit the form for the first time:

- Servlet1 creates a cookie with your institute name.
- The response displays “Hello, welcome to XYZ Institute.”

When you click on “Go to Servlet 2”:

- Servlet2 reads the cookie and displays:

  “Welcome back, XYZ Institute! Thank you for visiting again.”

If you open Servlet2 directly in incognito mode:

- No cookies are found.
- The output displays:

  “New user detected. Please go to the home page and enter your institute name.”

This demonstrates how cookies track user identity across servlets.

Real-World Applications of Cookies

1. User Authentication: Cookies store session identifiers that represent logged-in users.

For example, after login, a cookie like authToken helps the server recognize the user without requiring credentials again.

2. Personalization

Cookies remember preferences such as language, theme, or font size, making the user experience smoother.

3. Shopping Carts

E-commerce websites use cookies to store cart data temporarily even if the user navigates across multiple pages.

4. Analytics and Tracking

Cookies track user activity for analytics and advertising personalization.

Http Session Interface

In web applications, maintaining information about a user across multiple HTTP requests is known as **session tracking**. The HTTP protocol is **stateless**, meaning that each client request is treated as a new connection the server does not remember previous interactions. To overcome this limitation, servlets provide several mechanisms for session tracking, such as **cookies**, **URL rewriting**, **hidden form fields**, and the **HttpSession interface**.

Among these, the **HttpSession interface** is one of the most powerful and secure ways to maintain session state because it allows the server to store user-specific information for the duration of a session.

In simple terms, a **session** is the limited time interval in which a client and a server communicate continuously. When a session is created, the server generates a **unique session ID** for each user. This ID helps the server identify returning users and retrieve data associated with them.

When a client makes an HTTP request for the first time, the servlet container automatically creates a new HttpSession object and assigns it a unique ID (stored internally as a cookie named JSESSIONID). This ID is sent back to the client, which stores it in the browser. On subsequent requests, the browser automatically sends this ID to the server, allowing the server to identify the correct session.

Advantages of Using HttpSession

1. **Supports All Data Types:** You can store any kind of Java object in a session — strings, arrays, collections, or even custom objects.
2. **Independent of Browser:** Session management does not rely on the client's browser settings (unlike cookies).

3. **Secure:** Session data is stored on the server, not in the client's browser, which makes it safer.
4. **Convenient:** It automatically creates and tracks sessions, simplifying state management for developers.

Disadvantages of HttpSession

1. **Memory Overhead:** Since session data is stored on the server, it increases memory usage.
2. **Performance Overhead:** Each session object must be created, serialized, and deserialized when necessary, which adds a slight performance cost.

Commonly Used Methods of HttpSession

Method	Description
getSession()	Returns the current HttpSession object, creating one if it doesn't exist.
getSession(boolean create)	Returns the current session; creates a new one only if create is true.
getId()	Returns the unique ID assigned to the session.
getCreationTime()	Returns the timestamp when the session was first created.
getLastAccessedTime()	Returns the timestamp of the last client request associated with the session.
setAttribute(String name, Object value)	Stores an attribute (data) in the session.
getAttribute(String name)	Retrieves an attribute previously stored in the session.
removeAttribute(String name)	Removes an attribute from the session.
invalidate()	Destroys the session and removes all its attributes.
setMaxInactiveInterval(int seconds)	Sets the maximum time interval before the session expires.

Table. 3.3 Commonly Used Methods of HttpSession.

How HttpSession Works

The flow of session creation and management can be visualized as follows:

1. **User request:** The client sends a request to the server.
2. **Session creation:** The servlet container checks if a session already exists. If not, a new HttpSession object is created.
3. **Session ID generation:** A unique session ID is created and stored in a cookie (JSESSIONID) on the client's browser.
4. **Attribute storage:** Servlets can store data using setAttribute() and retrieve it using getAttribute().
5. **Session reuse:** For subsequent requests, the same session ID is sent back to the server, and the existing session is retrieved.
6. **Session termination:** When the user logs out or after a defined period of inactivity, the session is invalidated using invalidate()

Example : Session Tracking Between Two Servlets

This example demonstrates how data can be stored in a session in one servlet and retrieved in another servlet using HttpSession.

index.html

```
<html>
<head><title>Session Example</title></head>
<body>
<form action="servlet1">
   Name: <input type="text" name="userName"/><br/>
   <input type="submit" value="Submit"/>
</form>
</body>
</html>
```

First.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class First extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
```

```
        String name = request.getParameter("userName");
        out.println("<h1>Welcome, " + name + "</h1>");

        // Create or get existing session
        HttpSession session = request.getSession();

        // Set attribute in session
        session.setAttribute("uname", name);

        out.println("<a href='servlet2'>Visit Servlet 2</a>");
        out.close();
    }
}
```

Second.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Second extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Fetch existing session
        HttpSession session = request.getSession(false);

        if (session != null) {
            String name = (String) session.getAttribute("uname");
            out.println("<h2>Hello again, " + name + "!</h2>");
        } else {
            out.println("<h2>No session found. Please start from the home page.</h2>");
        }

        out.close();
    }
}
```

web.xml

```
<web-app>
  <servlet>
```

```
  <servlet-name>s1</servlet-name>
  <servlet-class>First</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>s1</servlet-name>
  <url-pattern>/servlet1</url-pattern>
 </servlet-mapping>

 <servlet>
  <servlet-name>s2</servlet-name>
  <servlet-class>Second</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>s2</servlet-name>
  <url-pattern>/servlet2</url-pattern>
 </servlet-mapping>
</web-app>
```

Output Description

1. **Step 1 – index.html:** The user enters their name and clicks "Submit".

Output:

Welcome, Greenfield!
Visit Servlet 2

2. **Step 2 – servlet2 (Second.java):** When the user clicks the "Visit Servlet 2" link, the second servlet fetches the session created earlier and retrieves the stored username.

Output:

Hello again, Greenfield!

3. **If the session has expired or user opens Servlet2 directly:** No session found. Please start from the home page.

Working with Files

Working with files is an essential capability in most applications. Files are used to store configuration data, logs, user uploads, cached results, exported reports, and much more. Java gives several complementary APIs for file I O ranging from the classic java.io streams to the

modern java.nio.file utilities. Below I explain concepts, patterns, pitfalls, and practical examples you can reuse in real projects.

Types of file access and when to use them

File access falls into a few broad categories. Text file processing is for human readable data such as CSV, logs, or plain configuration. Binary file processing is for images, archives, or serialized objects. Random access is needed when you must read or write arbitrary positions inside a file, for example with indexes or resumable downloads. Streaming is important for very large files that cannot be fully loaded into memory. Choose the approach that matches your size constraints and performance needs.

Paths and the file system API

Use java.nio.file.Path and java.nio.file.Files for most file system operations. Path is superior to the older java.io.File because it is immutable, composable, and integrates with a richer set of operations.

Example

```
import java.nio.file.*;

Path dir = Paths.get("/data/app");
Path file = dir.resolve("users.csv"); // safe composition
boolean exists = Files.exists(file);
```

Key useful Files methods include createDirectories, exists, readAllLines, write, copy, move, delete, and probeContentType. Many of these throw IOException, so handle or propagate appropriately.

Reading and writing small text files

For small files where memory is not a concern, Files.readAllLines and Files.write are concise and robust. Always specify a charset when working with text.

Example

```
Path path = Paths.get("config/settings.txt");
List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);

List<String> out = List.of("settingA=true", "settingB=42");
Files.write(path, out, StandardCharsets.UTF_8, StandardOpenOption.CREATE,
StandardOpenOption.TRUNCATE_EXISTING);
```

Streaming large files

For large files stream them rather than load them fully into memory. Use BufferedReader for text and InputStream for binary. Try with resources ensures streams are closed automatically.

Text streaming example

```
try (BufferedReader br = Files.newBufferedReader(path, StandardCharsets.UTF_8)) {
   String line;
   while ((line = br.readLine()) != null) {
      process(line);
   }
}
```

Binary streaming and copying

```
try (InputStream in = Files.newInputStream(src);
    OutputStream out = Files.newOutputStream(dst, StandardOpenOption.CREATE,
StandardOpenOption.TRUNCATE_EXISTING)) {
   byte[] buffer = new byte[8192];
   int n;
   while ((n = in.read(buffer)) > 0) {
      out.write(buffer, 0, n);
   }
}
```

Files.copy(src, dst, StandardCopyOption.REPLACE_EXISTING) is a ready made option when you only need to copy.

Character encoding and line endings

Always use a specified Charset like StandardCharsets.UTF_8. Relying on platform default charset leads to bugs when your code runs on different operating systems. Be aware of platform line ending differences. Use System.lineSeparator() when you generate text to be consumed on the same platform, but for cross platform format prefer \n and document the expected format.

Random access and seeking

Use RandomAccessFile or a FileChannel if you need to read or write at arbitrary offsets. FileChannel supports file locking and transfer to other channels.

Example reading at offset

```
try (RandomAccessFile raf = new RandomAccessFile("data.bin", "r")) {
```

```
    raf.seek(1024);
    byte[] buf = new byte[256];
    int read = raf.read(buf);
}
```

FileChannel example with lock

```
try (FileChannel ch = FileChannel.open(path, StandardOpenOption.WRITE)) {
    try (FileLock lock = ch.lock()) {
        // safe exclusive update
    }
}
```

Atomic updates and durability

To avoid corrupted or partially written files use atomic replace patterns. Write to a temp file then move into place using Files.move(temp, target, ATOMIC_MOVE) when supported. This is important for config files and indexes.

Example atomic write

```
Path temp = Files.createTempFile(dir, "tmp-", ".tmp");
Files.write(temp, dataBytes);
Files.move(temp, target, StandardCopyOption.ATOMIC_MOVE,
StandardCopyOption.REPLACE_EXISTING);
```

Serialization and structured data

For Java object persistence use JSON, XML or binary serialization. Java native serialization (ObjectOutputStream) is convenient but brittle and version sensitive. Prefer JSON with libraries such as Jackson or Gson for interoperability.

JSON example with Jackson

```
ObjectMapper mapper = new ObjectMapper();
try (OutputStream os = Files.newOutputStream(path)) {
    mapper.writeValue(os, somePojo);
}
```

File uploads and downloads in web applications

In servlets handle uploads by streaming Part objects rather than buffering whole files in memory. Annotate the servlet with @MultipartConfig or use frameworks that abstract it.

Upload servlet example

```
@MultipartConfig(fileSizeThreshold = 1024 * 1024, // buffer threshold
                 maxFileSize = 50L * 1024 * 1024,  // 50 MB
                 maxRequestSize = 60L * 1024 * 1024)
@WebServlet("/upload")
public class UploadServlet extends HttpServlet {
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        Part filePart = req.getPart("file");
        String fileName =
Paths.get(filePart.getSubmittedFileName()).getFileName().toString();
        try (InputStream in = filePart.getInputStream()) {
            Files.copy(in, Paths.get("/var/uploads").resolve(fileName),
StandardCopyOption.REPLACE_EXISTING);
        }
        resp.getWriter().println("Uploaded " + fileName);
    }
}
```

Download servlet example

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    Path file = Paths.get("/var/uploads/report.pdf");
    resp.setContentType(Files.probeContentType(file));
    resp.setHeader("Content-Disposition", "attachment; filename=\"report.pdf\"");
    try (OutputStream out = resp.getOutputStream();
         InputStream in = Files.newInputStream(file)) {
        byte[] buf = new byte[8192];
        int len;
        while ((len = in.read(buf)) > 0) out.write(buf, 0, len);
    }
}
```

When sending large files set buffer sizes appropriately and avoid response.getWriter which is for text.

Security considerations

Prevent directory traversal by validating or canonicalizing user provided paths. Never concatenate user input directly into file paths.

Path validation example

```
Path base = Paths.get("/var/uploads").toRealPath();
Path requested = base.resolve(userInput).normalize();
if (!requested.startsWith(base)) throw new SecurityException("Invalid path");
```

Check content type with Files.probeContentType and restrict uploads to allowed types. Use HttpOnly and Secure flags for cookies when you store file related tokens. Sanitize file names to avoid special characters. Limit file size to prevent denial of service. Consider virus scanning for uploaded content.

Concurrency and locking

Multiple threads may read or write files concurrently. For read only shared access is fine. For writes coordinate with locks. Use FileChannel.lock() for inter process locks, or external coordination like a database or distributed lock for cluster environments. Beware that file locks semantics differ by platform.

Files in distributed systems

When you deploy across multiple instances you should avoid storing important files only on local disk. Use shared storage such as S3 or NFS, or store metadata in a database and file contents in an object store. For session replication or temporary caching use Redis or another shared cache rather than local disk.

Logging and audit trails

When your application writes or deletes files keep an audit trail. Log the user id, timestamp, file path, and action type. This helps debugging and security forensics.

Useful libraries and utilities

Use Apache Commons IO for convenience helpers like IOUtils.copy, FileUtils.readFileToString, and FilenameUtils. Use Guava for some IO helpers if present. For CSV handling use OpenCSV or Apache Commons CSV. For streamed JSON use Jackson streaming API.

Example with Commons IO

```
try (InputStream in = Files.newInputStream(src);
     OutputStream out = Files.newOutputStream(dst)) {
    IOUtils.copy(in, out);
}
```

Non- blocking I/O in Servlets.

Non-blocking input/output (I/O) in servlets is a powerful mechanism that allows web applications to perform data transmission and processing efficiently without tying up server threads while waiting for I/O operations to complete. In the traditional model, a servlet thread blocks while reading data from a client or writing data back to it. This approach works fine for small or simple interactions but becomes inefficient when handling large uploads, slow clients, or a high number of concurrent connections. Non-blocking I/O changes this behavior by letting the servlet handle I/O asynchronously, allowing the same number of threads to serve many more requests simultaneously.

Traditional (Blocking) I/O vs Non-Blocking I/O

In the **blocking I/O model**, when a servlet reads from a request stream or writes to a response stream, the operation halts the servlet thread until data is fully read or written. If a client is slow to send data or to receive the response, the thread remains idle, wasting valuable server resources.

For example, if a servlet reads an uploaded file or sends a large report, each client consumes one thread for the entire duration of the data transfer.

In **non-blocking I/O**, instead of waiting, the servlet registers interest in I/O events. When the container detects that data is available to read or that the output stream is ready to accept more data, it notifies the servlet through callback methods. During this time, the servlet thread is free to handle other requests, improving scalability and responsiveness.

Asynchronous Processing

Before performing non-blocking I/O, the servlet must support asynchronous execution. Asynchronous processing allows a servlet to release the request thread back to the container while the I/O operation continues in the background.

To enable asynchronous support, a servlet can be declared as asynchronous either in the web deployment descriptor or by annotation:

```
@WebServlet(urlPatterns = "/asyncDemo", asyncSupported = true)
public class AsyncDemoServlet extends HttpServlet {
   // Implementation goes here
}
```

Once asynchronous mode is enabled, the servlet can start asynchronous processing using the request object:

```
AsyncContext asyncContext = request.startAsync();
```

The AsyncContext object manages the lifecycle of the asynchronous operation, allowing the servlet to complete, dispatch, or time out the request later. This mechanism forms the foundation for non-blocking I/O.

Non-Blocking Reading Using ReadListener

When the servlet reads incoming data (such as a file upload or form submission), it can use a **ReadListener** to perform non-blocking reads from the request's input stream. The container triggers specific methods of the ReadListener whenever data becomes available, has been completely read, or when an error occurs.

Example: Reading Data Without Blocking

```
@WebServlet(urlPatterns = "/readData", asyncSupported = true)
public class NonBlockingReadServlet extends HttpServlet {
   @Override
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
         throws IOException {
      AsyncContext asyncContext = request.startAsync();
      ServletInputStream input = request.getInputStream();
      input.setReadListener(new ReadListenerImpl(input, asyncContext));
   }

   private static class ReadListenerImpl implements ReadListener {
      private final ServletInputStream input;
      private final AsyncContext asyncContext;
      private final StringBuilder data = new StringBuilder();

      ReadListenerImpl(ServletInputStream input, AsyncContext asyncContext) {
         this.input = input;
         this.asyncContext = asyncContext;
      }

      @Override
      public void onDataAvailable() throws IOException {
         byte[] buffer = new byte[1024];
         int bytesRead;
         while (input.isReady() && (bytesRead = input.read(buffer)) != -1) {
            data.append(new String(buffer, 0, bytesRead));
         }
      }

      @Override
      public void onAllDataRead() throws IOException {
      HttpServletResponse response = (HttpServletResponse) asyncContext.getResponse();
```

```
            response.setContentType("text/plain");
            response.getWriter().write("Received: " + data.toString());
            asyncContext.complete();
        }

        @Override
        public void onError(Throwable throwable) {
            throwable.printStackTrace();
            asyncContext.complete();
        }
    }
}
```

Explanation

1. The servlet enables asynchronous support.
2. When a POST request is received, the servlet retrieves the ServletInputStream from the request.
3. The servlet assigns a custom ReadListener to handle input events.
4. The listener methods (onDataAvailable(), onAllDataRead(), and onError()) are automatically invoked by the container:
    - ❖ onDataAvailable() executes when part of the data is ready for reading.
    - ❖ onAllDataRead() executes once the entire body is read.
    - ❖ onError() handles I/O errors.
5. During all these operations, no servlet thread remains blocked waiting for input.

Non-Blocking Writing Using WriteListener

For sending large amounts of data (for example, streaming files or long reports), servlets can use **WriteListener** with the response's output stream. This mechanism ensures that the servlet writes data only when the client is ready to receive it.

Example: Writing Data Without Blocking

```
@WebServlet(urlPatterns = "/writeData", asyncSupported = true)
public class NonBlockingWriteServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {
        AsyncContext asyncContext = request.startAsync();
        ServletOutputStream output = response.getOutputStream();

        // Prepare large data
        List<String> messages = new ArrayList<>();
        for (int i = 1; i <= 1000; i++) {
```

```
            messages.add("Message " + i + "\n");
        }

        output.setWriteListener(newWriteListenerImpl(output,asyncContext,
messages.iterator()));
    }

    private static class WriteListenerImpl implements WriteListener {
        private final ServletOutputStream output;
        private final AsyncContext asyncContext;
        private final Iterator<String> iterator;

        WriteListenerImpl(ServletOutputStream output, AsyncContext asyncContext,
Iterator<String> iterator) {
            this.output = output;
            this.asyncContext = asyncContext;
            this.iterator = iterator;
        }

        @Override
        public void onWritePossible() throws IOException {
            while (output.isReady() && iterator.hasNext()) {
                output.write(iterator.next().getBytes());
            }
            if (!iterator.hasNext()) {
                asyncContext.complete();
            }
        }

        @Override
        public void onError(Throwable throwable) {
            throwable.printStackTrace();
            asyncContext.complete();
        }
    }
}
```

Explanation

1. The servlet uses asynchronous mode for writing.
2. The ServletOutputStream is configured with a WriteListener.
3. When the output stream becomes ready, the container calls onWritePossible().
4. The servlet writes data gradually instead of sending it all at once.
5. Once all data is sent, asyncContext.complete() is called to finalize the response.

This approach allows the servlet to handle slow clients gracefully, since it writes data only when the network can accept it, preventing buffer overflow or thread blocking.

How the Event-Driven Model Works

Non-blocking I/O in servlets follows an **event-driven model**. Instead of continuously polling streams or waiting, the container monitors the readiness of the underlying sockets and triggers appropriate callbacks when:

- Input data arrives (onDataAvailable).
- Output buffer is ready for more data (onWritePossible).
- The operation completes (onAllDataRead or complete()).

This model is similar to how asynchronous frameworks like Node.js or reactive systems handle events, but it is built into the servlet API, making it compatible with existing Java EE or Jakarta EE infrastructure.

Benefits of Non-Blocking I/O

1. Scalability

Non-blocking I/O allows a smaller number of threads to handle many connections, since threads aren't waiting on slow I/O. This greatly improves scalability in high-traffic environments.

2. Efficient Resource Utilization

By freeing up threads during data waits, the server uses CPU and memory more effectively. It can handle both active and idle clients simultaneously.

3. Responsiveness

Applications remain responsive even under heavy load or when serving clients with low bandwidth. Other requests can be processed while I/O operations are ongoing.

4. Streaming Capabilities

Non-blocking I/O is ideal for real-time applications, such as video streaming, chat systems, and continuous data feeds, where partial reads and writes are common.

Real-World Use Cases

- **File Uploads and Downloads:** Handle large file transfers without exhausting the thread pool.

- **Live Data Feeds:** Stream updates (like stock prices or IoT sensor data) to multiple clients simultaneously.
- **Web APIs with Large Responses:** Send big JSON or CSV responses efficiently to slow or mobile clients.
- **Server Push Mechanisms:** Push server updates asynchronously without blocking threads.

Practical

1. Implement a servlet for uploading files to the server and downloading them using Multipart Config and File Output Stream

Aim

Create a pair of servlets that let a user upload a file from a browser to the server and later download that file. The upload servlet uses @MultipartConfig and saves the uploaded bytes using FileOutputStream. The download servlet reads the stored file with FileInputStream and streams it to the client.

Procedure

1. Create a web application (Maven or plain dynamic web project) with servlet support.
2. Add the required servlet API dependency if using Maven. For Jakarta Servlet API: jakarta.servlet:jakarta.servlet-api with scope provided.
3. Create an upload HTML form with enctype="multipart/form-data".
4. Implement UploadServlet annotated with @MultipartConfig that receives the Part, validates and sanitizes the filename, then writes the bytes to a server folder using FileOutputStream.
5. Implement DownloadServlet that receives a filename parameter, validates it, sets proper response headers, reads the file using FileInputStream and writes to response.getOutputStream() in a buffered loop.
6. Deploy to Tomcat / Jetty / any servlet container and test using browser and curl.

Program

Maven snippet (optional)

If you use Maven, add the servlet API dependency (scope provided):

```
<dependency>
  <groupId>jakarta.servlet</groupId>
```

```
  <artifactId>jakarta.servlet-api</artifactId>
  <version>5.0.0</version>
  <scope>provided</scope>
</dependency>
```

HTML Form (upload.html)

Place this in your webapp root or WebContent folder.

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>Upload File</title>
</head>
<body>
  <h2>Upload a File</h2>
  <form method="post" action="upload" enctype="multipart/form-data">
    Select file: <input type="file" name="file" required />
    <br/><br/>
    <input type="submit" value="Upload" />
  </form>

  <h3>Download Example</h3>
  <!-- replace sample-file.txt with actual uploaded filename -->
  <a href="download?file=sample-file.txt">Download sample-file.txt</a>
</body>
</html>
```

Upload Servlet (UploadServlet.java)

This servlet saves the uploaded file to a server directory using FileOutputStream. It also returns a simple HTML response showing success and a download link.

```
package com.example.filedemo;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.MultipartConfig;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.Part;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.*;
```

```
@WebServlet("/upload")
@MultipartConfig(
    fileSizeThreshold = 1024 * 1024,  // 1MB before writing to disk
    maxFileSize = 50L * 1024 * 1024,  // 50MB per file
    maxRequestSize = 60L * 1024 * 1024 // 60MB total
)
public class UploadServlet extends HttpServlet {

    // Directory where uploaded files will be saved, outside webapp for safety
    private static final String UPLOAD_DIR = "/tmp/uploads"; // change as needed

    @Override
    public void init() throws ServletException {
        super.init();
        // Make sure upload directory exists
        File uploadDir = new File(UPLOAD_DIR);
        if (!uploadDir.exists()) {
            uploadDir.mkdirs();
        }
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");

        Part filePart = request.getPart("file");
        if (filePart == null || filePart.getSize() == 0) {
            response.getWriter().println("<p>No file selected</p>");
            return;
        }

        // Extract submitted filename in a safe manner
        String submittedFileName = getFileName(filePart);
        if (submittedFileName == null || submittedFileName.isBlank()) {
            response.getWriter().println("<p>Invalid file name</p>");
            return;
        }

        // Sanitize filename to avoid directory traversal and strange characters
        String safeFileName = submittedFileName.replaceAll("[\\\\/\\s]+", "_");

        File savedFile = new File(UPLOAD_DIR, safeFileName);
```

```
        // Write uploaded file to disk using FileOutputStream
        try (InputStream input = filePart.getInputStream();
             FileOutputStream fos = new FileOutputStream(savedFile);
             BufferedOutputStream bos = new BufferedOutputStream(fos)) {

            byte[] buffer = new byte[8192];
            int bytesRead;
            while ((bytesRead = input.read(buffer)) != -1) {
                bos.write(buffer, 0, bytesRead);
            }
            bos.flush();
        } catch (IOException e) {
            response.getWriter().println("<p>Error saving file: " + e.getMessage() + "</p>");
            return;
        }

        // Successful response with link to download
        String downloadLink = request.getContextPath() + "/download?file=" +
java.net.URLEncoder.encode(safeFileName, "UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<html><body>");
            out.println("<h3>File uploaded successfully</h3>");
            out.println("<p>Saved as: " + safeFileName + "</p>");
            out.println("<p><a href=\"" + downloadLink + "\">Download " + safeFileName +
"</a></p>");
            out.println("</body></html>");
        }
    }

    // Utility to parse file name from Part header
    private String getFileName(Part part) {
        String cd = part.getHeader("content-disposition");
        if (cd == null) return null;
        for (String token : cd.split(";")) {
            token = token.trim();
            if (token.startsWith("filename")) {
                String fileName = token.substring(token.indexOf('=') + 1).trim().replace("\"", "");
                // Some browsers include full path, so get only the final name
                return fileName.substring(Math.max(fileName.lastIndexOf('/'),
fileName.lastIndexOf('\\')) + 1);
            }
        }
        return null;
    }
}
```

Download Servlet (DownloadServlet.java)

This servlet streams a requested file back to the client. It uses FileInputStream and writes to response.getOutputStream() with proper headers. It validates the filename and enforces that files are served only from the upload directory.

```
package com.example.filedemo;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.*;

@WebServlet("/download")
public class DownloadServlet extends HttpServlet {

    private static final String UPLOAD_DIR = "/tmp/uploads"; // must match UploadServlet

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String requested = request.getParameter("file");
        if (requested == null || requested.isBlank()) {
            response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Missing file parameter");
            return;
        }

        // Decode and sanitize
        String fileName = new File(requested).getName(); // this strips any path info
        File file = new File(UPLOAD_DIR, fileName);

        // Validate file exists and is a file
        if (!file.exists() || !file.isFile()) {
            response.sendError(HttpServletResponse.SC_NOT_FOUND, "File not found");
            return;
        }

        // Set content type and headers
        String mime = getServletContext().getMimeType(file.getName());
        if (mime == null) mime = "application/octet-stream";
```

```
        response.setContentType(mime);
        response.setContentLengthLong(file.length());
        // Force download dialog
        response.setHeader("Content-Disposition", "attachment; filename=\"" + file.getName()
+ "\"");

        // Stream file content via FileInputStream
        try (FileInputStream fis = new FileInputStream(file);
             BufferedInputStream bis = new BufferedInputStream(fis);
             OutputStream os = response.getOutputStream();
             BufferedOutputStream bos = new BufferedOutputStream(os)) {

            byte[] buffer = new byte[8192];
            int bytesRead;
            while ((bytesRead = bis.read(buffer)) != -1) {
                bos.write(buffer, 0, bytesRead);
            }
            bos.flush();
        } catch (IOException e) {
            // If client aborts connection, writing will fail; just log or ignore
            log("Error streaming file " + file.getAbsolutePath(), e);
        }
    }
}
```

Example Output

1. Browser after upload (UploadServlet response)

File uploaded successfully
Saved as: localfile.txt
[Download localfile.txt]

2. Browser when downloading (download dialog or saved file)

The browser prompts to save or directly opens the file depending on MIME type and settings. The downloaded file matches the uploaded bytes exactly.

3. curl output for download

$ curl -v -o downloaded.txt "http://localhost:8080/app/download?file=localfile.txt"
<... HTTP/1.1 200 OK ...>

downloaded.txt equals the original.

Result

The servlet program for file upload and download using @MultipartConfig and FileOutputStream was successfully implemented, executed, and tested.

2. Develop a servlet to handle HTTP GET and POST methods. Accept form data (e.g., user registration) and display it.

Aim

To develop a servlet that handles HTTP GET and POST methods. The servlet will present a user registration form on GET and accept the submitted form data on POST, then display the submitted user details back to the client.

Procedure

1. Create a dynamic web project or a Maven webapp.
2. Add servlet API dependency if using Maven.
3. Create an HTML registration form that sends data with method POST. Optionally let the servlet render the form on GET.
4. Implement a servlet that overrides doGet() to show the form and doPost() to process the form data.
5. Validate and sanitize user inputs.
6. Display the registration details in an HTML response.
7. Deploy to a servlet container such as Tomcat and test using a browser and curl.

HTML Form (registration.html)

Place this file in the webapp root (or have doGet generate the same form).

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>User Registration</title>
</head>
<body>
  <h2>User Registration Form</h2>
  <form action="register" method="post">
    Full Name: <input type="text" name="fullname" required /><br/><br/>
    Email: <input type="email" name="email" required /><br/><br/>
    Password: <input type="password" name="password" required /><br/><br/>
    Gender:
      <input type="radio" name="gender" value="Male" checked/> Male
```

```
    <input type="radio" name="gender" value="Female" /> Female
    <input type="radio" name="gender" value="Other" /> Other
  <br/><br/>
  Hobbies:
    <input type="checkbox" name="hobby" value="Reading" /> Reading
    <input type="checkbox" name="hobby" value="Traveling" /> Traveling
    <input type="checkbox" name="hobby" value="Gaming" /> Gaming
  <br/><br/>
  Country:
    <select name="country">
      <option value="India">India</option>
      <option value="USA">USA</option>
      <option value="UK">UK</option>
      <option value="Other">Other</option>
    </select>
  <br/><br/>
  <input type="submit" value="Register" />
 </form>
</body>
</html>
]
```

Servlet Program (RegisterServlet.java)

This servlet shows the form on GET and processes it on POST. It demonstrates basic validation and displays the submitted data (except password is masked when displayed for safety).

```
package com.example.registration;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.URLEncoder;
import java.util.Arrays;

@WebServlet(name = "RegisterServlet", urlPatterns = {"/register"})
public class RegisterServlet extends HttpServlet {
```

```
    // Show the registration form (GET)
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<!doctype html><html><head><meta charset='utf-8'/><title>User
Registration</title></head><body>");
            out.println("<h2>User Registration Form</h2>");
            out.println("<form method='post' action='register'>");
            out.println("Full Name: <input type='text' name='fullname' required /><br/><br/>");
            out.println("Email: <input type='email' name='email' required /><br/><br/>");
            out.println("Password: <input type='password' name='password' required
/><br/><br/>");
            out.println("Gender: <input type='radio' name='gender' value='Male' checked/>
Male ");
            out.println("<input type='radio' name='gender' value='Female'/> Female ");
            out.println("<input type='radio' name='gender' value='Other'/> Other<br/><br/>");
            out.println("Hobbies: <input type='checkbox' name='hobby' value='Reading'/>
Reading ");
            out.println("<input type='checkbox' name='hobby' value='Traveling'/> Traveling ");
            out.println("<input type='checkbox' name='hobby' value='Gaming'/>
Gaming<br/><br/>");
            out.println("Country: <select name='country'>");

out.println("<option>India</option><option>USA</option><option>UK</option><option
>Other</option>");
            out.println("</select><br/><br/>");
            out.println("<input type='submit' value='Register'/>");
            out.println("</form>");
            out.println("</body></html>");
        }
    }

    // Process registration data (POST)
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        // Ensure correct character encoding for form parameters
        request.setCharacterEncoding("UTF-8");
        response.setContentType("text/html;charset=UTF-8");
        // Retrieve parameters
        String fullName = safe(request.getParameter("fullname"));
        String email = safe(request.getParameter("email"));
        String password = request.getParameter("password"); // do not echo raw password
```

```
        String gender = safe(request.getParameter("gender"));
        String[] hobbies = request.getParameterValues("hobby"); // may be null
        String country = safe(request.getParameter("country"));

        // Simple server side validation
        StringBuilder errors = new StringBuilder();
        if (fullName == null || fullName.isBlank()) errors.append("Full name is
required.<br/>");
        if (email == null || email.isBlank()) errors.append("Email is required.<br/>");
        if (password == null || password.length() < 6) errors.append("Password is required and
must be at least 6 characters.<br/>");

        try (PrintWriter out = response.getWriter()) {
            out.println("<!doctype html><html><head><meta charset='utf-
8'/><title>Registration Result</title></head><body>");
            out.println("<h2>Registration Result</h2>");

            if (errors.length() > 0) {
                // Show errors and a link back
                out.println("<div style='color:red'><strong>Errors:</strong><br/>" +
errors.toString() + "</div>");
                out.println("<p><a href='register'>Go back to form</a></p>");
            } else {
                // Success: display user details (mask password)
                out.println("<p><strong>Full Name:</strong> " + escapeHtml(fullName) +
"</p>");
                out.println("<p><strong>Email:</strong> " + escapeHtml(email) + "</p>");
                out.println("<p><strong>Password:</strong> " + maskPassword(password) +
"</p>");
                out.println("<p><strong>Gender:</strong> " + escapeHtml(gender) + "</p>");
                out.println("<p><strong>Hobbies:</strong> " + (hobbies == null ? "None" :
escapeHtml(Arrays.toString(hobbies))) + "</p>");
                out.println("<p><strong>Country:</strong> " + escapeHtml(country) + "</p>");

                // Optionally provide a link for a fresh registration
                out.println("<p><a href='register'>Register another user</a></p>");
            }
            out.println("</body></html>");
        }
    }

    // Basic null safe method
    private String safe(String s) {
        return s == null ? "" : s.trim();
    }
```

```
    // Mask password when displaying
    private String maskPassword(String pwd) {
        if (pwd == null) return "";
        int len = Math.min(pwd.length(), 8);
        return "*".repeat(len) + (pwd.length() > len ? "..." : "");
    }

    // Simple HTML escape to avoid XSS in responses
    private String escapeHtml(String s) {
        if (s == null) return "";
        return s.replace("&", "&amp;")
                .replace("<", "&lt;")
                .replace(">", "&gt;")
                .replace("\"", "&quot;")
                .replace("'", "&#39;");
    }
}
```

1. The servlet is mapped to /register by the @WebServlet annotation.
2. doGet emits the registration form; doPost handles the submitted data.
3. request.setCharacterEncoding("UTF-8") ensures correct reading of Unicode form input.
4. The program masks the password when displayed and escapes HTML to reduce XSS risk.

If you prefer web.xml mapping instead of annotation, use:

```
<servlet>
  <servlet-name>RegisterServlet</servlet-name>
  <servlet-class>com.example.registration.RegisterServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>RegisterServlet</servlet-name>
  <url-pattern>/register</url-pattern>
</servlet-mapping>
```

How to test

Using browser

1. Deploy the application to Tomcat at http://localhost:8080/yourapp/.
2. Open http://localhost:8080/yourapp/register to see the form rendered by doGet.
3. Fill in the form and click Register. The doPost response page will display the submitted details.

Using curl

Submit form data via curl:

```
curl -X POST \
  -d "fullname=Greenfield User" \
  -d "email=green@field.com" \
  -d "password=secret123" \
  -d "gender=Male" \
  -d "hobby=Reading" \
  -d "hobby=Gaming" \
  -d "country=India" \
  http://localhost:8080/yourapp/register
```

The server will return HTML containing the registration result.

Sample Outputs

1. GET request to /register (form page)

Browser will show the registration form with fields Full Name, Email, Password, Gender, Hobbies and Country and a Register button.

2. POST request with valid data

Page displays:

Registration Result

Full Name: Greenfield User
Email: green@field.com
Password: ********
Gender: Male
Hobbies: [Reading, Gaming]
Country: India

[Register another user]

3. POST request with invalid data (e.g., short password)

Page displays:

Registration Result

Errors:
Password is required and must be at least 6 characters.

[Go back to form]

Result

The servlet that handles HTTP GET and POST methods for user registration was implemented successfully.

CHAPTER – 4

JAVA SERVER PAGES AND JSTL

4.1 JSP Architecture and Lifecycle

JavaServer Pages (JSP) architecture follows a **3-tier architecture** that separates the web application into three distinct layers **Presentation Layer**, **Logic Layer**, and **Data Layer**. This separation of concerns ensures better **maintainability**, **reusability**, and **scalability** of the application. JSP allows developers to create dynamic and interactive web pages by embedding Java code directly within HTML pages. It acts as an interface between the client and the server, simplifying web development on the Java EE platform.

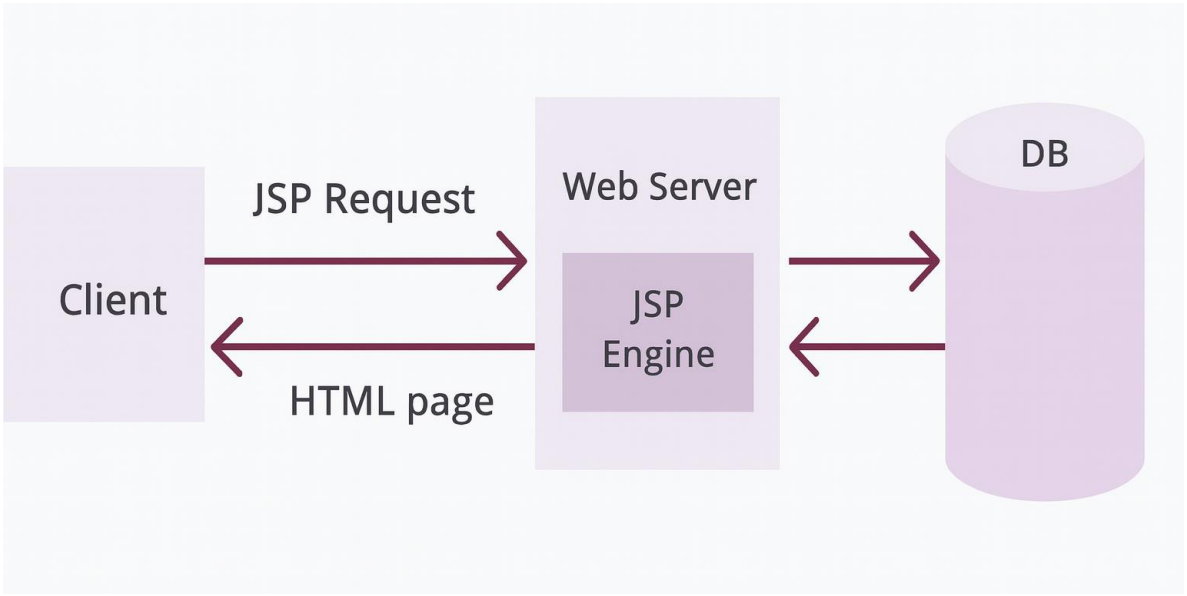


Fig. 4.1 JSP Architecture.

Three-Tier Structure of JSP Architecture

1. Presentation Layer (Client Side)

The presentation layer is the **front-end interface** that interacts with the user. It is responsible for **displaying content** and **collecting user input**. In a JSP-based web application, this layer consists of **JSP pages** that combine HTML and JSP tags to produce dynamic web content. When a user makes a request (such as submitting a form or clicking a link), it is sent to the

server for processing. The presentation layer ensures that users have a clear and interactive interface to communicate with the application.

2. Logic Layer (Server Side)

The logic layer, also known as the **business logic layer**, handles **request processing**, **decision-making**, and **data manipulation**. It is implemented using **Servlets** and **JavaBeans**. When a JSP page is requested, it is first **converted into a servlet** by the **JSP engine**, and the servlet handles all request and response processing. JavaBeans are used here to encapsulate business logic, making the application more modular and reusable. This layer ensures that all the business rules and operations are executed correctly before sending data back to the presentation layer.

3. Data Layer (Database Layer)

The data layer is responsible for **data storage, retrieval, and management**. It interacts with databases or external data sources to perform **CRUD operations** (Create, Read, Update, Delete). This layer is typically accessed through JavaBeans or JDBC (Java Database Connectivity). By separating the data layer, JSP architecture ensures that database changes do not affect the presentation or logic layers directly, improving flexibility and maintainability.

Main Components of JSP Architecture

1. JSP Pages

JSP pages are the **core components** of a JSP application. They contain a mix of **HTML**, **Java code**, and **JSP tags** (like scriptlets, expressions, and directives) that generate dynamic web content. JSP pages are easier to design and maintain since they separate the static layout from the dynamic logic.

2. Servlets

Behind the scenes, every JSP page is **compiled into a servlet** by the JSP engine. Servlets are Java classes that handle **HTTP requests and responses**. They execute the logic embedded in JSP pages and generate the dynamic content that will be sent to the client browser. This conversion allows JSP to take advantage of the Java platform's robustness and security.

3. JSP Engine (Web Container)

The **JSP engine**, also known as the **web container**, is responsible for managing and executing JSP pages. It handles tasks such as **compiling JSP pages into servlets**, **loading them into the JVM**, and **executing them** when requested. Common JSP engines include Apache Tomcat, GlassFish, and Jetty. The web container also manages the lifecycle of servlets, ensuring efficient memory and resource utilization.

4. JavaBeans

JavaBeans are reusable Java components used to **encapsulate business logic and data**. They allow developers to store and process data independently from JSP pages, promoting a clean MVC (Model-View-Controller) structure. For example, a JavaBean might handle customer data retrieval or validation logic.

5. JSTL (JavaServer Pages Standard Tag Library)

The **JSTL** provides a set of **standard tags** for common tasks like iteration, conditional processing, and internationalization. It reduces the need for embedding Java code directly in JSP pages, making them cleaner and easier to maintain.

6. Custom Tag Libraries

JSP also allows developers to define **custom tags**, which encapsulate complex logic into reusable components. These custom tag libraries can be imported and used across multiple JSP pages, promoting modular development and code reuse.

JSP Architecture Flow

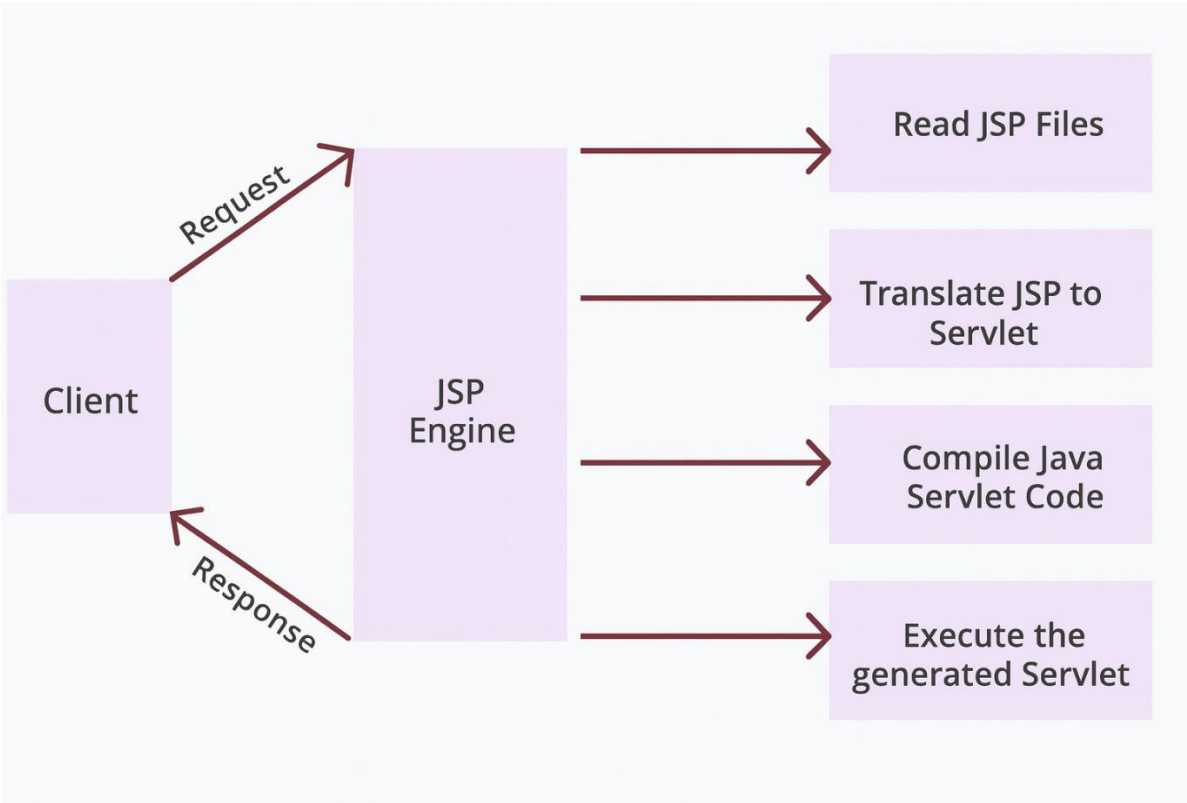


Fig. 4.2 JSP architecture flow.

The **JSP architecture flow** describes how a JSP page is processed from the moment a client sends a request until the response is received. The steps are as follows:

1. **Client Request:** A client, such as a web browser, sends an HTTP request to the web server for a specific JSP page.
2. **Forwarding to JSP Engine:** The web server forwards this request to the **JSP engine**, which is responsible for handling JSP processing.
3. **Translation and Compilation:** The JSP engine checks whether the JSP page has been previously compiled into a servlet.
    - If not, it **translates** the JSP file into a **Java servlet source file**.
    - The servlet is then **compiled** into a `.class` file and loaded into the JVM.
4. **Execution of Servlet:** The servlet's **service()** method is executed. This method generates the **dynamic content** (HTML or XML) by combining static template data with dynamic data generated by Java code.
5. **Response Generation:** The JSP engine sends the generated HTML response back to the **web server**, which then forwards it to the **client's browser**.
6. **Caching for Efficiency:** Once a JSP page is compiled into a servlet, it is **cached** by the JSP engine. Subsequent requests to the same JSP page are served faster without recompilation unless the JSP file changes.

Advantages of JSP Architecture

- **Separation of Concerns:** Clearly separates the presentation, logic, and data layers.
- **Reusability:** Components like JavaBeans and custom tags can be reused across multiple pages.
- **Ease of Maintenance:** Developers can modify one layer without affecting others.
- **Platform Independence:** Runs on any platform supporting Java.
- **Performance Optimization:** Once compiled, JSP pages behave like servlets, providing efficient performance.

JSP Life Cycle

The **JavaServer Pages (JSP) Life Cycle** refers to the complete process that a JSP page undergoes from the moment it is requested by a client until it is destroyed by the web container. This life cycle is managed automatically by the **web container or JSP engine** which is part of the application server, such as Apache Tomcat or GlassFish.

The JSP life cycle is very similar to the servlet life cycle because every JSP page is internally converted into a servlet before execution. However, JSP has an additional phase at the beginning which involves translation and compilation. Understanding this life cycle helps developers know when to initialize resources, handle requests, and release resources properly. It also ensures that JSP pages are optimized for performance and maintainability.

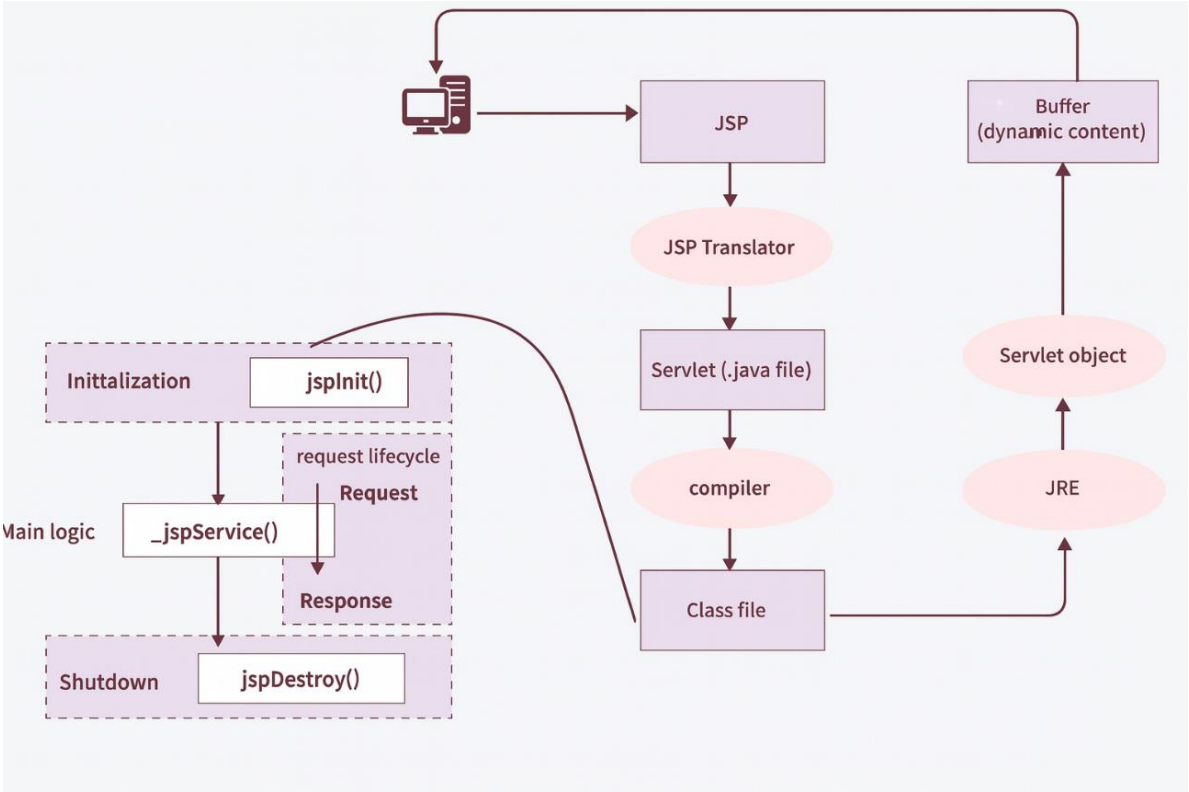


Fig. 4.3 Lifecycle of JSP.

Phase of JSP

The JSP life cycle consists of **seven important phases** that define how a JSP page is created, executed, and destroyed by the container. Each phase performs a specific function that contributes to the smooth processing of web applications.

1. Translation Phase (JSP to Servlet Conversion)

When a client sends a request for a JSP page for the very first time, the web container checks if the page has already been converted into a servlet. If not, the JSP engine begins the **translation phase**. In this phase, the JSP file is parsed and converted into an equivalent **Java servlet source file**.

This process transforms all JSP tags and elements such as directives, expressions, declarations, and scriptlets into corresponding servlet code. For example, HTML code remains as static content while the Java code written inside the JSP tags is converted into Java statements within the servlet class.

Suppose the JSP file name is test.jsp. During this phase, the JSP engine generates a Java source file like test.java, which contains servlet code as follows:

```
public class TestServlet extends HttpServlet {
   // Generated servlet code
}
```

The main purpose of this phase is to ensure that the JSP page can be executed as a Java servlet by the server. Any syntax errors or tag mismatches are also detected and reported during this step.

2. Compilation Phase

After the JSP file has been successfully translated into servlet source code, the next step is **compilation**. In this phase, the generated servlet file (for example, test.java) is compiled by the Java compiler into **bytecode** in the form of a .class file (for example, test.class).

This bytecode file contains platform-independent instructions that can be executed by the **Java Virtual Machine (JVM)**. Once compiled, the JSP page now behaves exactly like a servlet and can be loaded and executed by the container.

This process is automatic and happens only once unless the JSP file is modified later. If any changes are made to the JSP file, the translation and compilation phases are repeated.

3. Class Loading Phase

After the compilation, the **web container loads the generated servlet class into memory**. This is done using the class loader of the container. The class loader is responsible for finding the servlet class file and bringing it into the JVM for execution.

By loading the class into memory, the container ensures that it is available to process client requests. The loaded servlet class now contains both the static content (HTML or XML) and dynamic Java code that will generate the final output to be displayed in the web browser.

4. Instantiation Phase

Once the class has been loaded, the web container creates an **instance of the generated servlet class**. This means that an object of the servlet is created in memory.

This object is responsible for handling all the requests made to that particular JSP page. The container typically maintains one instance per JSP page to handle multiple client requests efficiently. Creating this instance marks the point where the JSP page officially becomes an active component of the web application and can begin interacting with users.

For example:

```
TestServlet servlet = new TestServlet();
```

The container automatically handles this process, and developers do not need to manually create the instance.

5. Initialization Phase (jspInit Method)

After creating the servlet instance, the web container calls the **jspInit()** method. This method is executed only once during the entire life cycle of the JSP page, when the page is first loaded into memory.

The purpose of the jspInit() method is to perform **initial setup tasks** that the JSP might need during its execution. These tasks can include establishing database connections, loading configuration parameters, setting up logging systems, or initializing shared resources.

For example:

```
public void jspInit() {
   System.out.println("JSP Initialized");
   // Initialization tasks like opening a database connection
}
```

Since jspInit() is called only once, it is similar to the init() method in servlets. It ensures that all required resources are ready before handling client requests.

6. Request Processing Phase (_jspService Method)

After initialization, the JSP page is ready to handle client requests. Every time a client (such as a web browser) sends a request to the JSP page, the container calls the **_jspService()** method of the servlet instance.

This method is the core part of the JSP life cycle. It receives two objects as parameters:

- HttpServletRequest which contains the information sent by the client, and
- HttpServletResponse which is used to send data back to the client.

The _jspService() method generates **dynamic web content** by combining the static HTML in the JSP with the dynamic Java code execution results. The output is then sent to the web browser as a complete HTML page.

Example:

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
      throws IOException {
   response.getWriter().write("<html><body>Hello, World!</body></html>");
}
```

It is important to note that developers cannot override the _jspService() method because it is automatically generated and declared as final by the JSP engine. This ensures consistency in how client requests are handled.

7. JSP Cleanup Phase (jspDestroy Method)

The final phase in the JSP life cycle is the **cleanup or destruction phase**. When the web container decides that the JSP page is no longer needed (for example, during server shutdown or when the application is undeployed), it calls the **jspDestroy()** method.

This method is called only once before the JSP instance is removed from memory. It is used to **release resources** that were allocated during initialization or request processing. For example, you can close database connections, terminate background threads, or release file handles.

Example:

```
public void jspDestroy() {
   System.out.println("JSP Destroyed");
   // Cleanup tasks such as closing database connections
}
```

Proper use of this method ensures that resources are not wasted and that the application runs efficiently even under heavy load.

Important Characteristics of JSP Life Cycle

1. The JSP life cycle is **fully managed by the web container**, meaning that translation, compilation, loading, and destruction are automatic.
2. The jspInit() and jspDestroy() methods **can be overridden** by developers to perform custom initialization and cleanup tasks.
3. The _jspService() method **cannot be overridden** because it is generated automatically by the container.
4. The **translation and compilation** phases occur only the first time a JSP page is requested or when the JSP file is modified.
5. Once compiled, the JSP behaves like a **servlet** and follows the servlet execution model.

Action Elements

In **JavaServer Pages (JSP)**, **action elements** are special **XML-based tags** that allow developers to perform dynamic operations such as including resources, forwarding requests, managing JavaBeans, and creating reusable and interactive web components. These tags are

predefined in the JSP specification and are designed to perform specific tasks during the **request processing phase** of a JSP page.

JSP action elements are written in the form of **XML tags** that begin with <jsp:...> and follow XML syntax rules. This makes them portable, readable, and easy to integrate with XML-based tools and frameworks. Unlike scriptlets (which mix Java code with HTML), action elements help developers write **cleaner, modular, and maintainable code** by separating business logic from presentation logic.

Purpose and Importance of JSP Action Elements

The main purpose of using action elements is to allow the **JSP container** to perform predefined operations such as including pages, managing objects, and transferring control between resources dynamically, **without writing explicit Java code**. They enhance the functionality of JSP pages by:

- Promoting code reusability
- Reducing the amount of embedded Java code (scriptlets)
- Improving readability and maintainability
- Supporting XML-compliant syntax

By using action elements, JSP pages become more modular and easier to update or extend, which is especially beneficial for large web applications.

Types of JSP Action Elements

The JSP specification defines several action elements, each designed to perform a particular task in the JSP execution process. The following are the most commonly used action elements.

1. <jsp:include> – Including Another Resource

The <jsp:include> action element is used to **dynamically include the content of another resource** such as a JSP file, HTML file, or servlet into the current page during **runtime**. This means the inclusion happens when the page is requested, not during translation or compilation.

Syntax:

<jsp:include page="relativeURL" flush="true|false" />

- The page attribute specifies the resource to be included.
- The flush attribute, when set to true, ensures that the current output buffer is sent to the client before the inclusion occurs.

Example:

<jsp:include page="header.jsp" flush="true" />

This action dynamically inserts the content of header.jsp into the current page. Using <jsp:include> promotes code reuse since common components like headers, footers, or navigation bars can be maintained separately.

2. <jsp:forward> – Forwarding the Request

The <jsp:forward> action element is used to **forward the client request and response** to another resource, such as another JSP, servlet, or HTML file. After the forwarding, control is completely transferred to the new resource, and no further output from the current page is sent to the client.

Syntax:

<jsp:forward page="relativeURL" />

Example:

<jsp:forward page="welcome.jsp" />

In this example, when the current page executes, the request is forwarded to welcome.jsp. This is commonly used for redirecting users based on certain conditions, such as successful login or form submission.

3. <jsp:useBean> – Declaring and Using JavaBeans

The <jsp:useBean> element is one of the most important action elements in JSP. It is used to **instantiate or locate a JavaBean object** that can be used within the page to handle data and business logic.

Syntax:

<jsp:useBean id="beanName" class="package.ClassName"
scope="page|request|session|application" />

- id specifies the name of the bean instance.
- class specifies the full class name of the JavaBean.
- scope defines where the bean is stored (page, request, session, or application).

Example:

<jsp:useBean id="user" class="com.example.UserBean" scope="session" />

If the bean instance already exists in the specified scope, it is reused; otherwise, a new instance is created. This allows JSP pages to interact with backend Java classes in a simple, structured way.

4. <jsp:setProperty> – Setting Bean Properties

The <jsp:setProperty> element is used to **set the value of a property** in a JavaBean object. It can either assign a literal value or retrieve a value from a request parameter.

Syntax:

<jsp:setProperty name="beanName" property="propertyName" value="value" />

Example:

<jsp:setProperty name="user" property="username" value="John" />

You can also set a property using a request parameter:

<jsp:setProperty name="user" property="username" param="uname" />

This element provides a simple and declarative way to pass data from the user interface (form inputs) to the business logic represented by JavaBeans.

5. <jsp:getProperty> – Retrieving Bean Properties

The <jsp:getProperty> action element is used to **retrieve the value of a JavaBean property** and insert it directly into the response output (HTML).

Syntax:

<jsp:getProperty name="beanName" property="propertyName" />

Example:

<jsp:getProperty name="user" property="username" />

If the username property in the user bean has the value "John", this tag will output John on the web page. This allows dynamic display of data stored in JavaBeans without writing explicit Java code.

6. <jsp:param> – Passing Parameters

The <jsp:param> element is used inside <jsp:include> or <jsp:forward> tags to **pass additional parameters** to the target resource.

Syntax:

```
<jsp:param name="paramName" value="paramValue" />
```

Example:

```
<jsp:forward page="welcome.jsp">
   <jsp:param name="user" value="John" />
</jsp:forward>
```

Here, the parameter user with value John will be passed to welcome.jsp, allowing the target page to access the value using request.getParameter("user").

7. <jsp:plugin> – Embedding Java Applets or Beans

The <jsp:plugin> action element is used to **embed Java applets or JavaBeans into a JSP page** by generating browser-specific HTML tags such as <object> or <embed>.

Syntax:

```
<jsp:plugin type="applet|bean" code="classFileName" codebase="URL">
   <jsp:params />
   <jsp:fallback>Alternative content</jsp:fallback>
</jsp:plugin>
```

This tag ensures that the applet or bean runs correctly in browsers that support Java. The <jsp:fallback> tag provides alternate content if the plugin fails to load.

8. <jsp:fallback> – Alternative Content for Plugins

The <jsp:fallback> tag is used **inside a <jsp:plugin> element** to define the content that should be displayed if the plugin or applet fails to load on the client browser.

Example:

```
<jsp:plugin type="applet" code="HelloApplet.class" codebase="/applets">
   <jsp:fallback>
      Your browser does not support Java Applets.
   </jsp:fallback>
</jsp:plugin>
```

This ensures that users still see a message or alternate content even if the applet cannot be displayed.

9. <jsp:element> – Creating Dynamic XML Elements

The <jsp:element> action allows developers to **dynamically create XML elements** at runtime. It is often used with <jsp:attribute> and <jsp:body> to define element attributes and body content dynamically.

Syntax:

```
<jsp:element name="elementName">
   <jsp:attribute name="attrName">value</jsp:attribute>
   <jsp:body>content</jsp:body>
</jsp:element>
```

This tag provides flexibility when generating XML or XHTML documents dynamically from JSP pages.

10. <jsp:body> and <jsp:attribute> – Defining Content Dynamically

The <jsp:body> tag defines the **body content** of a dynamically created XML element, while <jsp:attribute> defines the **attributes** for that element. These tags are generally used within <jsp:element> to create dynamic structures.

Example:

```
<jsp:element name="message">
   <jsp:attribute name="type">info</jsp:attribute>
   <jsp:body>Hello, this is a dynamic message!</jsp:body>
</jsp:element>
```

This example dynamically generates:

```
<message type="info">Hello, this is a dynamic message!</message>
```

Advantages of Using JSP Action Elements

1. **Cleaner Code:** Reduces Java code within JSP pages by providing XML-based tags.
2. **Reusability:** Common components like headers or beans can be reused across multiple pages.
3. **Maintainability:** Easier to update and debug because presentation and logic are separated.
4. **XML Compatibility:** Follows XML syntax, making it compatible with XML parsers and tools.
5. **Dynamic Resource Management:** Allows including and forwarding resources dynamically at runtime.

4.2 Implicit Objects and Scripting Elements

Implicit Objects

In Java Server Pages, implicit objects are special predefined objects that are automatically created by the JSP container for every page request. These objects are called implicit because they are available by default to the developer without the need to declare or instantiate them manually. They can be directly accessed within scripting elements such as expressions, declarations, and scriptlets.

Implicit objects simplify the interaction between the server and the client by providing easy access to important request and response information, session data, application configuration, and more. There are a total of nine implicit objects available in JSP, and each one serves a specific purpose during the execution of a JSP page.

List of JSP Implicit Objects

S.No	Implicit Object	Class Type	Description
1	request	HttpServletRequest	Represents the client request and provides access to form data and request information.
2	response	HttpServletResponse	Represents the response sent to the client and allows modification of headers or redirection.
3	out	JspWriter	Used to send output to the client browser.
4	session	HttpSession	Used to store and manage user specific data during a session.
5	application	ServletContext	Provides application wide data and resources accessible by all JSP pages.
6	config	ServletConfig	Provides configuration information for the current JSP page.
7	pageContext	PageContext	Provides access to all JSP objects and manages different scopes.
8	page	Object	Refers to the current JSP page similar to the keyword this in Java.
9	exception	Throwable	Represents exceptions and errors thrown during JSP execution.

Table. 4.1 JSP Implicit Objects.

The Request Object

The request object is an instance of the javax.servlet.http.HttpServletRequest class. It represents the client request made to the server. Every time a user sends a request such as submitting a form or clicking a link, the web container automatically creates a new request object.

This object allows the developer to access form data, request parameters, header information, and cookies. It also provides details about the client and the server such as IP address, port, and protocol.

Common Methods of request object:

- getParameter(String name) – Returns the value of a form field
- getHeader(String name) – Returns the value of a specific header
- getCookies() – Returns an array of cookies sent by the client
- getMethod() – Returns the request method such as GET or POST

Example:

```
<html>
<body>
<form action="welcome.jsp" method="post">
Enter Name: <input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

welcome.jsp

```
Hello, <%= request.getParameter("username") %>
```

When the user submits the form, the entered value is retrieved using the request object.

The Response Object

The response object is an instance of javax.servlet.http.HttpServletResponse. It represents the response that the server sends back to the client. This object allows developers to modify HTTP headers, redirect users to other pages, and add cookies to the response.

Common Methods of response object:

- sendRedirect(String url) – Redirects the client to another page
- addCookie(Cookie cookie) – Adds a new cookie to the response

- setContentType(String type) – Sets the MIME type of the response
- setStatus(int code) – Sets the HTTP status code

Example:

```
<%
String name = request.getParameter("username");
if(name == null || name.equals("")){
   response.sendRedirect("error.jsp");
}else{
   out.println("Welcome " + name);
}
%>
```

If the name is not entered, the user is redirected to the error page.

The Out Object

The out object is an instance of javax.servlet.jsp.JspWriter. It is used to send output or content to the client browser. It works similar to System.out.println() in Java but is designed specifically for web output.

Common Methods of out object:

- print(dataType dt) – Prints data to the client
- println(dataType dt) – Prints data and moves to the next line
- flush() – Flushes the output buffer

Example:

```
<%
out.println("Hello from JSP!");
out.print("This page is served at: " + new java.util.Date());
%>
```

This example displays text and the current date on the web page.

The Session Object

The session object is an instance of javax.servlet.http.HttpSession. It is used to store and manage user specific data across multiple requests during a single session. This means data stored in the session remains available until the session is terminated or expires.

Common Methods of session object:

- setAttribute(String name, Object value) – Stores data in the session
- getAttribute(String name) – Retrieves data from the session
- invalidate() – Destroys the session

Example:

```
<%
session.setAttribute("user", "John");
out.print("Welcome, " + session.getAttribute("user"));
%>
```

This code stores the user name in the session and displays it on the page.

The Application Object

The application object is an instance of javax.servlet.ServletContext. It represents the overall web application and provides a way to share information among all JSP pages and servlets in the same application.

This object is often used for storing global variables or initialization parameters that are accessible across the entire application.

Example:

```
<%
Integer count = (Integer)application.getAttribute("visitorCount");
if(count == null){
   count = 1;
}else{
   count++;
}
application.setAttribute("visitorCount", count);
out.print("Total Visitors: " + count);
%>
```

This example maintains a global visitor counter shared by all users of the application.

The Config Object

The config object is an instance of javax.servlet.ServletConfig. It provides configuration and initialization information specific to the current JSP page.

Common Method of config object:

- getServletName() – Returns the name of the servlet
- getInitParameter(String name) – Returns initialization parameter values

Example:

```
<%
out.print("Servlet Name: " + config.getServletName());
%>
```

This prints the name of the servlet as defined in the deployment descriptor file.

The PageContext Object

The pageContext object is an instance of javax.servlet.jsp.PageContext. It provides a single access point to all other implicit objects and helps manage attributes across different scopes such as page, request, session, and application.

Common Methods of pageContext object:

- setAttribute(String name, Object value) – Adds an attribute
- getAttribute(String name) – Retrieves an attribute
- removeAttribute(String name) – Removes an attribute

Example:

```
<%
pageContext.setAttribute("message", "Hello Page Context!");
String msg = (String) pageContext.getAttribute("message");
out.print(msg);
%>
```

This example stores and retrieves a message using the pageContext object.

The Page Object

The page object refers to the current JSP page itself. It acts similar to the keyword this in Java and can be used to call methods of the current servlet or to refer to the current JSP instance.

Example:

```
<%
out.print("Current page object: " + page.toString());
%>
```

This displays information about the current JSP page instance.

The Exception Object

The exception object is an instance of java.lang.Throwable. It is used to handle runtime errors in JSP pages and is available only on pages that are designated as error pages.

Example (error.jsp):

```
<%@ page isErrorPage="true" %>
<h3>Error Occurred:</h3>
<%= exception.getMessage() %>
```

When an error occurs, the exception object captures and displays the error message on the error page.

Advantages of Implicit Objects

1. The developer does not need to create these objects manually as they are provided automatically by the JSP container.
2. They simplify the interaction between the client and the server by providing built in access to important data.
3. They improve productivity by reducing repetitive code.
4. They allow better organization and easier maintenance of web applications

Scripting Elements

In **Java Server Pages (JSP)**, **scripting elements** are used to embed Java code directly inside an HTML page. These elements allow developers to write dynamic and interactive content by mixing HTML with Java code.

Scripting elements in JSP must always be written **within the special tags <% %>**. The **JSP engine** processes any Java code enclosed in these tags, while all other text outside the tags is treated as normal HTML content. This enables the creation of web pages that can respond dynamically to user input or data from a database.

The JSP engine converts these scripting elements into a **Java servlet** during the translation phase, and the Java code is executed on the server before the result is sent to the client browser.

There are **five main types of JSP scripting elements**:

1. JSP Comments
2. JSP Directives

3. JSP Declarations
4. JSP Scriptlets
5. JSP Expressions

1. JSP Comments

JSP comments are used to add notes or explanations in a JSP file that are **not visible in the client's browser**. The JSP container completely ignores these comments during translation and execution. They are useful for developers to include helpful notes, explanations, or debugging information.

There are **two types of comments** in a JSP page:

1. **JSP Comment:** Enclosed within <%-- --%> and ignored by the JSP engine.
2. **HTML Comment:** Enclosed within <!-- --> and visible in the page source when viewed in a browser.

Example:

```
<%@ page import="java.util.Date" %>
<!DOCTYPE html>
<html>
<head>
   <title>Comments in JSP Page</title>
</head>
<body>
   <%-- This is a JSP comment and will not appear in the HTML output --%>
   <!-- This is an HTML comment and can be seen in the browser source -->
   <p>The current date and time is: <%= new Date() %></p>
</body>
</html>
```

Explanation:

- The JSP comment inside <%-- --%> is processed only on the server and never appears in the client's HTML page.
- The HTML comment inside <!-- --> is visible if you view the page source in a web browser.

2. JSP Directives

JSP Directives are special instructions that provide global information to the JSP container about how to process the page. These directives are enclosed within <%@ %> and are processed during the **translation phase**, not during request handling.

There are three main types of directives:

1. Page Directive: Provides information about the entire JSP page such as import statements, error handling, session usage, and buffering.
Syntax:

2. <%@ page attribute="value" %>

Example:

```
<%@ page import="java.util.Date" contentType="text/html" %>
<html>
<body>
   <p>Today's date is: <%= new Date() %></p>
</body>
</html>
```

3. Include Directive: Includes another file during the translation phase. This is used for including static resources such as headers, footers, or navigation bars.
Syntax:

4. <%@ include file="filename.jsp" %>

Example:

```
<%@ include file="header.jsp" %>
<p>Main content of the page</p>
<%@ include file="footer.jsp" %>
```

5. Taglib Directive: Declares a custom tag library for use within the JSP page.
Syntax:

6. <%@ taglib uri="uri" prefix="prefixName" %>

Explanation: JSP directives are not used to generate output directly but to configure how the JSP page behaves during translation and execution.

3. JSP Declarations

A **declaration element** in JSP is used to **declare variables or methods** that can be used throughout the JSP page. Declarations are enclosed within <%! %>. The declared variables and methods become part of the servlet class that the JSP engine generates.

The declaration tag allows defining fields or functions outside the _jspService() method, so they can be accessed globally within the page.

Syntax:

<%! declaration; [declaration;]+ ... %>

Example 1: Declaring a Variable

```
<!DOCTYPE html>
<html>
<body>
   <%! int counter = 10; %>
   <p>The value of counter is: <%= counter %></p>
</body>
</html>
```

Example 2: Declaring a Method

```
<!DOCTYPE html>
<html>
<body>
   <%!
      int addNumbers(int a, int b) {
         return a + b;
      }
   %>
   <p>Sum of two numbers: <%= addNumbers(5, 10) %></p>
</body>
</html>
```

Explanation: The variable counter and the method addNumbers() are declared within <%! %> tags. These can be reused anywhere within the JSP file.

4. JSP Scriptlets

A **scriptlet** in JSP allows developers to write **Java code statements** that are executed each time the page is requested. Scriptlets are enclosed within <% %>.

All code written inside a scriptlet becomes part of the _jspService() method of the servlet generated from the JSP page. Scriptlets are often used to implement dynamic logic, retrieve parameters, perform calculations, or interact with JavaBeans.

Syntax:

<% Java code %>

Example 1: Basic Scriptlet

```
<!DOCTYPE html>
<html>
<head>
   <title>Scriptlet Example</title>
</head>
<%
int count = 5;
%>
<body>
   The count value is: <% out.println(count); %>
</body>
</html>
```

Example 2: Scriptlet with Loop

```
<table border="1">
<% for (int i = 1; i <= 5; i++) { %>
<tr>
   <td>Number</td>
   <td><%= i %></td>
</tr>
<% } %>
</table>
```

Explanation: In the first example, the integer variable count is declared and printed. In the second example, a loop is used within scriptlets to dynamically generate table rows.

Scriptlets are very powerful but should be used carefully to keep the code readable and maintainable. Modern JSP development encourages the use of tag libraries and JSTL instead of large scriptlets.

5. JSP Expressions

The **expression element** is used to output the result of a Java expression directly to the client. Expressions are enclosed within <%= %> tags.

The JSP engine evaluates the expression, converts it to a string, and inserts it into the response output. You do not need to use out.print() when using an expression tag because it automatically prints the value.

Syntax:

```
<%= expression %>
```

Example 1: Display a String

```
<!DOCTYPE html>
<html>
<body>
   <%= "Welcome to JSP Expression Example" %>
</body>
</html>
```

Example 2: Display a Calculation Result

```
<!DOCTYPE html>
<html>
<body>
   <p>The sum of 8 and 12 is: <%= 8 + 12 %></p>
</body>
</html>
```

Example 3: Display Current Date

```
<%@ page import="java.util.Date" %>
<!DOCTYPE html>
<html>
<body>
   <p>Current Date and Time: <%= new Date() %></p>
</body>
</html>
```

Explanation: Each expression inside <%= %> is automatically evaluated and sent to the browser. It is a concise and clean way to print dynamic values without explicitly calling output methods.

4.3 Scope and EL (Expression Language)

Expression Language (EL) is one of the most powerful and user-friendly features introduced in **JSP 2.0**. It was created to make it easier for web developers to **access and manipulate data stored in JSP scopes** such as request, session, application, and page without using Java code directly inside JSP files.

Before the introduction of EL, developers had to use **scriptlets** and **expression tags** like <%= request.getParameter("name") %> or <%= session.getAttribute("user") %> to retrieve and display data. This approach made the code lengthy, hard to read, and difficult to maintain.

EL provides a **simpler, cleaner, and more readable syntax** that allows developers to work with data directly using expressions inside ${}. The JSP engine automatically evaluates these expressions during runtime and inserts their values into the HTML output.

For example:
Without EL

<%= request.getParameter("username") %>

With EL

${param.username}

The above EL expression retrieves the same data in a single line, without requiring any Java method calls.

Why Expression Language is Used

The main purpose of EL is to **reduce the amount of Java code inside JSP pages** and make the presentation layer more readable. It bridges the gap between HTML design and Java-based server-side logic.

Key benefits include:

1. **Simplified data access:** You can directly access data from request, session, application, and page scopes without using Java code.
2. **Improved readability:** Pages look more like HTML templates and less like Java programs.
3. **Less maintenance:** With fewer Java statements in JSP pages, debugging and maintaining the code becomes easier.
4. **Type conversion:** EL automatically converts values (such as Strings, numbers, and Booleans) without the need for explicit casting.
5. **Powerful operations:** Supports arithmetic, logical, relational, and conditional operations.
6. **Integration with JSTL:** EL works seamlessly with JSTL (JSP Standard Tag Library) for looping, conditional checks, and displaying data.

Syntax of Expression Language

The basic syntax of EL is simple and uniform across all JSP pages. Every EL expression is enclosed between ${ and }.

Syntax:

${expression}

The JSP container evaluates the expression at runtime and automatically replaces it with its corresponding value in the final HTML output.

Example:

<h3>Welcome, ${param.username}</h3>

If the user entered “Arjun” in a form field named username, then the output will be:

Welcome, Arjun

Working Principle of Expression Language

When a JSP page containing EL is executed, the **JSP engine** performs the following steps:

1. It scans the page and identifies expressions enclosed within ${}.
2. It evaluates these expressions at runtime.
3. It searches for the variable in the following scopes in this order:
    - Page Scope
    - Request Scope
    - Session Scope
    - Application Scope
4. Once the variable is found, its value is returned and automatically converted into a string.
5. The evaluated value is inserted into the HTML response sent to the client browser.

If the variable is not found in any of the scopes, EL simply returns **null** instead of throwing an exception.

Common Uses of Expression Language

EL can perform a variety of tasks in JSP pages. Below are some of its most common applications.

1. Accessing Variables

You can directly access variables or attributes without using Java method calls.

${name}
${sessionScope.userName}
${applicationScope.totalVisitors}

2. Accessing Object Properties

EL can access properties of JavaBeans or custom objects using the dot operator.

${student.name}
${student.rollNumber}
${employee.salary}

If student or employee is an object available in one of the scopes, EL will automatically fetch and display the property value.

3. Accessing Request Parameters

EL can easily access form parameters sent by the client.

${param.email}
${param.password}

If a form field with the name email is submitted, EL automatically retrieves its value.

4. Arithmetic Operations

EL supports mathematical calculations directly inside the expression.

${10 + 5}
${price * quantity}
${100 / 4}
${total % 2}

Example:

```
<%
request.setAttribute("price", 250);
request.setAttribute("quantity", 3);
%>
<p>Total Cost: ₹${price * quantity}</p>
```

Output:

Total Cost: ₹750

5. Logical and Relational Operations

EL supports comparison and logical operators.

${age >= 18}
${marks < 40 || result == "fail"}
${salary != 0}

These expressions return either true or false.

6. Conditional Expressions

EL allows conditional decision-making using the ternary operator.

${age >= 18 ? "You can vote" : "You are too young to vote"}

7. Accessing Arrays and Collections

EL allows accessing elements of arrays, lists, or maps easily.

${fruits[0]}
${students[2].name}
${map.key}

Practical Examples of Expression Language

Example 1: Accessing Request Attributes

```
<%
request.setAttribute("city", "Bangalore");
%>
<h3>Your city is: ${requestScope.city}</h3>
```

Output:

Your city is: Bangalore

Example 2: Accessing Session Attributes

```
<%
session.setAttribute("user", "Priya");
%>
<p>Logged in as: ${sessionScope.user}</p>
```

Output:

Logged in as: Priya

Example 3: Accessing Application Attributes

```
<%
application.setAttribute("visitors", 1234);
%>
```

```
<p>Total Visitors: ${applicationScope.visitors}</p>
```

Output:

Total Visitors: 1234

Example 4: Conditional Expression

```
<%
request.setAttribute("age", 16);
%>
<h3>${age >= 18 ? "Eligible for voting" : "Not eligible for voting"}</h3>
```

Output:

Not eligible for voting

Example 5: Using EL in a Form

form.jsp

```
<form action="welcome.jsp" method="post">
Enter Name: <input type="text" name="username"><br>
<input type="submit" value="Submit">
</form>
```

welcome.jsp

```
<p>Welcome, ${param.username}!</p>
```

If the user enters “Ravi” in the form, the output will be:

Welcome, Ravi!

Example 6: Using EL with JSTL for Iteration

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%
List<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Mango");
fruits.add("Orange");
request.setAttribute("fruitList", fruits);
%>
```

```
<table border="1">
<tr><th>Fruit Name</th></tr>
<c:forEach var="fruit" items="${fruitList}">
<tr><td>${fruit}</td></tr>
</c:forEach>
</table>
```

Output:

Fruit Name
Apple
Mango
Orange

Operators in Expression Language

EL supports several categories of operators, similar to Java.

1. Arithmetic Operators

+, -, *, /, %

Example:

${10 + 5} → 15
${20 % 3} → 2

2. Relational Operators

==, !=, <, >, <=, >=

Example:

${marks >= 35}
${num1 == num2}

3. Logical Operators

&&, ||, !

Example:

${age > 18 && country == "India"}

4. Conditional Operator

condition ? value1 : value2

Example:

${gender == "M" ? "Male" : "Female"}

5. Empty Operator

Checks if a value or collection is empty or null.

Example:

${empty name}
${empty studentList}

Implicit Objects in Expression Language

EL provides **implicit objects** that automatically give access to commonly used data sources such as request parameters, session data, and application attributes.

Implicit Object	Description
pageScope	Refers to attributes available only within the current JSP page.
requestScope	Refers to attributes for the current client request.
sessionScope	Refers to attributes stored for the user session.
applicationScope	Refers to attributes shared across the entire web application.
param	Provides access to request parameters.
paramValues	Provides access to request parameters as arrays.
header	Provides access to HTTP request headers.
cookie	Provides access to cookies sent by the client.
initParam	Accesses initialization parameters from web.xml.
pageContext	Provides access to JSP-specific information like request, response, and session.

Table. 4.2 Implicit Objects in Expression Language

Scopes in Expression Language

Expression Language works closely with JSP scopes to manage data across pages and requests. There are four main scopes in EL.

1. Page Scope

- Attributes stored here are accessible only within the current JSP page.
- The data is lost once the page response is sent to the client.
- Accessed using ${pageScope.variable}.

2. Request Scope

- Attributes are available for a single client request and can be shared across servlets or JSPs using forward or include.
- Accessed using ${requestScope.variable}.

3. Session Scope

- Attributes are maintained throughout the user session.
- Data remains available across multiple pages until the session ends.
- Accessed using ${sessionScope.variable}.

4. Application Scope

- Attributes are available to all servlets and JSP pages in the application.
- Data remains until the server stops or the application is undeployed.
- Accessed using ${applicationScope.variable}.

Example Demonstrating All Scopes

```
<%
pageContext.setAttribute("data", "Page Data");
request.setAttribute("data", "Request Data");
session.setAttribute("data", "Session Data");
application.setAttribute("data", "Application Data");
%>

<p>Page Scope: ${pageScope.data}</p>
<p>Request Scope: ${requestScope.data}</p>
<p>Session Scope: ${sessionScope.data}</p>
<p>Application Scope: ${applicationScope.data}</p>
```

Output:

Page Scope: Page Data
Request Scope: Request Data
Session Scope: Session Data
Application Scope: Application Data

Reserved Words in Expression Language

EL includes several reserved keywords that cannot be used as variable names. These include: true, false, null, and, or, not, eq, ne, lt, le, gt, ge, div, mod, instanceof, and empty.

Comparison Between JSP Scriptlets and Expression Language

Operation	Using JSP Scriptlets	Using Expression Language
Retrieve request parameter	<%= request.getParameter("name") %>	${param.name}
Retrieve session attribute	<%= session.getAttribute("user") %>	${sessionScope.user}
Retrieve application attribute	<%= application.getAttribute("count") %>	${applicationScope.count}
Conditional check	<%= (age >= 18) ? "Adult" : "Minor" %>	${age >= 18 ? "Adult" : "Minor"}
Access bean property	<%= user.getEmail() %>	${user.email}

Table. 4.3 Comparison Between JSP Scriptlets and Expression Language.

4.4 JSP Standard Tag Libraries (JSTL)

JSTL stands for JavaServer Pages Standard Tag Library. It is a set of standardized tag libraries that encapsulate many common tasks that JSP developers used to implement with scriptlets and custom tags. The goals are

- make JSP pages more declarative and readable
- reduce and ideally eliminate Java scriptlets in view pages

- provide portable tags for control flow, iteration, formatting, localization, XML processing and simple database prototyping
- integrate cleanly with Expression Language so pages are terse and maintainable

JSTL is not a replacement for application architecture. It is a view layer convenience. Business logic, transaction management, data access and heavy XML or database work belong in services and DAOs. JSTL handles presentation tasks and light prototyping tasks in the JSP itself.

How to declare JSTL in a JSP page

At the top of a JSP add taglib directives. Use the official URIs and choose a short prefix.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

Once declared the tags appear as XML like <c:forEach>, <fmt:formatNumber>, <sql:query>, <x:parse>, and functions like ${fn:length(myList)} can be used in EL.

The core tag library c in depth

Core tags are the workhorses. They provide variable management, flow control, iteration, dynamic includes, URL building, and redirection.

c:set

Purpose

- assign or modify an attribute in a given scope or a page variable
  Key attributes
- var name of the variable
- value expression to assign
- scope optional one of page request session application
  Two forms

1. empty body with value attribute

```
<c:set var="username" value="${param.username}" />
```

2. body form collects body text

```
<c:set var="intro">
```

```
  Welcome to the advanced JSTL guide
</c:set>
```

Scope rules and precedence

- if scope omitted the variable is placed in page scope
- recommended to always document scope when sharing data across pages to avoid ambiguity

c:out

Purpose

- print a value while escaping XML special characters by default
  Key attributes
- value the expression to print
- default if value is null print default instead

Example

```
<c:out value="${user.displayName}" default="Guest" />
```

Use this instead of direct EL output when the value may contain user input and you want safe HTML.

c:if

Purpose

- conditional rendering of body content when test evaluates to true
  Key attribute
- test EL expression

Example

```
<c:if test="${not empty param.email}">
  <p>Thanks for providing your email</p>
</c:if>
```

c:choose, c:when, c:otherwise

Purpose

- mutually exclusive branching similar to switch or chained if else

Example

```
<c:choose>
  <c:when test="${score >= 90}">Excellent</c:when>
  <c:when test="${score >= 75}">Very Good</c:when>
  <c:otherwise>Keep working</c:otherwise>
</c:choose>
```

c:forTokens

Purpose

- tokenize a delimited string and iterate tokens
  Attributes
- items the string
- delims delimiter characters
- var token variable name

Example

```
<c:forTokens items="${param.tags}" delims="," var="tag">
  <span>${tag}</span>
</c:forTokens>
```

c:url, c:param, c:redirect

Purpose

- build safe URLs and redirect
- c:url builds a context relative URL and optionally includes parameters. If you use sessions with URL rewriting, c:url handles adding ;jsessionid correctly.

Example build URL

```
<c:url var="link" value="/search">
  <c:param name="q" value="${param.q}" />
  <c:param name="page" value="${param.page != null ? param.page : 1}" />
</c:url>
<a href="${link}">Search</a>
```

c:redirect performs a server side redirect

```
<c:redirect url="${loginUrl}" />
```

c:import

Purpose

- import content from another resource at runtime and expose it as the page output or as a string variable
  Attributes
- url the resource to import
- var optional to capture output into a variable
  Example import into var

```
<c:import url="/header.jsp" var="headerHtml" />
<c:out value="${headerHtml}" escapeXml="false" />
```

Use with caution because importing remote URLs may block the page if the network is slow.

c:catch

Purpose

- capture exceptions that occur inside the body and handle them gracefully
  Attributes
- var optional to bind the exception
  Example

```
<c:catch var="err">
  <c:import url="http://remote-service/api?param=${param.q}" />
</c:catch>
<c:if test="${not empty err}">
  <p>Service is unavailable. Please try later.</p>
</c:if>
```

common usage patterns and tips for core tags

- Use c:set with explicit scope when you intend to pass values to include pages or servlets
- Prefer <c:out> for user generated content to avoid XSS
- When iterating large collections consider pagination in the controller rather than fetching entire collection in JSP
- Keep logic light; complex calculations and data access belong to controllers or services

Formatting and internationalization fmt in depth

The fmt library focuses on locale aware formatting and message bundles.

fmt:formatNumber and fmt:parseNumber

Format number for display

Attributes

- value number or EL expression
- type number currency percent
- pattern optional Java number format pattern
  Example

<fmt:formatNumber value="${order.total}" type="currency" />
<fmt:formatNumber value="${distance}" maxFractionDigits="1" />

Parse string to number (careful with localization)

<fmt:parseNumber var="n" value="${param.amount}" />

fmt:formatDate and fmt:parseDate

Format Java Date or Calendar objects

Attributes

- value date object or milliseconds
- pattern a SimpleDateFormat pattern
- type date time or both
  Example

<fmt:formatDate value="${order.date}" pattern="dd MMM yyyy HH:mm" />

fmt:setLocale and fmt:setBundle and fmt:message

Locale selection and resource bundles

- fmt:setLocale sets the page locale
- fmt:setBundle loads a resource bundle by basename
- fmt:message prints localized messages from the bundle

Example

```
<fmt:setLocale value="${param.lang != null ? param.lang : 'en_US'}" />
<fmt:setBundle basename="i18n.messages" var="msgs" />
<p><fmt:message key="welcome" bundle="${msgs}" /></p>
```

fmt:requestEncoding

Set the request encoding for proper form handling

```
<fmt:requestEncoding value="UTF-8" />
```

Always set encoding early in the page to ensure parameters are read correctly.

SQL tag library sql with important caveats

The SQL tags are convenient for quick demos and learning but have serious limitations and risks if used in production.

Basic operations

- sql:setDataSource define a simple DataSource for prototyping
- sql:query run a SELECT and return a Result object with rows
- sql:update run INSERT UPDATE DELETE
- sql:param bind parameters

Example query

```
<sql:setDataSource var="ds" driver="com.mysql.cj.jdbc.Driver"
   url="jdbc:mysql://localhost:3306/shop" user="sa" password="sa" />

<sql:query dataSource="${ds}" var="result">
  SELECT id, name, price FROM products WHERE category = ?
  <sql:param value="${param.cat}" />
</sql:query>

<c:forEach var="r" items="${result.rows}">
  <div>${r.name} - ${r.price}</div>
</c:forEach>
```

Caveats and warnings

- sql:setDataSource is single connection style and not connection pooled. Not for production.
- Embedding SQL in JSP violates separation of concerns

- SQL tags are vulnerable to misuse. Use prepared statements and parameter binding always via <sql:param>
- Do not store credentials in JSP files. Use JNDI DataSource configured in container for production
- Prefer frameworks like JPA, MyBatis or JDBC DAOs for real applications

Transaction support

sql:transaction groups operations and commits or rollbacks together. Still not a substitute for proper container managed transactions.

XML tag library x with examples

XML JSTL is useful when you need light weight XML parsing, XPath queries and XSL transforms directly in the view layer.

x:parse

Parse XML from body or url into a variable

```
<x:parse var="doc">
  <root><item id="1">A</item><item id="2">B</item></root>
</x:parse>
```

Or parse a remote XML

```
<x:parse var="doc" xml="${param.xmlUrl}" />
```

x:out and XPath

Use XPath expressions to extract nodes or values

```
<x:out select="$doc/root/item[1]" />
```

x:forEach select

Iterate node sets returned by XPath

```
<x:forEach select="$doc/root/item" var="it">
   <div><x:out select="$it" /></div>
</x:forEach>
```

x:transform

Apply XSLT transformation to an XML node with optional parameters

```
<x:transform xml="${doc}" xslt="/WEB-INF/xslt/report.xsl">
  <x:param name="title" value="Monthly Report" />
</x:transform>
```

Practical tips for XML JSTL

- Ensure your container has the XML/XSLT implementation jars if not included
- For large XML documents, parsing in the controller and supplying parsed object to JSP may be more efficient
- Use XPath carefully and test expressions in isolation

Functions library fn explained

Functions are static utilities available for use inside EL. They return values and can be composed.

Common functions and usage patterns

```
${fn:length(list)}              // size of list or length of string
${fn:contains(name, 'Raj')}     // boolean
${fn:toLowerCase(username)}
${fn:toUpperCase(title)}
${fn:split(tags, ',')[0]}
${fn:replace(text, 'old', 'new')}
${fn:trim(name)}
```

Functions are especially helpful in views to avoid having to create small helper objects or scriptlets.

Integration with Expression Language

- JSTL tags are primarily designed to be used with EL expressions for values and tests
- The recommended approach is to do data setup in servlets or controllers and use JSTL tags for presentation
- Combine tags and EL to minimize Java code in JSP

Example combined usage

```
<c:set var="taxRate" value="${applicationScope.taxConfig.defaultRate}" />
```

```
<c:forEach var="p" items="${products}">
  <div>${p.name} - Total: ${p.price + (p.price * taxRate)}</div>
</c:forEach>
```

Security considerations

- **Cross site scripting:** Always escape untrusted user input. Use <c:out value="${...}" /> which escapes HTML by default. If you must render HTML, ensure the content is sanitized before printing and set escapeXml="false" only with caution.
- **SQL injection:** Never concatenate user input into SQL. Use <sql:param> to pass values. For production use parameterized queries in DAOs.
- **Remote import:** c:import and x:parse with remote URLs can introduce SSRF or availability issues. Validate or restrict remote endpoints.
- **Sensitive data:** Do not expose credentials, secrets or internal information in JSP. Use container level JNDI resources.

Performance and scalability tips

- Avoid heavy computation or database access in JSP. Prepare data in controllers.
- Use scoped attributes judiciously. Session and application scopes are memory consumers.
- Avoid repeatedly parsing or transforming the same XML in multiple requests. Cache results in application scope if safe.
- Use container pooling and JNDI DataSource for DB connections rather than sql:setDataSource.
- Minimize nested loops in JSP and expensive EL expressions inside loops.
- Consider using jsp:include versus c:import depending on whether you want translation time versus request time inclusion.

Developing Dynamic Web Applications with JSP

Developing dynamic web applications with **JavaServer Pages (JSP)** involves creating interactive, data-driven websites that respond intelligently to user requests. JSP is a **server-side technology** that allows developers to combine static content, such as HTML, CSS, and JavaScript, with dynamic elements generated using Java. It is part of the Java EE (Enterprise Edition) platform and provides an efficient way to build robust, scalable, and maintainable web applications.

JSP in Web Development

In traditional static websites, every page is fixed the same content is displayed for all users. However, most modern web applications require **dynamic behavior**, such as displaying personalized user information, processing forms, connecting to databases, and generating

content based on conditions or inputs. JSP was introduced to simplify this process by embedding Java code directly into HTML pages, allowing developers to **generate dynamic content on the server before sending it to the client's browser**.

When a user requests a JSP page, the **web server** forwards the request to a **JSP engine**. The JSP engine converts the page into a **servlet**, a Java class that runs on the server. The servlet processes the request, executes Java code (such as retrieving data from a database), and generates HTML as a response. This response is then sent back to the client's browser for display.

How JSP Supports Dynamic Content

JSP makes it possible to develop dynamic web applications by providing a seamless way to mix **presentation logic (HTML)** with **server-side logic (Java)**. Through **Expression Language (EL)** and **JSP Standard Tag Library (JSTL)**, developers can write clean, readable pages that separate design and functionality. Instead of writing complex Java code directly inside JSP, EL expressions such as ${user.name} can be used to access server-side data easily. Similarly, JSTL tags like <c:forEach> or <c:if> help control iteration and conditional rendering without using Java scriptlets.

For example:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<h3>Welcome, ${sessionScope.username}</h3>
<c:if test="${not empty sessionScope.username}">
   <p>Your account balance is: ${sessionScope.balance}</p>
</c:if>
```

This snippet dynamically displays user data stored in the session. The content changes depending on the user who is logged in this is the essence of a **dynamic web page**.

JSP in the MVC Architecture

In modern Java web applications, JSP is often used as the **view layer** in the **Model-View-Controller (MVC)** architecture.

- The **Model** represents the business logic and data (for example, JavaBeans or database records).
- The **Controller** (usually a servlet) handles client requests, processes business logic, and determines which JSP page to display.
- The **View** (JSP) is responsible for presenting data to the user in a readable format.

This separation of concerns improves maintainability and reusability. For example, if you need to change how data is displayed, you only modify the JSP file without touching the business logic.

Working with Databases Using JSP

Dynamic web applications typically interact with databases to store and retrieve data. JSP can easily connect to databases using **JDBC (Java Database Connectivity)** or through frameworks and tag libraries such as **JSTL SQL tags** for quick prototyping. For instance, you can use JSP to fetch and display user data stored in a MySQL database.

A servlet may retrieve data from the database and set it as an attribute:

```
request.setAttribute("products", productList);
request.getRequestDispatcher("/products.jsp").forward(request, response);
```

Then, in the JSP page, you can iterate over that data:

```
<c:forEach var="p" items="${products}">
  <p>${p.name} - Price: ${p.price}</p>
</c:forEach>
```

This demonstrates how JSP and servlets work together to generate content dynamically based on backend data.

Advantages of Using JSP for Dynamic Applications

JSP provides several advantages when developing dynamic web applications. It simplifies web development by allowing Java code and HTML to coexist seamlessly. It supports **rapid development** through reusable components such as **JavaBeans**, **custom tags**, and **JSTL**, which promote modular design. JSP pages are compiled into servlets automatically by the server, which ensures **high performance** and **portability** across platforms. It also integrates well with other Java technologies like Servlets, JDBC, and Enterprise JavaBeans (EJB), making it suitable for both small websites and large enterprise applications.

Another major advantage is the **separation of presentation and business logic**. Designers can focus on HTML and user interface, while developers handle logic through servlets and JavaBeans. This separation makes collaboration easier and the application more maintainable.

Example: Dynamic Login Page Using JSP and Servlet

Below is a simple example showing how JSP and Servlets work together to create a dynamic login feature:

login.jsp

```
<form action="LoginServlet" method="post">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
```

```
  <input type="submit" value="Login">
</form>
```

LoginServlet.java

```
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String user = request.getParameter("username");
    String pass = request.getParameter("password");

    if("admin".equals(user) && "1234".equals(pass)) {
       request.setAttribute("username", user);
       request.getRequestDispatcher("welcome.jsp").forward(request, response);
    } else {
       response.sendRedirect("error.jsp");
    }
  }
}
```

welcome.jsp

```
<h2>Welcome, ${requestScope.username}!</h2>
<p>You have successfully logged in.</p>
```

This simple example illustrates how a JSP page interacts with a servlet to generate dynamic responses based on user input.

Key Features Supporting Dynamic Development

1. **Expression Language (EL):** Simplifies accessing data from JavaBeans, request parameters, and scope objects.
2. **JSTL (JSP Standard Tag Library):** Provides standard tags for loops, conditions, database access, and internationalization.
3. **JavaBeans Integration:** Allows encapsulating data and logic in reusable components that can be accessed through JSP pages.
4. **Custom Tags:** Extend JSP functionality by defining reusable tag sets for specific application needs.
5. **Implicit Objects:** Predefined objects like request, response, session, and application provide access to the request-response cycle and stored data.

Practical

1. Design a responsive UI form using Bootstrap and React to capture user input and display the result

Aim

To **design and implement a responsive user input form** using **Bootstrap and React.js** that captures user data (like name, email, and message) and dynamically displays the entered information on the same page.

Procedure

1. **Create a React project**: Open the terminal and create a new React app:
2. npx create-react-app responsive-form
3. cd responsive-form
4. **Install Bootstrap**: Install Bootstrap for styling and responsiveness:
5. npm install bootstrap

   Import Bootstrap CSS in the index.js or App.js file:

   import 'bootstrap/dist/css/bootstrap.min.css';

6. **Design a responsive form**: Use Bootstrap grid and form classes to create a responsive form layout with input fields (Name, Email, Message) and a Submit button.
7. **Capture user input**: Use React's useState hook to manage form data and update state as the user types.
8. **Display the result**: After submission, display the entered data dynamically below the form using React components.
9. **Run and test the project**: Start the development server using:
10. npm start

Test the form in a browser and check the responsiveness by resizing the window.

Program (React Code)

File: App.js

```
import React, { useState } from "react";
import 'bootstrap/dist/css/bootstrap.min.css';

function App() {
  // State variables for form fields
  const [formData, setFormData] = useState({
```

```
  name: "",
  email: "",
  message: ""
});

const [submittedData, setSubmittedData] = useState(null);

// Handle input change
const handleChange = (e) => {
  const { name, value } = e.target;
  setFormData({ ...formData, [name]: value });
};

// Handle form submission
const handleSubmit = (e) => {
  e.preventDefault();
  setSubmittedData(formData);
};

return (
  <div className="container mt-5">
    <div className="card shadow-lg p-4 rounded-4">
      <h2 className="text-center text-primary mb-4">User Information Form</h2>
      <form onSubmit={handleSubmit}>
        <div className="mb-3">
          <label className="form-label fw-bold">Name:</label>
          <input
            type="text"
            className="form-control"
            name="name"
            placeholder="Your full name"
            value={formData.name}
            onChange={handleChange}
            required
          />
        </div>

        <div className="mb-3">
          <label className="form-label fw-bold">Email:</label>
          <input
            type="email"
            className="form-control"
            name="email"
            placeholder="you@example"
            value={formData.email}
```

```
            onChange={handleChange}
            required
          />
        </div>

        <div className="mb-3">
          <label className="form-label fw-bold">Message:</label>
          <textarea
            className="form-control"
            name="message"
            rows="3"
            placeholder="Write your message here"
            value={formData.message}
            onChange={handleChange}
            required
          ></textarea>
        </div>

        <button type="Submit" className="btn btn-primary w-100">
          Submit
        </button>
      </form>
    </div>

    {submittedData && (
      <div className="card mt-5 p-4 border-success border-2 rounded-4">
        <h4 className="text-success">Submitted Information</h4>
        <hr />
        <p><strong>Name:</strong> {submittedData.name}</p>
        <p><strong>Email:</strong> {submittedData.email}</p>
        <p><strong>Message:</strong> {submittedData.message}</p>
      </div>
    )}
   </div>
  );
}

export default App;
```

Explanation

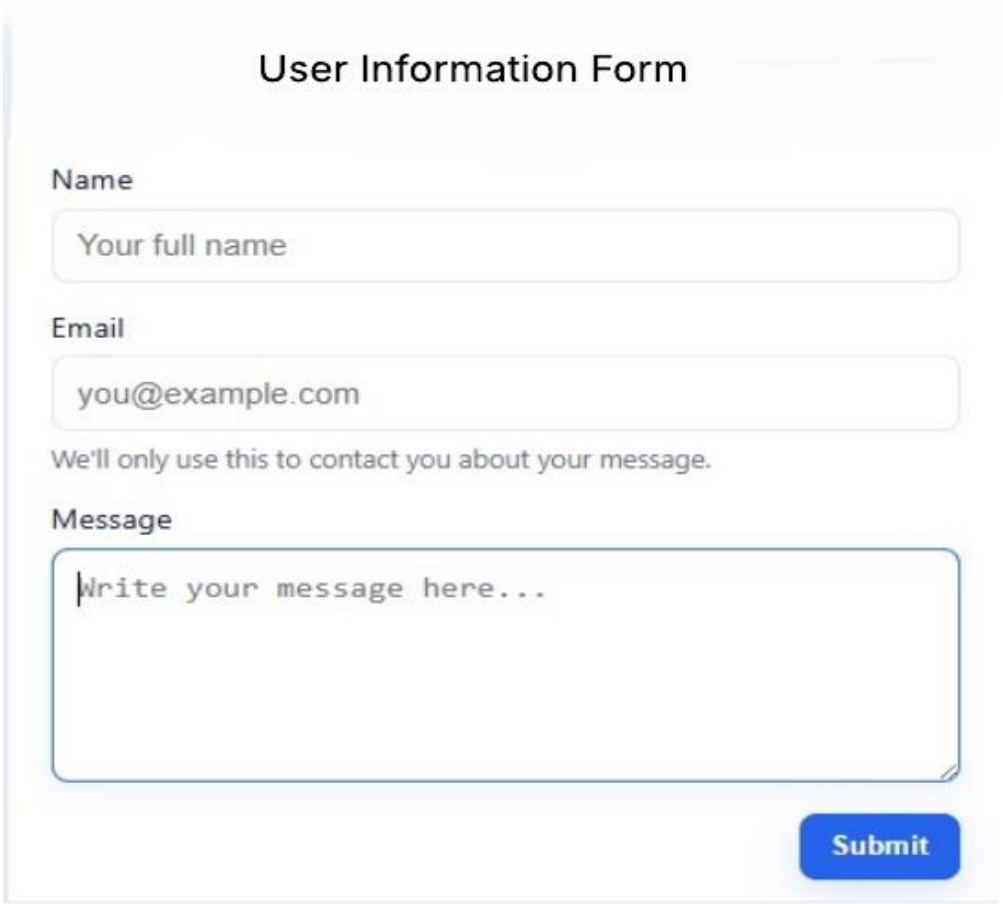
1. **React Hooks (useState)** are used to manage form state dynamically.
2. **Bootstrap classes** (container, form-control, card, btn, text-center) ensure the layout is responsive and visually appealing.

3. When the form is submitted, the input data is stored in a state variable (submittedData) and displayed immediately below the form.
4. The form is **mobile-friendly**, with all elements automatically resizing and aligning properly on smaller screens.

Sample Output

Before Submission

A clean, centered form appears:



After Submission

Once the user clicks “Submit”, the entered data appears dynamically below:

Submitted Information

Name: Priya Sharma
Email: priya@gmail.com

Message: Excited to learn React with Bootstrap!

Result

A responsive UI form using Bootstrap and React to capture user input was successfully implemented.

2. Create a JSP application using scripting elements, EL for data display, and JSTL for conditional and loop–based rendering.

Aim

To develop a **JSP web application** that uses **scripting elements** for server-side logic, **Expression Language (EL)** for data display, and **JSTL (JavaServer Pages Standard Tag Library)** for performing **conditional and loop-based rendering** dynamically on a JSP page.

Procedure

1. **Set up a web application project** in your IDE or create manually inside webapps folder in Tomcat.
2. **Add the JSTL library** to the project's WEB-INF/lib folder.
3. **Create a JSP file** (student.jsp) to demonstrate:
   - **Scripting elements** for initialization logic.
   - **Expression Language (EL)** for accessing attributes and displaying data.
   - **JSTL tags** for performing conditional checks and looping through data.
4. **Configure JSTL Tag Library Directive** at the top of the JSP file.
5. **Run the JSP file** on a Tomcat server and observe dynamic data rendering.
6. **Verify output** for both conditional rendering and looping sections.

Program: student.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<!DOCTYPE html>
<html>
<head>
   <title>JSP Application using Scripting, EL, and JSTL</title>
   <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">
</head>
<body class="bg-light">
<div class="container mt-5">
   <div class="card shadow-lg p-4 rounded-4">
```

```
    <h2 class="text-center text-primary mb-4">Student Details using JSP, EL, and JSTL</h2>

    <%-- ===== JSP SCRIPTING ELEMENTS ===== --%>
    <%
        // Scriptlet: Create and initialize student data
        String studentName = "Rahul Sharma";
        int studentAge = 20;
        double percentage = 85.5;

        // Declaration: Declare a method to return grade
        String getGrade(double per) {
            if (per >= 90) return "A+";
            else if (per >= 80) return "A";
            else if (per >= 70) return "B";
            else return "C";
        }

        // Expression: Display calculated grade directly
        String studentGrade = getGrade(percentage);

        // Set attributes for JSTL and EL access
        request.setAttribute("name", studentName);
        request.setAttribute("age", studentAge);
        request.setAttribute("percentage", percentage);
        request.setAttribute("grade", studentGrade);

        // Create an array of subjects
        String[] subjects = {"Mathematics", "Physics", "Chemistry", "English", "Computer Science"};
        request.setAttribute("subjects", subjects);
    %>

    <%-- ===== DATA DISPLAY USING EXPRESSION LANGUAGE ===== --%>
    <div class="mt-3">
        <h4 class="text-success">Student Information:</h4>
        <p><b>Name:</b> ${name}</p>
        <p><b>Age:</b> ${age}</p>
        <p><b>Percentage:</b> ${percentage}%</p>
        <p><b>Grade:</b> ${grade}</p>
    </div>

    <%-- ===== CONDITIONAL RENDERING USING JSTL ===== --%>
    <div class="mt-4">
        <h4 class="text-info">Performance Evaluation:</h4>
```

```
        <c:choose>
          <c:when test="${percentage >= 90}">
            <p class="text-success">Excellent Performance! Keep it up.</p>
          </c:when>
          <c:when test="${percentage >= 75}">
            <p class="text-primary">Good Job! You performed very well.</p>
          </c:when>
          <c:otherwise>
            <p class="text-danger">Needs Improvement! Work harder next time.</p>
          </c:otherwise>
        </c:choose>
      </div>

      <%-- ===== LOOP-BASED RENDERING USING JSTL FOR SUBJECT LIST
===== --%>
      <div class="mt-4">
        <h4 class="text-warning">Subjects Enrolled:</h4>
        <table class="table table-bordered table-striped">
          <thead class="table-dark">
            <tr>
              <th>Sl. No</th>
              <th>Subject Name</th>
            </tr>
          </thead>
          <tbody>
            <c:forEach var="sub" items="${subjects}" varStatus="st">
              <tr>
                <td>${st.index + 1}</td>
                <td>${sub}</td>
              </tr>
            </c:forEach>
          </tbody>
        </table>
      </div>

      <%-- ===== FOOTER SECTION ===== --%>
      <footer class="text-center text-muted mt-4">
        <p>&copy; 2025 JSP Dynamic Application Example</p>
      </footer>
    </div>
</div>
</body>
</html>
```

Sample Output

Before JSP Execution (in code):

Name = "Rahul Sharma"
Age = 20
Percentage = 85.5
Grade = "A"
Subjects = ["Mathematics", "Physics", "Chemistry", "English", "Computer Science"]

Rendered Output in Browser:

Student Details using JSP, EL, and JSTL

Student Information:
Name: Rahul Sharma
Age: 20
Percentage: 85.5%
Grade: A

Performance Evaluation:
Good Job! You performed very well.

Subjects Enrolled:

Sl. No	Subject Name
1	Mathematics
2	Physics
3	Chemistry
4	English
5	Computer Science

© 2025 JSP Dynamic Application Example

Result

A **JSP application** was successfully developed using **scripting elements**, **Expression Language (EL)**, and **JSTL**.The application dynamically displays student information, evaluates performance using conditional tags, and lists enrolled subjects using a looping tag.

CHAPTER – 5

JAVA FRAME WORKS

5.1 MVC Architecture

Model View Controller or MVC is a widely used software design pattern that separates an application into three cooperating components. The goal is to separate concerns so that data management, user interface, and input handling are independent. This separation improves organization, maintainability, testability, and the ability for teams to work in parallel. Below is an elaborate, practical, example rich explanation written in paragraphs with clear subheadings.

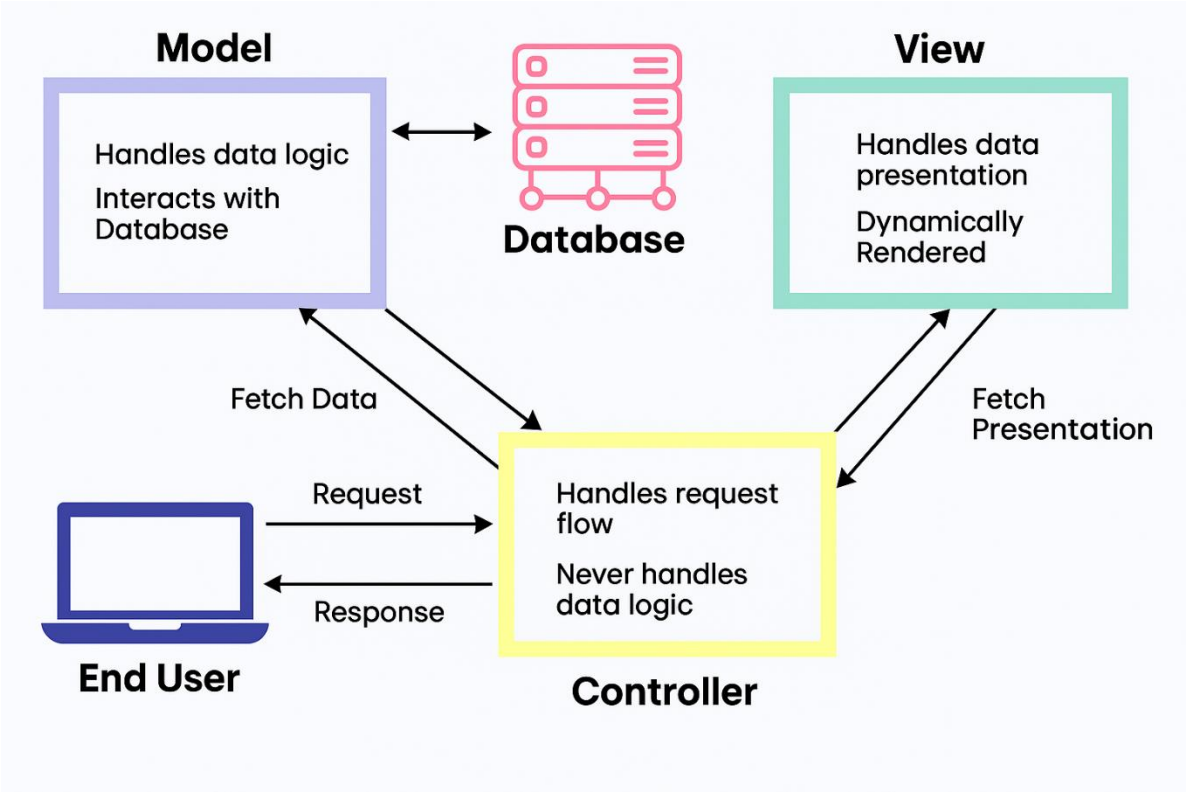


Fig. 5.1 MVC Architecture.

MVC

MVC is a structural pattern that divides an application into three parts. The **Model** holds data and business rules. The **View** renders the user interface and presents model data to the user. The **Controller** receives user input, interacts with the model to perform actions or fetch data, and chooses which view should render the result. The pattern does not mandate a single technical stack; it is an architectural idea that can be implemented in web frameworks, desktop apps, mobile apps, and single page applications.

Why MVC matters

The primary purpose of MVC is separation of concerns. By keeping data logic, UI rendering, and input handling separate, teams can develop components in parallel: backend engineers work on models and services, UI designers implement views, and middleware or routing logic is handled in controllers. This modularity improves reusability, easier testing at component level, and cleaner code bases that scale better as features grow.

The Model

The Model encapsulates application data, state and business logic. It may be a single object, a set of domain classes, or a layer of service classes that talk to databases and external APIs. Models are responsible for validation, persistence rules, domain operations and often expose methods the controller can call.

Example responsibilities

- Validation rules (for example, ensuring a booking date is in the future)
- Persistence operations (CRUD via repositories or DAOs)
- Business computations (price calculations, tax, availability checks)

Simple pseudo example in Java (domain class):

```
public class Booking {
  private String userId;
  private String hotelId;
  private LocalDate date;
  private BigDecimal price;
  // getters setters and domain logic
  public boolean isValid() { return date.isAfter(LocalDate.now()); }
}
```

In larger systems the model layer also includes service classes (BookingService) and repository classes (BookingRepository).

The View

The View is the presentation layer. Its job is to render the model into a human readable form and present UI controls for user interaction. In server rendered web apps the view is typically HTML templates or JSP pages. In single page applications views are React or Angular components. A key point is that the view should contain minimal business logic; it should be concerned with presentation, formatting, and calling controller actions or emitting UI events.

Example view responsibilities

- Rendering lists of items and detail pages
- Formatting numbers and dates for locale
- Binding UI inputs to controller actions or events

Small EJS view snippet (Node):

```
<ul>
  <% bookings.forEach(function(b) { %>
    <li><%= b.user %> booked <%= b.hotel %> on <%= b.date.toDateString() %></li>
  <% }) %>
</ul>
```

The Controller

The Controller is the coordinator. It receives input events (HTTP requests, button clicks), extracts user intent, invokes the appropriate model operations, and selects the view for response. Controllers handle routing, map request parameters to model calls, orchestrate transactions, and decide on redirects, errors, or rendering templates.

Example responsibilities

- Parsing and validating request parameters
- Calling model methods and services
- Choosing which view to return and what data to pass to it

Simple Express controller (Node):

```
// controllers/bookingController.js
exports.createBooking = async (req, res) => {
  const { user, hotel, date, price } = req.body;
  try {
    await bookingService.create({ user, hotel, date, price });
    res.redirect('/bookings');
  } catch (err) {
    res.status(500).render('error', { error: err });
```

```
  }
};
```

Typical request flow

A common flow in an MVC web app follows these steps:

1. Browser sends a request to a URL.
2. Router maps the URL to a controller action.
3. Controller reads inputs, calls model / service.
4. Model interacts with the database and returns results.
5. Controller chooses a view and provides model data.
6. View renders HTML (or JSON) and it is returned to the client.

This flow makes debugging easier because you can inspect each stage independently.

Implementation examples across stacks

1. Server side rendering with Java and Spring MVC

Spring MVC maps URLs to controller methods, returns a view name and adds model attributes:

```
@Controller
public class BookingController {
  @GetMapping("/bookings")
  public String list(Model model) {
    List<Booking> bookings = bookingService.findAll();
    model.addAttribute("bookings", bookings);
    return "bookings"; // renders bookings.jsp or bookings.html
  }
}
```

Model and repository are separate. Views can be JSP, Thymeleaf, or other template engines.

2. Node.js with Express and EJS (example in your prompt)

Express routers map to controller functions which render EJS views using data from Mongoose models:

```
router.get('/bookings', bookingController.getBookings);
```

Model is Booking Mongoose schema, controller fetches bookings and calls res.render('bookings', { bookings }).

3. Django (MVT variant)

Django uses Model View Template naming but maps similarly: view functions act like controllers, templates are views and models are ORM classes. Example:

```
# views.py
def bookings(request):
    bookings = Booking.objects.all()
    return render(request, 'bookings.html', {'bookings': bookings})
```

4. Single page applications with React and Redux

SPA architecture often maps MVC concepts differently: React components are views, actions and reducers (Redux) act as controllers and models respectively, or the model is the backend API. Example flow:

- UI dispatches action -> async thunk calls API -> API returns data -> reducer stores data in state -> view re-renders with new model state.

Asynchronous programming in MVC

Modern web apps need to handle asynchronous operations. MVC supports async by:

- Making model calls asynchronous (non blocking database or API calls).
- Controllers using async/await or callback mechanisms to call models and respond when results arrive.
- Views rendered client side updated via AJAX or fetch without full page reload.

Example Node controller with async:

```
exports.getBookings = async (req, res) => {
  const bookings = await Booking.find().exec();
  res.render('bookings', { bookings });
};
```

On the client an AJAX call can fetch JSON and update a view component.

Advantages of MVC summarized

- **Separation of concerns** makes codebase clearer.
- **Parallel development** allows front end and back end teams to work independently.
- **Reusability** of models across multiple views or controllers.
- **Testability** since controllers and models can be unit tested independent of views.
- **Maintainability and scalability** as features grow.

Common challenges and pitfalls

- **Over engineering for small apps** MVC can add unnecessary layers for trivial projects.
- **Improper separation** where views contain business logic or models directly render HTML defeats the purpose.
- **Too many responsibilities in controllers** leading to fat controllers. Controllers should orchestrate but not implement heavy business logic. That belongs in services or model layer.
- **Complex cross cutting concerns** like authentication and logging should be handled by middleware or filters rather than controllers.

5.2 Spring Framework

The Spring Framework is a lightweight and powerful Java framework widely used for developing scalable and maintainable enterprise-level applications. It provides a well-structured programming and configuration model for Java-based development. Spring simplifies the complexity of building enterprise software by providing an organized architecture that encourages good practices such as modularity, flexibility, and loose coupling between different components.

Evolution of the Spring Framework

The Spring Framework was first introduced in June 2003 under the Apache 2.0 license. Over the years, it has evolved through several major updates and improvements. Spring 2.0 introduced XML namespaces and support for AspectJ. Later, Spring 2.5 brought annotation-based configuration, allowing developers to configure applications more easily using annotations instead of XML files.

Spring 3.0 introduced the Java-based configuration model using the @Configuration annotation, while Spring 4.0 added support for Java EE 7 features such as JMS 2.0, JPA 2.1, Bean Validation 1.1, Servlet 3.1, and JCache. Spring 5.0 introduced support for reactive programming and required Java 8 or later. As of 2025, the latest version is Spring Framework 6, which supports Java 17 and above, Jakarta EE 10, and native compilation using GraalVM. Although older Java versions can still be used, Java SE 6 remains the minimum requirement.

Benefits of Using the Spring Framework

Spring offers several advantages that make it a preferred choice for Java developers. The first major benefit is simplified development. By using concepts such as dependency injection and aspect-oriented programming, Spring helps developers reduce repetitive and boilerplate code, making the process faster and more efficient.

Another important benefit of Spring is its ability to promote loose coupling between different parts of an application. With dependency injection, classes do not directly depend on one another. Instead, dependencies are managed by the Spring container, which improves maintainability and makes the code easier to test.

Spring also provides a modular architecture that allows developers to include only the components they need. This makes applications lighter, more efficient, and easier to customize.

Integration is another strong feature of Spring. It provides built-in support for technologies like JDBC, JMS, and JPA, enabling developers to easily connect with databases and other enterprise systems.

Finally, the framework is highly scalable. Its flexible and lightweight design supports the creation of applications that can grow from small systems to large-scale enterprise platforms without compromising performance.

Key Features of the Spring Framework

Dependency Injection

Dependency Injection is one of the most significant features of the Spring Framework. It is a design pattern where the Spring container provides the required dependencies to a class automatically. This approach eliminates the need for a class to create its own dependencies, leading to more modular and testable code.

For example, consider a Library class that depends on a Book class. Instead of the Library creating a Book object directly, Spring injects it from the outside.

```
// Constructor Injection
public class Library {
   private Book book;

   public Library(Book book) {
      this.book = book;
   }
}
```

Spring also supports setter-based injection, where dependencies are provided through setter methods after the object is created.

```
// Setter Injection
public class Library {
   private Book book;

   public void setBook(Book book) {
      this.book = book;
   }
}
```

Lastly, field injection allows Spring to inject dependencies directly into fields using annotations such as @Autowired.

```
// Field Injection
public class Library {
   @Autowired
   private Book book;
}
```

Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming in Spring allows developers to separate cross-cutting concerns like logging, security, and transaction management from the core business logic. This helps make the application code cleaner and easier to maintain. For instance, logging actions can be implemented separately as an aspect and automatically applied across multiple classes without rewriting the same code everywhere.

Transaction Management

Spring provides a consistent and reliable abstraction layer for managing transactions. It ensures that operations on databases or messaging systems are completed correctly and safely. Whether the underlying technology is JDBC or JPA, Spring handles the transaction logic uniformly.

Spring MVC

Spring MVC is a powerful framework that follows the Model View Controller pattern for web application development. It separates the business logic, presentation layer, and control flow, allowing each part to be managed independently. The DispatcherServlet in Spring MVC acts as the front controller, routing requests to appropriate controllers and returning responses to the user interface.

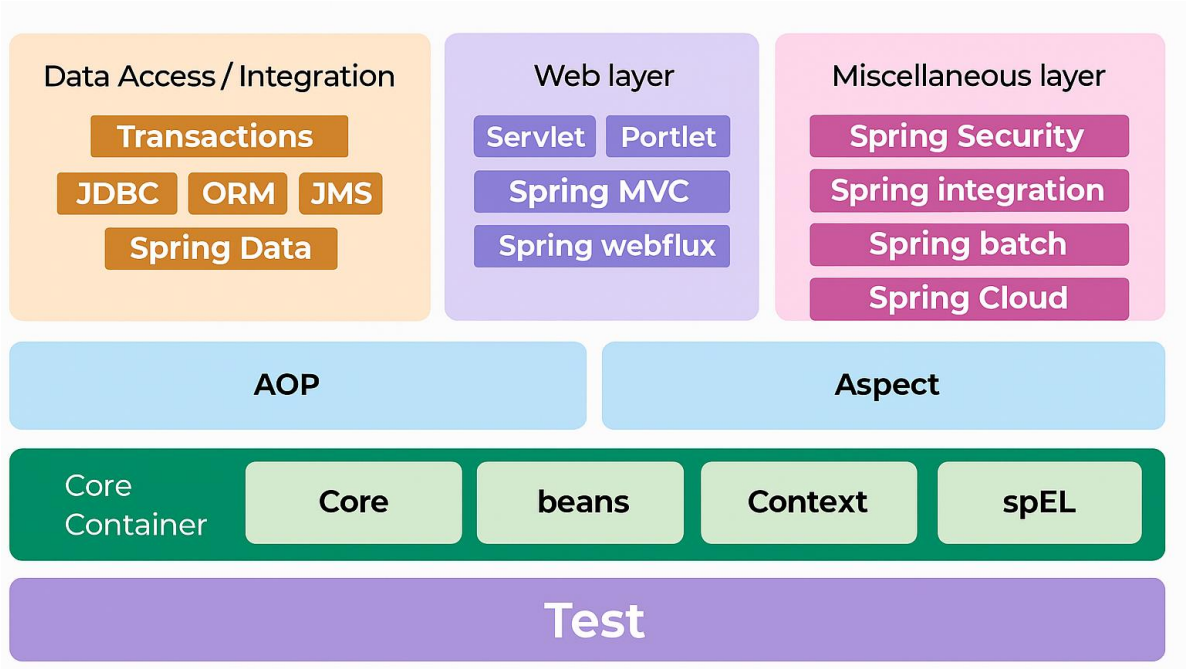


Fig. 5.2 Spring Framework Architecture.

Spring Security

Spring Security is a specialized module that provides a comprehensive security solution for enterprise applications. It manages authentication, authorization, and protection against vulnerabilities such as cross-site scripting and SQL injection. For example, you can easily restrict access to certain pages of a web application based on user roles with minimal configuration.

Spring Data

Spring Data simplifies the process of interacting with databases. It provides abstractions for working with both relational and non-relational databases, eliminating the need for writing repetitive data access code. For instance, developers can use interfaces like CrudRepository or JpaRepository to perform operations such as saving, deleting, or finding data without manually writing SQL queries.

Spring Batch

Spring Batch is designed for handling large-scale data processing tasks. It can efficiently read, process, and write bulk data such as financial transactions or inventory records. It is often used for automating scheduled jobs such as monthly report generation or daily data synchronization.

Integration with Other Frameworks

Spring integrates seamlessly with several other frameworks such as Hibernate, JPA, JMS, and Struts. This allows developers to build versatile applications by combining the strengths of multiple technologies within the same project.

Concepts of Spring Framework

Dependency Injection

Dependency Injection in Spring is a design technique used to achieve loose coupling and flexibility. It allows external sources to provide dependencies to a class instead of the class creating them internally. This makes the code more adaptable and easier to modify when requirements change.

There are three main types of dependency injection: constructor injection, setter injection, and field injection. Constructor injection provides dependencies when the object is created, setter injection provides them through a setter method, and field injection uses annotations to inject dependencies directly into class fields.

Inversion of Control (IoC) Container

Inversion of Control, also known as IoC, is a principle that delegates the responsibility of creating and managing objects to a container or framework rather than the application itself. In Spring, this responsibility is handled by the IoC container, which manages object creation and their dependencies.

Spring provides two main types of IoC containers: BeanFactory and ApplicationContext. BeanFactory is the simpler container that initializes beans only when needed. It is suitable for small or lightweight applications.

For example:

```
Resource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
Book book = (Book) factory.getBean("book");
```

In this example, the BeanFactory creates the Book bean only when it is requested. The ApplicationContext is a more advanced container that provides extra features such as event propagation, internationalization, and support for AOP. It is the preferred option in most enterprise applications.

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
Book book = (Book) context.getBean("book");
```

Spring Annotations

Spring Annotations provide metadata that defines how the framework should manage components and their dependencies. They allow configuration directly within the Java code, removing the need for extensive XML configuration.

Some commonly used annotations are @Component, which marks a class as a Spring-managed bean; @Autowired, which automatically injects dependencies into a class; @Bean, which explicitly defines a bean in a configuration class; and @Configuration, which specifies that a class contains bean definitions. These annotations make the configuration process easier and more readable.

Spring Modules

- The Spring Framework is divided into several modules that serve different purposes in application development. The Spring Core Module provides the basic functionality of the framework and includes the IoC container for managing beans and dependencies.

- The Spring AOP Module handles aspect-oriented programming, enabling developers to define reusable aspects for logging, transaction management, and performance monitoring.

- The Spring ORM Module facilitates database interactions by integrating with ORM frameworks such as Hibernate and JDO. It simplifies data access and transaction handling.

- The Spring Web MVC Module implements the model view controller architecture for building web-based applications. It manages user requests and responses through the DispatcherServlet and controller components.

- The Spring DAO Module provides a layer for accessing data through JDBC or other technologies while simplifying transaction management.
  The Spring Application Context Module extends the Core Module and provides additional features like validation, internationalization, event propagation, and resource loading.

- Finally, the Spring Web Flow Module helps in defining and managing the flow of user interactions across multiple pages or steps in a web application.

5.3 Spring MVC and Spring Boot

Spring MVC

Spring MVC implements the Model View Controller pattern for building web applications. The framework centers on a front controller named DispatcherServlet that receives every HTTP request, determines which controller should handle it, invokes that controller, and then selects and renders the appropriate view. In Spring MVC you mark controller classes with the @Controller annotation and you map incoming web requests to specific handler methods using @RequestMapping or the newer composed request mapping annotations.

Model view

The Model encapsulates the application data and generally consists of plain old Java objects. The View is responsible for rendering model data into a format the client can understand, typically HTML. The Controller handles incoming requests, invokes business logic or services, builds the model, and returns the logical name of the view to render.

DispatcherServlet and its responsibilities

DispatcherServlet is the core component in Spring MVC. It acts as the front controller for all HTTP requests. After receiving a request it finds the appropriate controller by consulting handler mappings. It then forwards the request to that controller. After the controller executes it returns a ModelAndView or a view name and model data. DispatcherServlet uses a view resolver to locate the actual view implementation and then passes the model data to the view for rendering.

Typical request processing flow

When a request arrives DispatcherServlet intercepts it and consults the configured handler mapping to identify the handler controller. The selected controller processes the request, typically by calling service layer methods and populating model attributes. The controller then returns a ModelAndView object or a logical view name. DispatcherServlet asks the view resolver to translate the logical view name into a concrete view, and the resolved view renders the model data into the HTTP response that is sent back to the client.

Spring MVC Flow

In the Spring MVC framework, the DispatcherServlet plays the central role in handling all incoming HTTP requests. When a request is received, the DispatcherServlet first intercepts it and consults the configured HandlerMapping to determine which controller should process the request. Once the appropriate controller is identified, the DispatcherServlet forwards the request to that controller.

The controller then executes the necessary business logic, interacts with the model layer if required, and prepares the data to be displayed. After processing, it returns a ModelAndView object containing both the model data and the logical view name.

Finally, the DispatcherServlet uses the configured ViewResolver to locate the corresponding view based on the logical view name. The view is then rendered with the model data, and the response is sent back to the client's browser.

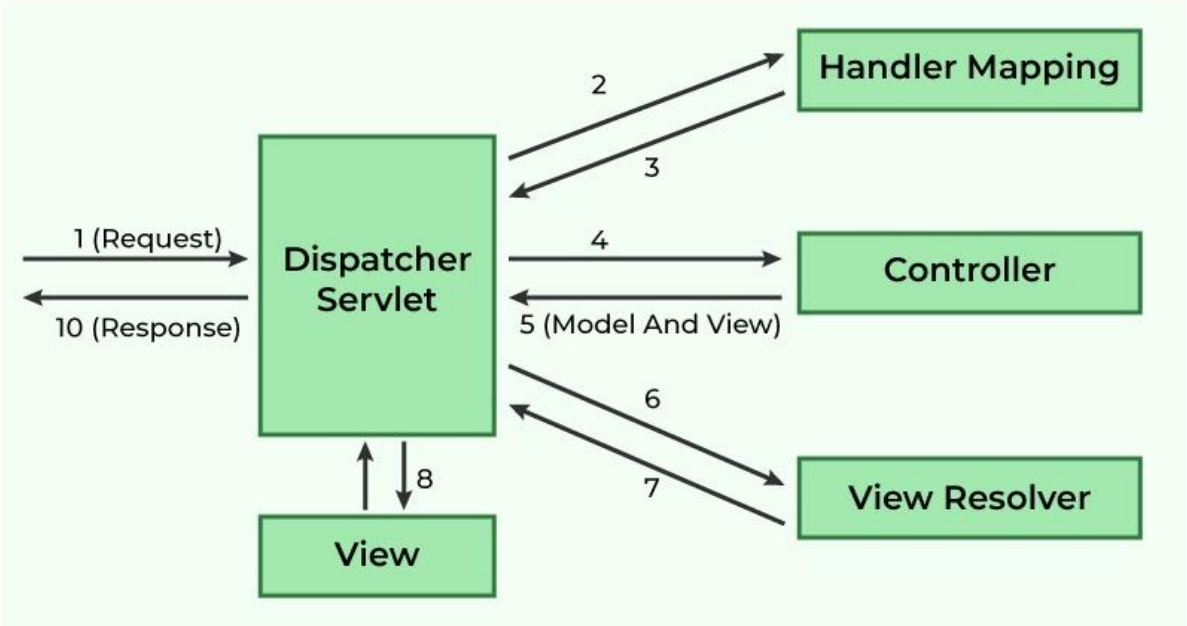


Fig. 5.3 Spring MVC Flow.

Components stored in WebApplicationContext

HandlerMapping, controllers, and view resolvers are part of the WebApplicationContext. WebApplicationContext extends the regular application context with web specific features such as request scope, session scope, and utilities for web applications.

How to create a Spring MVC application

Follow these steps to create a simple Spring MVC project.

Step 0 Set up the project structure

Create a Maven based web project and prepare the standard web application folders and a servlet container such as Tomcat.

Step 1 Add dependencies to pom.xml

Include Spring Web MVC and the servlet API. An example dependency configuration is shown below.

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.30</version>

</dependency>

<dependency>

<groupId>jakarta.servlet</groupId>

<artifactId>jakarta.servlet-api</artifactId>

<version>5.0.0</version>

<scope>provided</scope>

</dependency>

</dependencies>

Step 2 Define a controller

Create a Java class annotated with @Controller and map request paths to handler methods. For example:

@Controller

public class HelloGeek {

@RequestMapping("/")

public String display(Model model) {

model.addAttribute("message", "Spring MVC Tutorial!!");

```
return "index";

}

}
```

Step 3 Configure web.xml

Register DispatcherServlet in web.xml and map it to a URL pattern so it can handle incoming requests.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app_3_0.xsd"

version="3.0">

<display-name>SpringMVC</display-name>

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>
```

Step 4 Define servlet configuration

Place a configuration file named [servlet-name]-servlet.xml in WEB-INF to enable component scanning and configure view resolution. Example spring-servlet.xml:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<context:component-scan base-package="com.geeksforgeeks.controller"/>

<mvc:annotation-driven/>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/views/"/>

<property name="suffix" value=".jsp"/>

</bean>

</beans>

In this configuration context:component-scan tells Spring where to find annotated controllers and other components. The mvc:annotation-driven element enables annotation support for request mapping and form binding. The InternalResourceViewResolver maps logical view names to JSP files under /WEB-INF/views with a .jsp suffix.

Step 5 Create a JSP view

Create the JSP file that the view resolver will render. Example index.jsp in /WEB-INF/views/:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>

<!DOCTYPE html>

<html>

<head>

<title>Spring MVC Example</title>

</head>

<body>

<h2>${message}</h2>

</body>

</html>
```

Step 6 Run the application

Start your servlet container and open the application in a browser, for example http://localhost:8080/. The page will display the message set in the controller.

Alternative controller examples and explanations

You can map a controller at the class level and at the method level. Here is an example that maps the class to /hello and the method to handle GET requests.

```
@Controller

@RequestMapping("/hello")

public class HelloController {

@RequestMapping(method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");
```

```
return "hello";

}

}
```

You can combine value and method attributes on a single annotation like this.

```
@Controller

public class HelloController {

@RequestMapping(value = "/hello", method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}
```

A controller handler method usually performs business logic or calls services, populates the model with attributes that the view will read, and returns a logical view name. The view then renders those attributes.

View technologies

Spring supports many view types including JSP with JSTL, plain HTML, PDF generation, Excel output, JSON, XML, templates such as Velocity, XSLT, and reporting frameworks. JSP with JSTL is one of the most common choices for simple examples.

Example JSP for hello view located at /WEB-INF/hello/hello.jsp:

```
<html>

<head>

<title>Hello Spring MVC</title>

</head>

<body>
```

```
<h2>${message}</h2>

</body>

</html>
```

The expression ${message} refers to the model attribute added by the controller.

Required configuration details and customization

When DispatcherServlet is initialized it looks for a file named [servlet-name]-servlet.xml in WEB-INF by default. If you prefer a different location or name you can register a ContextLoaderListener and set the contextConfigLocation parameter in web.xml. The servlet mapping determines which URL patterns DispatcherServlet will process. For example mapping *.jsp will route requests ending with .jsp through that servlet.

Advantages of Spring MVC

Spring Web MVC enables clean separation of concerns by splitting input logic, business logic, and view logic. The framework supports rapid parallel development because designers and developers can work independently on views and controllers. The layered structure also makes debugging easier and it facilitates maintainability and updates in large projects.

Spring Boot

Spring Boot is an extension of the Spring Framework that simplifies the process of developing stand-alone, production-ready applications. It was introduced by the Spring team to eliminate the need for extensive configuration and boilerplate code that traditional Spring applications often required. With Spring Boot, developers can quickly set up and run applications with minimal setup, focusing more on the business logic rather than infrastructure and environment configuration.

The main goal of Spring Boot is to make Spring-based development easier, faster, and more efficient. It follows a “convention over configuration” approach, which means most configurations are automatically handled based on project dependencies and structure. This significantly reduces manual setup and allows developers to start developing applications almost immediately.

Need for Spring Boot

Before Spring Boot, creating a Spring application required defining multiple XML configuration files, manually setting up dependency injection, configuring view resolvers, and managing servlet containers. This process was time-consuming and complex, especially

for beginners. Developers had to configure frameworks like Hibernate, JPA, and security manually.

Spring Boot solved this problem by providing a pre-configured environment with sensible defaults. It automatically sets up the application based on the dependencies present in the classpath. For example, if Spring Boot detects a web dependency, it automatically configures an embedded server like Tomcat or Jetty, sets up a default dispatcher servlet, and prepares the application to handle web requests.

Core Features of Spring Boot

1. Auto Configuration

One of the most important features of Spring Boot is auto configuration. It automatically configures Spring applications based on the libraries and dependencies available in the classpath. For instance, if you include a database dependency, Spring Boot automatically sets up a DataSource and a connection pool without requiring any manual configuration. However, developers can override these default configurations if needed, providing flexibility and control.

2. Stand-Alone Applications

Spring Boot applications are stand-alone and self-contained. Unlike traditional Spring applications that require an external application server, Spring Boot includes embedded servers such as Tomcat, Jetty, or Undertow. This means developers can run their applications directly as Java programs without deploying them separately to a server. This feature simplifies the deployment process and makes testing and development faster.

3. Spring Boot Starters

Spring Boot provides a set of pre-defined starter dependencies known as “Spring Boot Starters.” These starters are convenient dependency descriptors that allow developers to include a set of commonly used dependencies for specific functionalities. For example, spring-boot-starter-web includes everything needed to build a web application, including Spring MVC, Tomcat, and JSON converters. This feature saves time and prevents version conflicts between different dependencies.

4. Spring Boot CLI (Command Line Interface)

Spring Boot comes with a Command Line Interface (CLI) that allows developers to run and test Spring applications quickly. It supports writing simple applications using Groovy scripts. This feature is particularly useful for prototyping, scripting, and testing ideas rapidly without setting up a complete Java project.

5. Spring Boot Actuator

Spring Boot Actuator provides built-in tools for monitoring and managing applications in production. It offers various endpoints that expose metrics such as application health, performance, and environment details. Actuator makes it easier to integrate with monitoring systems and manage the lifecycle of the application effectively. For example, endpoints like /actuator/health or /actuator/metrics provide real-time insights into the application's performance and state.

6. Spring Boot DevTools

Spring Boot DevTools enhances the developer experience by providing automatic restarts, live reloads, and improved productivity features. Whenever a developer makes changes in the code, the application restarts automatically, reflecting the latest updates without requiring a manual restart. This significantly reduces development time and improves efficiency.

Spring Boot Architecture

The architecture of Spring Boot is designed to simplify the traditional Spring framework. It is built on top of the Spring Framework and follows a layered architecture that includes several essential components.

At the core lies the Spring Framework, which provides dependency injection, AOP, and data access support. On top of this, Spring Boot adds auto configuration, which eliminates the need for manual XML configuration. The Spring Boot Starters act as dependency managers, automatically including all required libraries for specific functionalities.

The Embedded Server Layer allows applications to run independently without deploying to external servers. The Actuator Layer handles monitoring, metrics, and management tasks, while the Spring Boot CLI Layer supports fast testing and execution of applications.

How Spring Boot Works

When a Spring Boot application starts, it follows a specific startup process. The main class of the application is annotated with @SpringBootApplication, which is a combination of three important annotations @EnableAutoConfiguration, @ComponentScan, and @Configuration.

The @EnableAutoConfiguration annotation activates Spring Boot's auto-configuration mechanism. The @ComponentScan annotation instructs Spring to scan the specified package for components, configurations, and services, automatically registering them in the application context. The @Configuration annotation allows the class to define bean definitions and configuration settings.

During startup, Spring Boot checks for dependencies on the classpath, identifies the type of application being created (web, data, security, etc.), and configures the appropriate beans

automatically. It then starts the embedded server (if applicable) and makes the application ready to handle requests.

Advantages of Using Spring Boot

Spring Boot offers several advantages that make it ideal for modern software development. It simplifies configuration, reducing the need for manual setup. The use of embedded servers allows applications to be easily deployed and executed without external dependencies.

Its auto-configuration and starter dependency system save time and reduce errors. The inclusion of DevTools improves productivity, and Actuator enhances application monitoring and management. Furthermore, Spring Boot is compatible with cloud platforms such as AWS, Azure, and Google Cloud, making it ideal for microservices and cloud-native application development.

Example of a Simple Spring Boot Application

Below is a basic example of a Spring Boot application that displays a greeting message.

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

@RestController

public class HelloSpringBoot {

public static void main(String[] args) {

SpringApplication.run(HelloSpringBoot.class, args);

}

@GetMapping("/")

public String sayHello() {

return "Welcome to Spring Boot Application!";
```

```
}

}
```

In this example, the @SpringBootApplication annotation marks the main class as the entry point of the Spring Boot application. The @RestController annotation indicates that this class handles HTTP requests, and the @GetMapping annotation maps the root URL to the method sayHello(), which returns a simple greeting message.

5.4 Hibernate Framework

Hibernate is an open-source Object Relational Mapping (ORM) framework for Java that simplifies the interaction between Java applications and relational databases. It provides a bridge between Java objects and database tables, allowing developers to work with objects instead of writing complex SQL queries. Hibernate maps Java classes to database tables and Java data types to corresponding SQL data types, making data handling more intuitive and object-oriented.

By using Hibernate, developers can perform CRUD (Create, Read, Update, Delete) operations using its API or Hibernate Query Language (HQL) instead of traditional SQL. This approach significantly reduces boilerplate JDBC code, making development faster, cleaner, and easier to maintain.

Why Hibernate is Used

Traditional database interaction in Java is often handled through JDBC, which introduces several challenges. JDBC code is not easily portable across different database systems, making database switching difficult and time-consuming during a project's lifecycle. Additionally, JDBC requires explicit exception handling for every database operation, leading to verbose and repetitive code that can quickly become difficult to manage.

Another major limitation of JDBC is that it operates at the table level, offering no direct way to manage relationships between Java objects. Developers must manually write code to handle these associations, which adds to the complexity. Moreover, JDBC demands repetitive code for tasks like establishing connections, executing queries, and managing results, which reduces code readability and maintainability.

Hibernate was developed to overcome these challenges. It eliminates repetitive JDBC code, simplifies database operations, and provides advanced features such as object-level relationships, caching, and transaction management. In addition, Hibernate supports database independence, allowing applications to switch between different databases with minimal changes.

Key Features of Hibernate

The most notable feature of Hibernate is its ability to perform Object-Relational Mapping (ORM). It maps Java classes to database tables and class attributes to table columns, enabling developers to work with data using Java objects rather than SQL queries.

Hibernate also promotes database independence. Applications developed with Hibernate can run on multiple databases with little or no modification to the codebase, making them highly portable and flexible.

Another key feature is Hibernate Query Language (HQL), a powerful, object-oriented query language that allows developers to write queries independent of the underlying database.

Transaction management is seamlessly integrated in Hibernate, providing consistent and reliable handling of transactions through JDBC or Java Transaction API (JTA).

Hibernate also enhances performance through caching mechanisms. It supports first-level caching at the session level and optional second-level caching across sessions to reduce database access time.

Furthermore, Hibernate supports relationship mapping, allowing developers to define associations between objects such as one-to-one, one-to-many, many-to-one, and many-to-many relationships directly in Java classes.

Hibernate Architecture Overview

The architecture of Hibernate is modular and consists of several interrelated components that work together to manage database operations efficiently. Each layer in the architecture plays a specific role, from configuration and session management to transaction handling and query execution.

Main Components of Hibernate Architecture

1. Configuration

The Configuration class, found in the org.hibernate.cfg package, is the starting point of any Hibernate application. It is responsible for initializing and configuring the Hibernate framework. The configuration object reads the hibernate.cfg.xml file and mapping files that define the connection settings and mappings between Java classes and database tables.

Example:

```
Configuration cfg = new Configuration();

cfg.configure(); // Reads and validates hibernate.cfg.xml
```

This process activates the Hibernate framework, reads the configuration files, validates them, and generates in-memory metadata representing the configuration. If the configuration is invalid, Hibernate throws an exception.

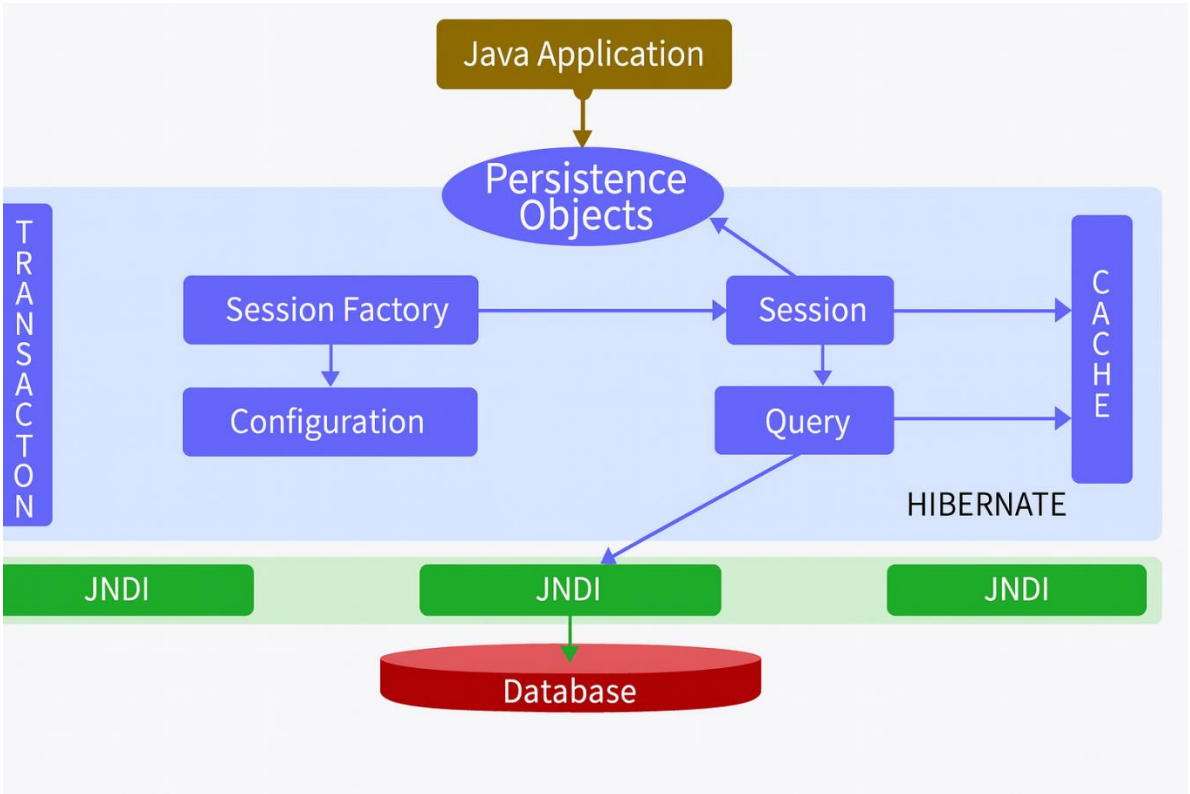


Fig. 5.4 Hibernate Architecture.

2. SessionFactory

The SessionFactory interface is responsible for creating and managing Session objects. It is a heavyweight and thread-safe object that should be created once per database and shared across the application. The SessionFactory also manages the second-level cache, improving performance across multiple sessions.

Example:

```
SessionFactory factory = cfg.buildSessionFactory();
```

3. Session

The Session interface represents a single unit of work between the application and the database. It provides methods for performing CRUD operations and manages the first-level

cache, which holds objects within the session scope. A new session is typically opened from the SessionFactory for each transaction.

Example:

Session session = factory.openSession();

4. Transaction

The Transaction interface ensures atomic operations within the database, maintaining data integrity by providing mechanisms for commit and rollback. It ensures that either all database operations within a transaction are executed successfully or none are applied in case of an error.

Example:

Transaction tx = session.beginTransaction();

tx.commit(); **// or tx.rollback();**

5. Query

The Query interface allows developers to perform database operations using HQL, Criteria API, or native SQL queries. It is used to fetch, update, or delete records from the database in an object-oriented manner.

Example:

Query<Student> query = session.createQuery("from Student");

List<Student> students = query.list();

6. Persistent Classes (Entities)

Persistent classes represent the actual data model of the application. Each class corresponds to a database table, and its fields correspond to table columns. These classes are annotated with @Entity and must have a primary key annotated with @Id.

Example:

@Entity

class Student {

@Id

private Long id;

private String name;

}

Each object of a persistent class represents a single record (row) in the corresponding database table.

7. Mapping

Mapping defines the relationship between Java classes and database tables. It specifies how class attributes correspond to table columns and how relationships between entities are structured. This can be achieved through annotations or XML mapping files.

Example:

@Entity

@Table(name="student")

class Student {

@Id

private int id;

private String name;

}

Hibernate supports several types of relationships, such as one-to-one, one-to-many, many-to-one, and many-to-many.

8. JDBC Layer

Although Hibernate abstracts the underlying database interactions, it still uses JDBC internally to execute SQL queries and retrieve results. The JDBC layer manages connections, executes generated SQL statements, and returns data to Hibernate for conversion into Java objects.

This layer handles connection pooling, result fetching, and communication with the database, ensuring smooth operation between the ORM and the relational data source.

Hibernate Workflow

The typical workflow of a Hibernate application follows a sequence of steps:

- The application first loads the Hibernate configuration file (hibernate.cfg.xml) which contains the database connection settings and mapping details.
- Hibernate uses this configuration to create a SessionFactory.
- The application requests a Session from the SessionFactory.
- Within the session, a transaction is initiated. CRUD operations are performed using Hibernate APIs or HQL queries.
- The transaction is then either committed or rolled back depending on the outcome.
- Hibernate translates the object-oriented operations into SQL commands and executes them through JDBC.
- The results are retrieved and mapped back into Java objects that can be used by the application.
- Through this process, Hibernate provides a seamless, object-oriented way to interact with relational databases while minimizing manual coding and configuration.

Other Popular Frameworks

Java has remained one of the most powerful and versatile programming languages for enterprise, desktop, and web applications. Over time, developers have built frameworks on top of Java to simplify complex tasks such as web development, configuration management, and server handling. These frameworks provide a foundation for developing scalable, maintainable, and efficient applications by minimizing repetitive coding and promoting reusability.

Apart from the Spring Framework, there are several other popular frameworks that have significantly influenced Java development. Some of the most widely used among them include Apache Struts, JavaServer Faces (JSF), Play Framework, Grails Framework, and Vaadin Framework. Each of these frameworks offers unique features, architectural patterns, and benefits depending on the nature of the application being developed.

Apache Struts Framework

The Apache Struts Framework is one of the earliest and most widely used open-source Java web frameworks developed by the Apache Software Foundation. It is based on the Model-View-Controller (MVC) architecture, which separates the application logic from the user interface, promoting better maintainability and scalability. Struts gained popularity for its organized structure and ability to handle large, enterprise-level web applications efficiently.

Working Principle

In a Struts application, the controller is represented by the ActionServlet which intercepts client requests. It reads the configuration file (struts-config.xml) to identify the correct Action class to handle the request. The Action class processes business logic and interacts with the model layer to retrieve or update data. Finally, it forwards the results to the view layer, typically implemented using JSP (JavaServer Pages), to display the output to the user.

Key Features and Advantages

Struts provides a robust tag library for managing forms, validation, and data presentation. It supports internationalization and offers built-in mechanisms for exception handling. The framework promotes reusable components and helps developers create applications that are easy to maintain and expand. Although Struts has been succeeded by modern frameworks like Spring MVC, it remains relevant in many legacy enterprise systems and continues to serve as an excellent foundation for understanding MVC-based development in Java.

JavaServer Faces (JSF)

JavaServer Faces (JSF) is an official Java EE framework developed by Oracle. It simplifies the process of developing server-side user interfaces for web applications. JSF is a component-based framework, meaning developers can use pre-built UI components instead of writing HTML, CSS, and JavaScript manually. This approach allows developers to focus on application logic while the framework manages rendering and event handling.

Architecture and Functionality

In JSF, applications are built around UI components that correspond to elements on a web page, such as buttons, forms, and input fields. These components are connected to managed beans, which handle user input and execute business logic. JSF operates through a well-defined request-processing lifecycle that includes restoring the view, applying request values, processing validations, updating model values, invoking application logic, and rendering the final response.

Developers can define page navigation, event handling, and validations through configuration files or annotations. JSF also integrates seamlessly with other Java EE technologies like Enterprise JavaBeans (EJB) and Java Persistence API (JPA), making it suitable for large-scale enterprise systems.

Advantages of JSF

JSF promotes reusable UI components, simplifies data binding, and reduces the need for boilerplate code. Its strong integration with Java EE provides built-in support for dependency injection, validation, and internationalization. However, JSF can be complex for beginners

due to its multiple configuration layers. Despite that, it remains a stable choice for enterprise environments that prioritize maintainability and long-term support.

Play Framework

The Play Framework is a modern, high-performance framework for developing scalable web applications using Java and Scala. It was designed to address the limitations of traditional Java web frameworks by emphasizing simplicity, developer productivity, and scalability. Play follows the Reactive Model and promotes a stateless and asynchronous architecture, making it suitable for handling concurrent requests efficiently.

Architecture and Features

Play is built on top of the Akka toolkit, which supports reactive programming and concurrency management. It allows developers to build applications that are non-blocking, event-driven, and responsive. The framework uses the principle of convention over configuration, which eliminates the need for complex XML configuration files.

Developers can write RESTful APIs easily using Play's routing system. It also includes features such as hot reloading, which allows developers to see code changes instantly without restarting the server. Additionally, Play provides integration with JSON processing, form validation, testing tools, and template engines like Twirl for dynamic HTML rendering.

Why Play Framework is Popular

Play is particularly suited for cloud-based, distributed, and high-performance systems such as social networking platforms, chat servers, and streaming services. Its scalability, simplicity, and native support for microservices make it a preferred choice among developers building modern web applications that demand real-time performance and responsiveness.

Grails Framework

The Grails Framework is an open-source web application framework that uses the Groovy programming language, which runs on the Java Virtual Machine (JVM). It combines the power of Java with the simplicity and flexibility of Groovy, allowing developers to write concise and expressive code. Grails builds upon the foundations of Spring Boot, Spring MVC, and Hibernate, providing a cohesive environment for rapid application development.

Core Concepts and Functionality

Grails follows the convention over configuration principle, meaning that it automatically identifies and configures application components without requiring manual setup. It includes a feature called scaffolding, which can automatically

5.5 Maven

Maven is a powerful **build automation and project management tool** developed in Java and primarily used for **Java-based applications**. It simplifies the process of building, testing, and deploying Java projects by providing a standard way to manage dependencies, compile source code, run tests, and generate project reports.

At its core, Maven follows the concept of the **Project Object Model (POM)**, which is defined in an XML file named pom.xml. This file contains all the essential information about the project such as dependencies, plugins, build configurations, and project structure. Maven reads this file and automatically performs the necessary build operations.

The primary goal of Maven is to **standardize the build process** and make project management consistent across multiple development teams. By using a declarative approach rather than a procedural one, Maven allows developers to describe what needs to be done, while Maven handles how it is done.

Core Concepts of Maven

1. Project Object Model (POM)

The **POM (Project Object Model)** is the heart of every Maven project. It is an XML file named pom.xml that resides in the project's root directory. This file defines the project structure, configuration details, version information, and the external libraries (dependencies) that the project requires.

Each POM file includes important details such as the project's **groupId**, **artifactId**, and **version**, which together uniquely identify the project across repositories. For example, groupId represents the organization or company, artifactId represents the name of the project, and version indicates the release version of the artifact.

The POM file also contains configuration information for **plugins**, **repositories**, and **build profiles**. When Maven runs a build command, it reads the POM file and executes tasks according to the defined lifecycle and configuration.

2. Dependency Management

One of the most powerful features of Maven is **dependency management**. In traditional Java projects, developers had to manually download JAR files, configure classpaths, and manage library versions. Maven automates this entire process by allowing developers to declare required dependencies in the POM file.

Maven then connects to an online **central repository** (or custom repositories) to automatically download the required libraries and add them to the project's classpath. This

ensures consistency and eliminates compatibility issues between different versions of libraries.

For instance, if a project depends on Spring Framework, developers only need to declare it in the POM file, and Maven will automatically fetch the required JARs and all its transitive dependencies. This not only saves time but also ensures that the project remains up-to-date and compatible with future releases.

3. Standard Project Structure

Maven enforces a **standard directory structure** across all projects. This uniform structure helps developers understand and navigate any Maven project easily, even if they have never worked on it before.

The standard Maven project layout includes the following main directories:

- src/main/java – Contains the main source code of the project.
- src/main/resources – Stores configuration files and resource files used by the application.
- src/main/webapp – Used for web application resources like HTML, JSP, and CSS files.
- src/test/java – Contains the unit test source code.
- src/test/resources – Holds resource files required for testing.
- target – Stores compiled class files, generated reports, and packaged JAR or WAR files.
- pom.xml – The POM file defining the project's dependencies and configurations.

This standardization ensures consistency across teams and reduces setup errors, making project management more efficient and predictable.

4. Maven Build Lifecycle

The **Maven build lifecycle** defines the sequence of steps or phases involved in building a project. Maven provides three default lifecycles: **clean**, **default (build)**, and **site**, each of which contains a set of predefined phases. The default lifecycle is the most commonly used and consists of phases like compile, test, package, and deploy.

Here's how the main phases work in a Maven build process:

- **Validate:** Checks if the project structure and configuration are correct before proceeding.
- **Compile:** Compiles the Java source code from the src/main/java directory into .class files stored in the target/classes folder.
- **Test:** Executes unit tests located in src/test/java using frameworks like JUnit or TestNG.

- **Package:** Packages the compiled code into a distributable format such as JAR or WAR.
- **Integration Test:** Executes integration tests to ensure different modules work together correctly.
- **Verify:** Validates the results of tests and ensures the project meets quality standards.
- **Install:** Installs the packaged code (JAR/WAR) into the local Maven repository, making it available for use in other local projects.
- **Deploy:** Uploads the packaged code to a remote repository, allowing other developers or teams to access it.

This lifecycle ensures that every build process is consistent, automated, and reproducible.

5. Maven Plugins

Maven relies heavily on **plugins** to perform various tasks during the build process. Plugins are reusable components that extend Maven's functionality. Each phase of the build lifecycle is typically bound to a specific plugin.

For example, the **compiler plugin** handles code compilation, the **surefire plugin** runs unit tests, and the **jar plugin** creates JAR files. Developers can also add custom plugins for activities such as code generation, deployment, and documentation.

Plugins make Maven extremely flexible because they allow developers to integrate other tools like JUnit, Checkstyle, and Docker seamlessly into the build process.

6. Maven Repositories

Maven uses repositories to store and retrieve project dependencies and plugins. There are three main types of repositories: **local**, **central**, and **remote**.

The **local repository** is a directory on the developer's machine where Maven stores downloaded dependencies. The **central repository** is a public online repository maintained by the Maven community, which contains thousands of libraries and frameworks. The **remote repository** can be a custom repository hosted by an organization for internal use.

When Maven builds a project, it first checks the local repository for dependencies. If they are not available, Maven downloads them from the central or remote repository and stores them locally for future use.

7. Build Profiles

Maven allows developers to define **build profiles** in the POM file. A build profile is a set of configurations that enable different build environments such as development, testing, and production.

For instance, a developer can create a development profile that includes debugging libraries and test data, and a production profile that enables optimizations and excludes unnecessary dependencies. This feature allows teams to maintain different configurations in a single project without modifying code for different environments.

8. Maven Integration with IDEs

Maven integrates seamlessly with popular Integrated Development Environments (IDEs) such as **Eclipse**, **IntelliJ IDEA**, **Spring Tool Suite (STS)**, and **NetBeans**. These IDEs provide built-in Maven support that automatically imports the POM file, manages dependencies, and executes build commands directly from the interface.

This integration enhances productivity by allowing developers to compile, run, test, and deploy applications without switching between tools. IDE integration also ensures that the Maven configuration remains consistent across different development environments.

Advantages of Using Maven

Maven offers several advantages that make it a preferred choice for developers and organizations. It simplifies **dependency management** by automatically handling library downloads and version conflicts. Its **standard project structure** ensures consistency across teams, and the **build lifecycle** automates repetitive tasks like compiling, testing, and packaging.

Maven also promotes **reusability** by allowing developers to share artifacts through repositories. The tool supports **automated testing**, enabling continuous integration with platforms like Jenkins. In addition, Maven integrates effortlessly with IDEs and other development tools, enhancing overall efficiency and team collaboration.

Maven Installation

Before using Maven to build or manage Java projects, it must be installed and properly configured on your system. Maven is a Java-based tool, which means it requires Java to be installed first. The installation process involves downloading Maven, setting up environment variables, and verifying that it is correctly configured. Once installed, Maven can be used from the command line or integrated with IDEs such as Eclipse, IntelliJ IDEA, or NetBeans.

Prerequisites for Maven Installation

Java Development Kit (JDK) Installation

Since Maven is developed in Java, it needs the Java Development Kit (JDK) to compile and run. Before installing Maven, ensure that the latest version of JDK is installed on your computer. Maven requires at least JDK 8 or higher, though newer versions like JDK 11 or 17

are recommended for better compatibility.

To check if Java is installed, open the command prompt (Windows) or terminal (macOS/Linux) and type:

java -version

If Java is installed, it will display the version number. For example:

java version "17.0.1" 2021-10-19 LTS
Java(TM) SE Runtime Environment (build 17.0.1+12-LTS-39)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)

If Java is not installed, download and install it from the official Oracle website or OpenJDK distribution. After installation, set the JAVA_HOME environment variable to point to your JDK installation directory.

Downloading Apache Maven

The next step is to download Maven from the official Apache Maven website. The official URL is: https://maven.apache.org/download.cgi

On the download page, you will find different versions of Maven available. It is recommended to download the binary zip archive for Windows or the tar.gz file for macOS and Linux.

For example, the downloaded file might be named something like:
apache-maven-3.9.9-bin.zip (for Windows) or
apache-maven-3.9.9-bin.tar.gz (for Linux/Mac).

Extracting Maven Files

Once the file is downloaded, extract the archive to a directory of your choice.

For example:

On Windows, extract it to: C:\Program Files\Apache\Maven\apache-maven-3.9.9

On Linux/Mac, extract it to: /usr/local/apache-maven/apache-maven-3.9.9

After extraction, note the path where Maven is installed, because this path will be required while setting environment variables.

Inside the extracted Maven folder, you will find the following directories:

bin – contains the Maven executable files.

boot – contains bootstrap JARs used internally by Maven.

conf – contains the settings.xml file used for configuration.

lib – contains all the libraries required by Maven to function.

Configuring Environment Variables

To use Maven from the command line, it must be added to the system's environment variables.

For Windows Users:

Right-click on This PC or My Computer and choose Properties.

Click on Advanced system settings and then Environment Variables.

Under System variables, click New and add a new variable:

Variable name: MAVEN_HOME

Variable value: Path to your Maven installation folder (for example: C:\Program Files\Apache\Maven\apache-maven-3.9.9)

Now, edit the Path variable in system variables and add Maven's bin directory to it:

Add: %MAVEN_HOME%\bin

Also, make sure that the JAVA_HOME variable is already defined and points to your JDK installation directory, for example:

JAVA_HOME = C:\Program Files\Java\jdk-17

Click OK to save all the settings.

For macOS and Linux Users:

Open the terminal and edit the bash profile or zsh configuration file depending on the shell

you are using.

Use a text editor such as nano or vim to open the configuration file:

nano ~/.bashrc

or

nano ~/.zshrc

Add the following lines to the end of the file:

```
export M2_HOME=/usr/local/apache-maven/apache-maven-3.9.9
export MAVEN_HOME=/usr/local/apache-maven/apache-maven-3.9.9
export PATH=$M2_HOME/bin:$PATH
```

Save the file and run the following command to reload the configuration:

source ~/.bashrc

This makes Maven accessible from the command line globally.

Verifying Maven Installation

After setting up environment variables, verify that Maven has been successfully installed. Open the terminal or command prompt and type the following command:

mvn -version

If Maven is installed correctly, you will see output similar to this:

```
Apache Maven 3.9.9 (e1a9bdb7b9f450a9a3f09c6b414b450bfa8a1a49)
Maven home: C:\Program Files\Apache\Maven\apache-maven-3.9.9
Java version: 17.0.1, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk-17
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64"
```

This output confirms that Maven is correctly installed and can communicate with Java.

Maven Local Repository

Once Maven is installed, it automatically creates a local repository on your system the first

time you build a project. This local repository is where Maven stores all downloaded libraries, dependencies, and plugins from remote repositories.

The default location of the local repository is:

Windows: C:\Users\<username>\.m2\repository

Linux/Mac: /home/<username>/.m2/repository

You can customize this location by editing the settings.xml file located in the Maven conf directory.

This local repository allows Maven to reuse downloaded dependencies, which improves build performance and reduces network usage for future projects.

Integrating Maven with IDEs

After installation, Maven can be easily integrated with Integrated Development Environments (IDEs) such as Eclipse, IntelliJ IDEA, or NetBeans.

In Eclipse, for example, Maven support is built-in through the "Maven Integration for Eclipse" plugin. You can create a Maven project directly by navigating to:
File → New → Maven Project

Similarly, IntelliJ IDEA automatically detects Maven projects by scanning the pom.xml file and imports all dependencies. IDE integration allows developers to run Maven goals (such as clean, compile, test, or package) directly from the graphical interface, making it easier to manage builds.

Troubleshooting Common Maven Installation Issues

Sometimes, issues may occur during installation or configuration. Common problems include:

Maven command not recognized: This usually means the PATH variable is not set correctly. Ensure %MAVEN_HOME%\bin (Windows) or $M2_HOME/bin (Linux/Mac) is added to your system PATH.

Java not found: Maven requires Java to run. Verify that JAVA_HOME is set correctly and that the java -version command works.

Permission issues: On Linux or macOS, ensure you have the proper permissions to install Maven and modify system variables. You may need to use sudo while extracting files or editing configuration files.

Fixing these issues usually resolves Maven setup problems and ensures smooth operation.

Updating Maven

Over time, newer versions of Maven may be released with bug fixes and performance improvements. To update Maven, simply download the latest version from the official website, extract it to a new directory, and update the MAVEN_HOME environment variable to point to the new version.

There is no need to uninstall the previous version, as Maven is self-contained and does not modify the system registry.

Basic Structure of the POM File

A POM (Project Object Model) file pom.xml is the fundamental configuration file for a Maven project. It describes the project, its dependencies, build instructions, plugins, and metadata.

Workflow

Here is the workflow of a Maven POM file in any Maven project:

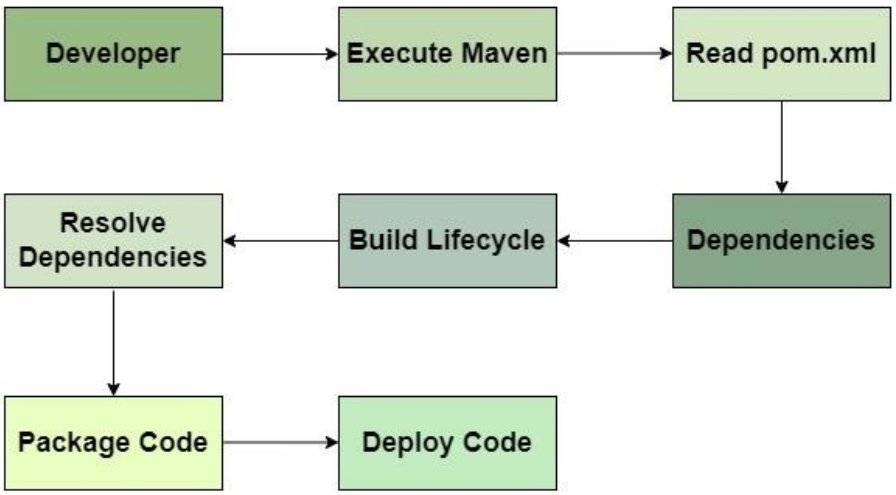


Fig. 5.5 Workflow of a Maven POM.

1. **Initialization:** The Maven reads the pom.xml file and initializes the build process.
2. **Dependency Resolution:** Downloads the Specified dependencies from remote repositories.
3. **Build Life cycle Execution:** Executes the build life cycle phases like compile, test, package, verify and other phases.
4. **Plugin Execution:** Runs the configured plugins for various takes like code analysis and other tasks.
5. **Packaging:** Packages the compiled code into specified format like JAR, WAR and other formats.
6. **Deployment:** Deploys the packaged code to a remote repository or serve

Top-level <project> element

Every POM is an XML document whose root is <project>. It must declare the Maven POM model version and usually the XML namespace. The root groups all other configuration elements.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  ...
</project>
```

Coordinates: groupId, artifactId, version

These three elements identify the artifact produced by the project. groupId is the organization or package namespace (e.g., com.example). artifactId is the project name (the resulting jar/war name). version is the artifact version. Together they uniquely identify an artifact in repositories.

```
<groupId>com.example</groupId>
<artifactId>my-app</artifactId>
<version>1.0.0-SNAPSHOT</version>
```

packaging

Specifies the packaging type Maven will build: jar (default), war, pom, ear, etc. When omitted, Maven assumes jar.

```
<packaging>jar</packaging>
```

Project metadata: name, description, url, inceptionYear

Human-friendly metadata that helps consumers of the POM learn what the project is about. Not used directly by the build, but useful for generated sites and reports.

```
<name>My Application</name>
<description>A short description of my application</description>
<url>https://example.com/my-app</url>
<inceptionYear>2024</inceptionYear>
```

parent

A project can inherit configuration from a parent POM. Using a parent is common in multi-module projects and when using an organization-wide parent (like Spring Boot's parent). Parent coordinates plus an optional relative path allow inheritance of dependencyManagement, plugins, properties, and more.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.1.0</version>
</parent>
```

modules (multi-module projects)

When a POM has <packaging>pom</packaging> and manages multiple sub-modules, you list them under <modules>. Each module is a directory with its own pom.xml and typically inherits from the parent.

```
<packaging>pom</packaging>
<modules>
  <module>module-a</module>
  <module>module-b</module>
</modules>
```

properties

A convenient place to store named values (like plugin or Java versions) that can be referenced with ${...} elsewhere in the POM. Centralized properties make upgrades and consistency easier.

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
```

```
</properties>
```

dependencies and <dependency>

Dependencies declare external libraries your project needs. Each dependency has groupId, artifactId, version, and optional scope (compile, provided, runtime, test, system, import). scope controls when the dependency is available (compile/test/runtime) and whether it's transitive.

```
<dependencies>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.13.0</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

dependencyManagement

Used primarily in parent POMs or aggregator POMs to centrally define versions and other dependency settings without forcing immediate inclusion. Child POMs can declare a dependency with only groupId and artifactId and inherit the version from dependencyManagement.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.google.guava</groupId>
      <artifactId>guava</artifactId>
      <version>31.1-jre</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

repositories and pluginRepositories

By default Maven uses Maven Central. If you need artifacts from other servers (private Nexus/Artifactory, or other public repos), add them here. pluginRepositories is the same idea but for Maven plugins.

```
<repositories>
  <repository>
    <id>my-nexus</id>
    <url>https://nexus.example.com/repository/maven-public/</url>
  </repository>
</repositories>
```

build - plugins, resources, finalName

The <build> element controls the build lifecycle. It contains <plugins> (plugin declarations and configuration), <resources> (non-source files to include), <finalName> (name of the built artifact), and other options (directory names, default goal).

```
<build>
  <finalName>${project.artifactId}-${project.version}</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <source>${maven.compiler.source}</source>
        <target>${maven.compiler.target}</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

profiles

Profiles allow conditional configuration (different dependencies, plugin settings, properties) that are activated via command-line (-P), environment variables, JDK version, OS, files present, etc. Useful for building different flavours (dev vs prod) or enabling platform-specific behavior.

```
<profiles>
  <profile>
```

```
    <id>production</id>
    <activation>
      <property>
        <name>env</name>
        <value>prod</value>
      </property>
    </activation>
    <properties>
      <app.env>production</app.env>
    </properties>
  </profile>
</profiles>
```

scm, developers, licenses, issueManagement

Optional but valuable metadata used by site generation and by tools that integrate with source control and issue trackers.

```
<scm>
  <connection>scm:git:git@example.com:myorg/myapp.git</connection>

<developerConnection>scm:git:ssh://git@example.com/myorg/myapp.git</developerConne
ction>
  <url>https://example.com/myorg/myapp</url>
</scm>

<developers>
  <developer>
    <id>alice</id>
    <name>Alice Example</name>
    <email>alice@example.com</email>
  </developer>
</developers>

<licenses>
  <license>
    <name>Apache License, Version 2.0</name>
    <url>https://www.apache.org/licenses/LICENSE-2.0.txt</url>
  </license>
</licenses>
```

distributionManagement

Configures where built artifacts should be deployed (release repository, snapshot repository) and can include site deployment settings. This is typically used in CI/CD to mvn deploy.

```
<distributionManagement>
  <repository>
    <id>releases</id>
    <url>https://repo.example.com/releases</url>
  </repository>
  <snapshotRepository>
    <id>snapshots</id>
    <url>https://repo.example.com/snapshots</url>
  </snapshotRepository>
</distributionManagement>
```

reporting

Define settings for reports generated by the Maven site plugin (Javadoc, surefire reports, code coverage integrations).

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.5.0</version>
    </plugin>
  </plugins>
</reporting>
```

Effective POM and inheritance/overrides

Maven calculates an effective POM by combining the project POM, parent POM(s), settings, and active profiles. Child POMs override parent values; many sections (like dependencyManagement and plugins in build) are merged according to Maven's rules. Use mvn help:effective-pom to see the final resolved POM.

Example POM (complete)

A compact but complete example showing the main elements:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>demo-app</artifactId>
  <version>0.1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Demo App</name>
  <description>Example project POM</description>
  <url>https://example.com/demo-app</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.13.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <finalName>${project.artifactId}-${project.version}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
```

```
      <version>3.11.0</version>
      <configuration>
        <source>${maven.compiler.source}</source>
        <target>${maven.compiler.target}</target>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

Maven Build Lifecycle

The **Maven Build Lifecycle** is a fundamental concept that defines the structured process Maven follows to build, test, and deploy a project. It consists of a predefined sequence of **phases** and **goals**, each responsible for a specific task in the build process. From source code compilation to deployment, Maven ensures consistency, automation, and standardization across projects.

Maven provides **three primary lifecycles Default**, **Clean**, and **Site** each designed for different aspects of project management. Executing a particular phase in any lifecycle automatically runs all preceding phases in order.

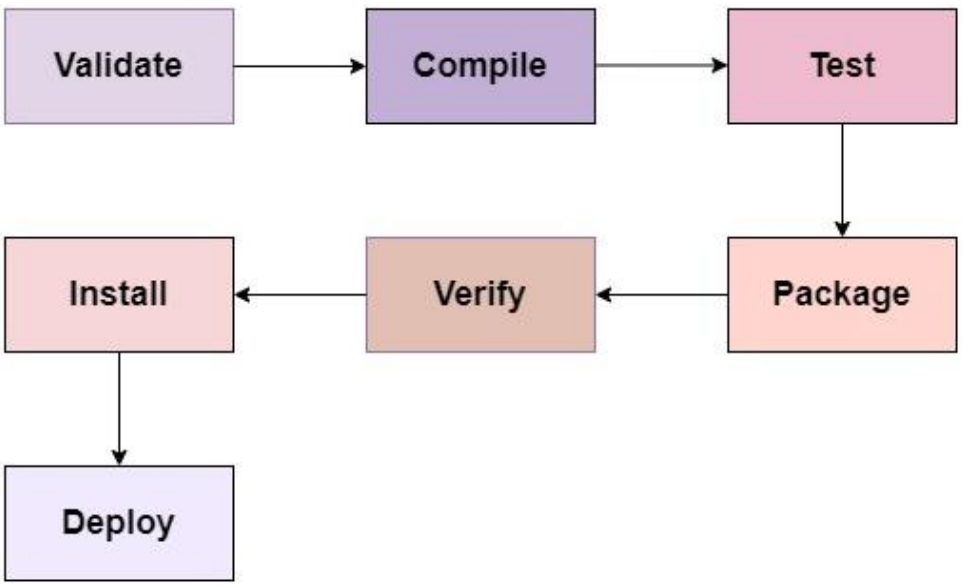


Fig. 5.6 Maven Build Lifecycle.

1. Default Lifecycle

The **Default Lifecycle** manages the full build and deployment process of a project. It handles everything from validating the configuration to generating the final deployable artifact (such as a JAR or WAR file).

Phases of the Default Lifecycle

- **Validate**: Ensures that all project information is correct and configuration files are valid.
- **Compile**: Compiles the project's source code into bytecode.
- **Test**: Executes unit tests using testing frameworks like JUnit to validate code correctness.
- **Package**: Packages the compiled code into a distributable format (JAR/WAR).
- **Verify**: Performs additional checks to ensure that the package is complete and consistent.
- **Install**: Installs the artifact into the local Maven repository so it can be reused in other projects locally.
- **Deploy**: Publishes the final artifact to a remote repository for sharing with other developers or systems.

Example – Building a Web Application

Consider a Maven-based **web application** project named StudentPortal. The project structure includes dependencies for **Spring Boot**, **Thymeleaf**, and **MySQL**.

Command Workflow:

```
mvn validate    # Checks configuration files and dependencies
mvn compile     # Compiles Java source files
mvn test        # Runs unit tests
mvn package     # Packages into a WAR file
mvn install     # Installs in local Maven repository
mvn deploy      # Deploys to remote repository
```

When these commands are executed sequentially, Maven automates each step from validation to deployment ensuring a smooth and reproducible build process.

2. Clean Lifecycle

The **Clean Lifecycle** ensures a fresh environment for every build by deleting previously generated build files and temporary data. This prevents conflicts and outdated artifacts from affecting new builds.

Phases of the Clean Lifecycle

- **pre-clean**: Executes any pre-cleanup tasks before the main cleaning process.
- **clean**: Removes all files generated by previous builds, typically from the target/ directory.
- **post-clean**: Performs additional tasks or cleanup actions after the main cleaning operation.

Example – Cleaning a Project

Before running a new build for the StudentPortal project, use:

mvn clean

This command removes the entire target folder and ensures that no residual files interfere with the next build. It is commonly used before packaging or deploying updates.

3. Site Lifecycle

The **Site Lifecycle** is designed for generating and publishing project documentation, such as reports, dependency listings, and Javadoc pages. This lifecycle helps maintain transparency and aids in project management.

Phases of the Site Lifecycle

- **pre-site**: Executes actions required before generating site documentation.
- **site**: Generates project documentation using reporting plugins.
- **post-site**: Executes tasks after documentation generation (e.g., final formatting).
- **site-deploy**: Publishes the generated documentation to a remote web server.

Example – Generating Documentation

To create a project report for StudentPortal, run:

mvn site

This command generates documentation in the target/site directory, which can include dependency reports, plugin usage, and code coverage statistics. To upload it to a documentation server, use:

mvn site-deploy

Example Project: StudentPortal

Below is an example **POM file** for the StudentPortal web application project, demonstrating dependencies and build configurations aligned with Maven lifecycles.

```
<project xmlns="https://maven.apache.org/POM/4.0.0"
         xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
                             https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.portal</groupId>
    <artifactId>studentportal</artifactId>
    <version>1.0.0</version>
    <packaging>war</packaging>
    <name>Student Portal</name>

    <properties>
        <java.version>17</java.version>
    </properties>

    <dependencies>
        <!-- Spring Boot Web Starter -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Thymeleaf Template Engine -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>

        <!-- MySQL Driver -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>

        <!-- JUnit Test Dependency -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
```

```
    </dependencies>

    <build>
        <plugins>
            <!-- Compiler Plugin -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>17</source>
                    <target>17</target>
                </configuration>
            </plugin>

            <!-- Spring Boot Plugin -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Phases and Goals

Maven is a powerful build automation tool that follows a **lifecycle-based approach** for managing a project's build process. Two fundamental concepts that form the backbone of this lifecycle are **Phases** and **Goals**. Understanding these concepts helps developers control the flow of the build and customize it as per project requirements.

Maven Phase

A **Maven Phase** represents a **stage in the Maven build lifecycle**. Each phase performs a specific task that contributes to the overall build process of a project. Maven's default build lifecycle consists of several predefined phases that are executed in a specific order from validating project structure to deploying the final package.

Phases are **executed sequentially**, meaning that running a specific phase will also execute all the phases that come before it. For instance, if we execute the mvn deploy command, Maven will run all the phases starting from validate up to deploy.

Common Maven Phases in the Default Lifecycle

1. **validate:** Ensures that all required information for the build is available and that the project structure is correct.
2. **compile:** Compiles the project's main source code located in the src/main/java directory.
3. **test-compile:** Compiles the test source code located in src/test/java, preparing it for testing.
4. **test:** Executes unit tests using a testing framework like JUnit or TestNG. These tests verify that individual components of the application work as expected.
5. **package:** Packages the compiled code into a distributable format such as a JAR or WAR file, depending on the project type.
6. **integration-test:** Deploys the packaged code to an environment where integration tests can be run to verify system-level interactions.
7. **install:** Installs the packaged artifact into the local Maven repository (~/.m2/repository) so it can be used as a dependency in other local projects.
8. **deploy:** Copies the final package to a remote repository for sharing with other developers or projects.

Each of these phases corresponds to a **logical step** in the project's build lifecycle, ensuring that every part of the process is handled systematically.

Maven Goal

A **Maven Goal** is a **specific task** that contributes to building and managing a project. Each phase is made up of one or more goals. When a phase is executed, all the goals attached to that phase are executed in order.

A goal can be executed **independently** or as part of a phase. For example, running mvn compiler:compile executes only the compile goal of the compiler plugin, while mvn compile executes all goals bound to the compile phase and all preceding phases.

Common Maven Goals and Their Bound Phases

1. **compiler:compile –**This goal from the **Maven Compiler Plugin** compiles the project's main source code. It is bound to the **compile** phase.
2. **compiler:testCompile –**Compiles the test source code and is bound to the **test-compile** phase.
3. **surefire:test –**Runs unit tests using the **Maven Surefire Plugin**. This goal is bound to the **test** phase.
4. **jar:jar / war:war –**Packages the project into a **JAR** or **WAR** file respectively. These goals are bound to the **package** phase depending on the project type.
5. **install:install –**Installs the generated package into the local repository and is bound to the **install** phase.

Each goal belongs to a specific **plugin**, and multiple goals can be associated with one plugin. For example, the **maven-compiler-plugin** has both the compile and testCompile goals.

Relationship Between Phases and Goals

Maven's lifecycle works by executing **phases**, which in turn execute the **goals** bound to them. When you invoke a phase, all the associated goals are executed automatically. This allows Maven to maintain an organized and predictable build process.

You can view the goals and plugins bound to a particular phase using the command:

mvn help:describe -Dcmd=PHASENAME

For example, to view all goals associated with the compile phase, you can run:

mvn help:describe -Dcmd=compile

This command displays the plugin and goal details, such as:

'compile' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile

Example of Execution Flow

If you run the command:

mvn deploy

Maven will execute all preceding phases in order:

validate → compile → test-compile → test → package → integration-test → install → deploy

This sequential execution ensures that the build process is complete and consistent.

Maven Build Profiles

Maven Build Profiles are a powerful feature that allow developers to customize the build process for different environments or situations such as development, testing, staging, or production. A build profile defines a set of configuration values that can modify the default behavior of the Maven build lifecycle. This helps in creating flexible builds that adapt easily to varying requirements without needing to maintain multiple separate POM files for each environment.

Purpose of Build Profiles

The main purpose of using build profiles is to provide environment-specific configurations. For instance, a developer may want to use a lightweight database like H2 during development but switch to a full production database such as MySQL during deployment. Similarly, resource directories, plugin configurations, compiler versions, or system properties may differ between environments. By using profiles, these variations can be easily managed within a single project configuration file.

Definition of Build Profiles

A build profile is defined within the POM file or in the Maven settings file. When a profile is activated, it modifies the build configuration according to the settings defined within that profile. Each profile is identified by a unique identifier known as the profile ID. The configuration inside the profile can include properties, dependencies, repositories, plugin executions, and resource settings. Maven combines the active profile's configuration with the main project's configuration during the build process.

Profiles can be declared in three main locations. They can be defined inside the project's pom.xml file which makes them project-specific. They can also be defined in the user's settings.xml file which applies only to the current user's environment. Finally, they can be defined in the global settings.xml file, which is shared across all Maven projects on a particular system.

Activation of Profiles

Profiles can be activated in several ways depending on the build requirements. The most common method of activation is through the command line by using the -P option followed by the profile name. For example, running the command mvn clean package -Pproduction activates the profile named "production" during the build process.

Profiles can also be activated automatically based on certain conditions. These conditions may include the presence of a specific file, the existence of a property, the value of an environment variable, or the operating system on which Maven is running. For example, a profile can be configured to activate only when the operating system is Windows or when a particular property such as "env=dev" is set. Maven also supports default activation of profiles which means a profile can be automatically activated without explicitly specifying it.

Components That Can Be Customized Using Profiles

A build profile can modify several elements of a Maven build. It can add or exclude dependencies, change plugin configurations, set system or environment properties, and specify alternate source or resource directories. In addition, it can define different output directories, repositories, or even modify reporting configurations. This level of customization

ensures that each build environment can have its own set of configurations without affecting other environments.

For example, in a development environment, the profile can include debugging dependencies and enable verbose logging, while in a production environment the profile can disable debug options and include optimization parameters. Similarly, resource filtering can be customized so that configuration files include different database URLs or API keys depending on the profile that is active.

Example of Build Profile in POM File

A simple example of a Maven profile in a pom.xml file is shown below:

```
<profiles>
   <profile>
      <id>development</id>
      <properties>
         <environment>dev</environment>
      </properties>
      <build>
         <resources>
            <resource>
               <directory>src/main/resources/dev</directory>
            </resource>
         </resources>
      </build>
   </profile>

   <profile>
      <id>production</id>
      <properties>
         <environment>prod</environment>
      </properties>
      <build>
         <resources>
            <resource>
               <directory>src/main/resources/prod</directory>
            </resource>
         </resources>
      </build>
   </profile>
</profiles>
```

In this example, two profiles named “development” and “production” are defined. Depending on which profile is activated, Maven uses the corresponding resource directory for the build process.

Advantages of Using Build Profiles

The use of build profiles offers several important advantages. It eliminates the need for maintaining multiple project configurations for different environments, thereby simplifying project management. It provides flexibility in customizing builds according to deployment targets, reduces errors caused by manual configuration changes, and improves automation in continuous integration pipelines. Additionally, profiles ensure consistency across environments by allowing controlled variation in configurations.

Practical

1. Build a Java application using Hibernate and JPA annotations to perform CRUD operations on a Student entity.

Aim

To Build a Java application using Hibernate and JPA annotations to perform CRUD operations on a Student entity.

Procedure

1. Create the Maven project structure above.
2. Add the dependencies to pom.xml (Hibernate core, JPA API and H2).
3. Create the Student entity class with JPA annotations.
4. Create a StudentDao class that uses EntityManager to implement create read update delete and list methods.
5. Create a MainApp class that demonstrates the CRUD operations in sequence.
6. Build and run the application using Maven or your IDE.
7. Observe console output and SQL logs.

Program

Maven pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                            http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
```

```
<artifactId>maven-hibernate-jpa-crud</artifactId>
<version>1.0-SNAPSHOT</version>
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
  <hibernate.version>5.6.15.Final</hibernate.version>
</properties>
<dependencies>
  <!-- JPA API -->
  <dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
  </dependency>

  <!-- Hibernate ORM -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
  </dependency>

  <!-- H2 in-memory database -->
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.1.214</version>
    <scope>runtime</scope>
  </dependency>

  <!-- Optional logging dependencies -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.36</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.36</version>
  </dependency>
</dependencies>

<build>
  <plugins>
```

```
    <!-- To run the main class via mvn exec:java -->
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>3.1.0</version>
      <configuration>
        <mainClass>com.example.app.MainApp</mainClass>
      </configuration>
    </plugin>
  </plugins>
 </build>
</project>
```

persistence.xml (src/main/resources/META-INF/persistence.xml)

This config tells JPA to use the Hibernate provider and H2 in memory database. The hibernate.hbm2ddl.auto is set to create to create tables at startup.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                     http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
         version="2.2">
 <persistence-unit name="studentPU" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

  <class>com.example.app.model.Student</class>

  <properties>
    <!-- JDBC connection -->
    <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
    <property                                          name="javax.persistence.jdbc.url"
value="jdbc:h2:mem:studentdb;DB_CLOSE_DELAY=-1"/>
    <property name="javax.persistence.jdbc.user" value="sa"/>
    <property name="javax.persistence.jdbc.password" value=""/>

    <!-- Hibernate dialect -->
    <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>

    <!-- Show SQL on console -->
    <property name="hibernate.show_sql" value="true"/>
    <property name="hibernate.format_sql" value="true"/>
```

```
    <!-- Create schema at startup -->
    <property name="hibernate.hbm2ddl.auto" value="create"/>

    <!-- Optional: log bind parameters -->
    <property name="hibernate.type" value="trace"/>
  </properties>
 </persistence-unit>
</persistence>
```

Student entity class (src/main/java/com/example/app/model/Student.java)

```
package com.example.app.model;

import javax.persistence.*;

@Entity
@Table(name = "students")
public class Student {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(nullable = false)
   private String name;

   private int age;

   private String email;

   public Student() {
   }

   public Student(String name, int age, String email) {
      this.name = name;
      this.age = age;
      this.email = email;
   }

   // getters and setters

   public Long getId() {
      return id;
   }
```

```
    // no setId because id is generated

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
            public int getAge() {
                    return age;
            }

            public void setAge(int age) {
                    this.age = age;
            }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                ", email='" + email + '\'' +
                '}';
    }
}
```

StudentDao class implementing CRUD
(src/main/java/com/example/app/dao/StudentDao.java)

```
package com.example.app.dao;

import com.example.app.model.Student;

import javax.persistence.*;
import java.util.List;
```

```
public class StudentDao {

    private final EntityManagerFactory emf;

    public StudentDao(EntityManagerFactory emf) {
        this.emf = emf;
    }

    // Create
    public Student save(Student student) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();
            em.persist(student);
            tx.commit();
            return student;
        } catch (RuntimeException ex) {
            if (tx.isActive()) tx.rollback();
            throw ex;
        } finally {
            em.close();
        }
    }

    // Read by id
    public Student findById(Long id) {
        EntityManager em = emf.createEntityManager();
        try {
            return em.find(Student.class, id);
        } finally {
            em.close();
        }
    }

    // Update
    public Student update(Long id, String newName, Integer newAge, String newEmail) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();
            Student s = em.find(Student.class, id);
            if (s == null) {
                tx.commit();
                return null;
```

```
            }
            if (newName != null) s.setName(newName);
            if (newAge != null) s.setAge(newAge);
            if (newEmail != null) s.setEmail(newEmail);
            em.merge(s);
            tx.commit();
            return s;
        } catch (RuntimeException ex) {
            if (tx.isActive()) tx.rollback();
            throw ex;
        } finally {
            em.close();
        }
    }

    // Delete
    public boolean delete(Long id) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();
            Student s = em.find(Student.class, id);
            if (s != null) {
                em.remove(s);
                tx.commit();
                return true;
            } else {
                tx.commit();
                return false;
            }
        } catch (RuntimeException ex) {
            if (tx.isActive()) tx.rollback();
            throw ex;
        } finally {
            em.close();
        }
    }

    // List all
    public List<Student> findAll() {
        EntityManager em = emf.createEntityManager();
        try {
            TypedQuery<Student> q = em.createQuery("SELECT s FROM Student s",
Student.class);
            return q.getResultList();
```

```
        } finally {
            em.close();
        }
    }
}
```

MainApp class to demonstrate CRUD (src/main/java/com/example/app/MainApp.java)

```
package com.example.app;

import com.example.app.dao.StudentDao;
import com.example.app.model.Student;

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import java.util.List;

public class MainApp {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("studentPU");
        StudentDao dao = new StudentDao(emf);

        System.out.println("Starting CRUD demonstration");

        // Create
        Student s1 = new Student("Alice Johnson", 20, "alice@example.com");
        Student s2 = new Student("Bob Smith", 22, "bob@example.com");
        dao.save(s1);
        dao.save(s2);
        System.out.println("Saved students:");
        dao.findAll().forEach(System.out::println);

        // Read
        Student found = dao.findById(s1.getId());
        System.out.println("Found by id " + s1.getId() + " : " + found);

        // Update
        Student updated = dao.update(s2.getId(), "Robert Smith", 23,
"robert.smith@example.com");
        System.out.println("Updated student id " + s2.getId() + " : " + updated);

        // List after update
        System.out.println("All students after update:");
```

```
        dao.findAll().forEach(System.out::println);

        // Delete
        boolean deleted = dao.delete(s1.getId());
        System.out.println("Deleted student id " + s1.getId() + " ? " + deleted);

        // Final list
        System.out.println("All students after delete:");
        List<Student> remaining = dao.findAll();
        remaining.forEach(System.out::println);

        emf.close();
        System.out.println("Done");
    }
}
```

Sample output

```
Starting CRUD demonstration
Hibernate:
    create table students (
        id bigint generated by default as identity,
        age integer,
        email varchar(255),
        name varchar(255) not null,
        primary key (id)
    )
Hibernate:
    insert into students (age, email, name) values (?, ?, ?)
Hibernate:
    insert into students (age, email, name) values (?, ?, ?)
Saved students:
Student{id=1, name='Alice Johnson', age=20, email='alice@example.com'}
Student{id=2, name='Bob Smith', age=22, email='bob@example.com'}

Found by id 1 : Student{id=1, name='Alice Johnson', age=20, email='alice@example.com'}

Hibernate:
    select student0_.id as id1_0_0_, student0_.age as age2_0_0_, student0_.email as email3_0_0_, student0_.name as name4_0_0_ from students student0_ where student0_.id=?
Updated student id 2 : Student{id=2, name='Robert Smith', age=23, email='robert.smith@example.com'}
```

All students after update:
Student{id=1, name='Alice Johnson', age=20, email='alice@example.com'}
Student{id=2, name='Robert Smith', age=23, email='robert.smith@example.com'}

Hibernate:
delete from students where id=?
Deleted student id 1 ? true

All students after delete:
Student{id=2, name='Robert Smith', age=23, email='robert.smith@example.com'}
Done

Result

Thus, a Java application was successfully developed using **Hibernate** and **JPA annotations** to perform **CRUD (Create, Read, Update, and Delete)** operations on the **Student** entity.

3. Mini-project integrating Spring and Hibernate framework

Aim

To build a Java mini project that integrates Spring and Hibernate to perform Create Read Update Delete operations on a Student entity. The application will expose a simple REST API to perform CRUD and will persist data in an in memory H2 database. Hibernate will be used as the JPA provider and Spring will manage dependency injection and transaction management.

Procedure

1. Create a new Maven project.
2. Add Spring Boot starter dependencies for web and data JPA and H2 database to pom.xml.
3. Create the Student entity class annotated with JPA annotations.
4. Create a Spring Data JPA repository interface for Student.
5. Create a service class to encapsulate business logic and transactional operations.
6. Create a REST controller that exposes endpoints for CRUD operations.
7. Configure application properties to use H2 and enable SQL logging so Hibernate SQL can be observed.
8. Run the application and test the REST endpoints using curl or Postman.
9. Observe console output and API responses to verify correct CRUD behavior.

Maven pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0"

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>spring-hibernate-mini</artifactId>
  <version>1.0.0</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.14</version>
    <relativePath/>
  </parent>

  <properties>
    <java.version>11</java.version>
  </properties>

  <dependencies>
    <!-- Spring Boot web starter -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Data JPA (uses Hibernate as default JPA provider) -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- H2 in memory database for demo -->
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>

    <!-- Optional: Lombok for brevity. If not wanted remove it and generate getters setters -->
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
```

```
    <!-- Optional logging -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Configuration application.properties (src/main/resources/application.properties)

```
# Use H2 in memory database
spring.datasource.url=jdbc:h2:mem:studentdb;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA and Hibernate settings
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Optional: Print bind parameters
logging.level.org.hibernate.type.descriptor.sql=TRACE

# H2 console for debugging
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

spring.jpa.hibernate.ddl-auto=create-drop will create schema at startup and drop at shutdown. For persistent data use update or real DB.

Student entity class (src/main/java/com/example/springhibernate/model/Student.java)

```
package com.example.springhibernate.model;
```

```
import javax.persistence.*;
import java.io.Serializable;

@Entity
@Table(name = "students")
public class Student implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    private Integer age;

    private String email;

    public Student() {}

    public Student(String name, Integer age, String email) {
        this.name = name;
        this.age = age;
        this.email = email;
    }

    // getters and setters

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

            public Integer getAge() {
                    return age;
            }

            public void setAge(Integer age) {
```

```
                this.age = age;
        }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                ", email='" + email + '\'' +
                '}';
    }
}
```

StudentRepository interface
(src/main/java/com/example/springhibernate/repository/StudentRepository.java)

```
package com.example.springhibernate.repository;

import com.example.springhibernate.model.Student;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    // JpaRepository provides standard CRUD methods out of the box
}
```

StudentService class
(src/main/java/com/example/springhibernate/service/StudentService.java)

```
package com.example.springhibernate.service;

import com.example.springhibernate.model.Student;
import com.example.springhibernate.repository.StudentRepository;
import org.springframework.stereotype.Service;
```

```
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import java.util.Optional;

@Service
@Transactional
public class StudentService {

    private final StudentRepository repo;

    public StudentService(StudentRepository repo) {
        this.repo = repo;
    }

    public Student create(Student s) {
        return repo.save(s);
    }

    public Optional<Student> findById(Long id) {
        return repo.findById(id);
    }

    public List<Student> findAll() {
        return repo.findAll();
    }

    public Student update(Long id, Student updates) {
        return repo.findById(id).map(existing -> {
            if (updates.getName() != null) existing.setName(updates.getName());
            if (updates.getAge() != null) existing.setAge(updates.getAge());
            if (updates.getEmail() != null) existing.setEmail(updates.getEmail());
            return repo.save(existing);
        }).orElse(null);
    }

    public boolean delete(Long id) {
        return repo.findById(id).map(existing -> {
            repo.delete(existing);
            return true;
        }).orElse(false);
    }
}
```

StudentController class (REST API)
(src/main/java/com/example/springhibernate/controller/StudentController.java)

```
package com.example.springhibernate.controller;

import com.example.springhibernate.model.Student;
import com.example.springhibernate.service.StudentService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.net.URI;
import java.util.List;

@RestController
@RequestMapping("/api/students")
public class StudentController {

    private final StudentService service;

    public StudentController(StudentService service) {
        this.service = service;
    }

    @PostMapping
    public ResponseEntity<Student> create(@RequestBody Student s) {
        Student saved = service.create(s);
        return ResponseEntity.created(URI.create("/api/students/" +
saved.getId())).body(saved);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Student> getById(@PathVariable Long id) {
        return service.findById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @GetMapping
    public List<Student> getAll() {
        return service.findAll();
    }

    @PutMapping("/{id}")
    public ResponseEntity<Student> update(@PathVariable Long id, @RequestBody Student
updates) {
```

```
        Student updated = service.update(id, updates);
        if (updated == null) return ResponseEntity.notFound().build();
        return ResponseEntity.ok(updated);
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        boolean deleted = service.delete(id);
        if (!deleted) return ResponseEntity.notFound().build();
        return ResponseEntity.noContent().build();
    }
}
```

Application main class
(src/main/java/com/example/springhibernate/SpringHibernateApplication.java)

```
package com.example.springhibernate;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringHibernateApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringHibernateApplication.class, args);
    }
}
```

Sample output showing Hibernate SQL

When you run the application with spring.jpa.show-sql=true you will see logs similar to the following during startup and CRUD operations. The exact formatting depends on Spring Boot and Hibernate versions.

```
2025-10-24 20:00:00.000  INFO 12345 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080

Hibernate:
    create table students (
        id bigint generated by default as identity,
        age integer,
        email varchar(255),
        name varchar(255) not null,
        primary key (id)
    )
```

Hibernate:
insert into students (age, email, name) values (?, ?, ?)
2025-10-24 20:00:01.234 INFO 12345 --- [main] com.example.springhibernate.SpringHibernateApplication : Started SpringHibernateApplication

 # When creating via REST
Hibernate:
insert into students (age, email, name) values (?, ?, ?)

 # When reading all
Hibernate:
select student0_.id as id1_0_, student0_.age as age2_0_, student0_.email as email3_0_, student0_.name as name4_0_ from students student0_

Result

A Spring integrated Hibernate application was developed and executed successfully to perform CRUD operations on the Student entity.

Question Bank

UNIT I: CORE JAVA FUNDAMENTALS

Part A - 2 Marks

1. What is JVM? Explain its significance.
2. Differentiate between == and .equals() method in Java.
3. What is the use of the final keyword?
4. Define a constructor. Can a constructor be final?
5. Differentiate between method overloading and method overriding.
6. What is an abstract class? Give its syntax.
7. What is the use of the super keyword?
8. Define Dynamic Method Dispatch.
9. List the benefits of using packages.
10. What are the main principles of OOP?
11. What is the difference between throw and throws?
12. What are the types of inheritance in Java?

Part B - 13 Marks

1. Explain the Object-Oriented features of Java in detail. Discuss Classes, Objects, Constructors, and the different access specifiers with examples.
2. Explain how Java implements polymorphism. Discuss in detail about method overloading, method overriding, and abstract classes with examples.
3. Describe the concept of Exception Handling in Java. Explain the usage of try, catch, finally, throw, and throws with a suitable example.
4. Explain the principles of Inheritance in Java with its various types. Discuss the use of the super keyword and method overriding in the context of inheritance.

5. Explain the concepts of Interfaces and Packages. How do they support abstraction and modularity in Java? Provide examples.

Part C - 15 Marks

1. Elaborate on the following with examples: a) static members b) final keyword c) this keyword d) Access Specifiers.
2. Write a detailed note on Exception Handling mechanisms in Java. Develop a Java application that demonstrates the use of built-in exceptions and the creation of user-defined exceptions.

UNIT II: MULTITHREADING AND JDBC

Part A - 2 Marks

1. Define a Thread. List the states in a thread lifecycle.
2. Differentiate between process and thread.
3. What is Synchronization and why is it needed?
4. How can you create a thread in Java?
5. What is the difference between extends Thread and implements Runnable?
6. Name the different types of JDBC drivers.
7. Write the syntax for Class.forName() to load a JDBC driver.
8. What are the main steps to connect to a database in JDBC?
9. What is Inter-thread Communication?
10. What is a Daemon Thread?

Part B - 13 Marks

1. Explain the Java Thread Model and the life cycle of a thread. Write a Java program to create two threads, one for printing even numbers and another for odd numbers.
2. Explain the concept of thread synchronization. Why is it important? Write a program to demonstrate synchronized method.
3. Explain the architecture of JDBC with a neat diagram.

4. Write the steps to connect to a MySQL database. Develop a Java application to perform INSERT and UPDATE operations on a Student table.
5. Explain the methods wait(), notify(), and notifyAll() with an example program for producer-consumer problem.

Part C - 15 Marks

1. Discuss the concept of Multithreading in Java. Explain how to create threads using both the Thread class and Runnable interface. Also, explain the need for and implementation of thread synchronization.
2. Explain the JDBC architecture in detail. Develop a complete Java application to demonstrate all CRUD (Create, Read, Update, Delete) operations on a database table of your choice.

UNIT III: WEB CLIENT UI AND JAVA SERVLETS

Part A - 2 Marks

1. What is a Servlet?
2. Draw the Servlet Lifecycle.
3. Differentiate between GET and POST methods.
4. What is the role of the RequestDispatcher?
5. What is a ServletContext?
6. How are sessions maintained using Cookies?
7. What is the purpose of the HttpSession object?
8. List the advantages of Servlets over CGI.
9. What is the use of the web.xml file?
10. What are the different scope objects in Servlets?

Part B - 13 Marks

1. Explain the lifecycle of a Servlet with a neat diagram.
2. Write a servlet program to handle an HTTP POST request for a user login form and display a welcome message.
3. Describe Session Management techniques in Servlets. Compare the use of Cookies and HttpSession.
4. Develop a servlet for uploading files to the server. Explain the @MultipartConfig annotation.
5. Explain the role of ServletConfig and ServletContext objects with examples.

Part C - 15 Marks

1. Explain the Java EE architecture and the role of servlets in it. Develop a servlet-based user registration system that accepts form data, stores it in a database, and manages the user session.
2. Discuss the Servlet API in detail. Write a servlet program that demonstrates the use of RequestDispatcher for including and forwarding requests, and also manages user state using HttpSession.

UNIT IV: JAVA SERVER PAGES (JSP) AND JSTL

Part A - 2 Marks

1. What is JSP?
2. List the JSP scripting elements.
3. What is Expression Language (EL) in JSP?
4. What are JSTL Core tags? Give two examples.
5. Differentiate between JSP include directive and include action.
6. What is the scope of a session object in JSP?
7. List any four JSP Implicit Objects.

8. Differentiate between Servlet and JSP.
9. What is the purpose of the <jsp:useBean> action?
10. What are the different types of JSTL tags?

Part B - 13 Marks

1. Explain the JSP lifecycle phases with a neat diagram.
2. Describe the JSP Standard Tag Library (JSTL). Write a JSP code snippet using JSTL core tags to iterate over a collection of books and display them in a table.
3. Explain the various JSP implicit objects and their uses.
4. Develop a JSP page for a simple login application that uses EL to display error messages.
5. Compare and contrast JSP Scriptlets, Declarations, and Expressions. Why is it recommended to avoid Scriptlets?

Part C - 15 Marks

1. Explain the architecture of JSP and its various implicit objects. Develop a JSP application for a simple student registration system that uses JSTL for displaying data and performing conditional checks.
2. Compare and contrast Servlets and JSP. Develop a dynamic web application using JSP and Servlets (MVC Model 1) that performs user authentication and displays a personalized dashboard.

UNIT V: JAVA FRAMEWORKS AND BUILD TOOLS

Part A - 2 Marks

1. What is Dependency Injection in the Spring framework?
2. Define MVC architecture.
3. What is the purpose of the pom.xml file in Maven?

4. List the key features of Hibernate ORM.
5. What is Spring Boot? State its advantages.
6. What is an ORM tool?
7. What is the role of the DispatcherServlet in Spring MVC?
8. What are the different modules in the Spring framework?
9. What is a Maven Repository?
10. What are the core concepts of Maven?

Part B - 13 Marks

1. Explain the core concepts of the Spring Framework, including IOC (Inversion of Control) and Dependency Injection.
2. Describe the Hibernate architecture with a neat diagram.
3. Explain the MVC architecture in the context of web applications. Describe how Spring MVC implements this architecture and its workflow.
4. Explain the Maven build lifecycle and its core phases.
5. Write a short note on Spring Boot starters and auto-configuration.

Part C - 15 Marks

1. Explain the MVC architecture in detail. Describe the workflow of a Spring MVC application with a neat diagram. Develop a simple application using Spring Boot to demonstrate a REST endpoint.
2. Discuss the structure of a pom.xml file. Develop a simple CRUD application using Spring Boot and Hibernate for a Product entity.