WEB TECHNOLOGIES

MASTER OF COMPUTER APPLICATIONS(MCA)

SEMESTER-II, PAPER-III

LESSON WRITERS

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Vasantha Rudrarnalla
Faculty, Department of CS&E
Acharya Nagarjuna University

Mrs. Appikatla Pushpa Latha
Faculty, Department of CS&E
Acharya Nagarjuna University

Dr. U. Surya Kameswari
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

EDITOR

Dr. Neelima Guntupalli
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

ACADEMIC ADVISOR

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

DIRECTOR, I/c.

Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W., M.Phil., Ph.D.
Professor
Centre for Distance Education
Acharya Nagarjuna University
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208
0863- 2346259 (StudyMaterial)
Website www.anucde.info
E-mail:anucdedirector@gmail.com

MCA : WEB TECHNOLOGIES

First Edition : 2025

No. of Copies :

This book is exclusively prepared for the use of students of MASTER OF COMPUTER APPLICATIONS(MCA),Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited circulation only.

Publishedby:

Prof. V. VENKATESWARLU
Director, I/c
CentreforDistanceEducation,
AcharyaNagarjunaUniversity

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University.

MASTER OF COMPUTER APPLICATIONS (MCA)

Semester-II, Paper-III

203MC24:Web Technologies

SYLLABUS

UNIT I

Java Basics: Java buzzwords, Review of OOP concepts, dynamic binding, abstract classes and methods, interfaces, Packages.

GUI Programming with JAVA: Event Handling, Applets, Swing - Introduction to Swing, Swing vs. AWT,MVC architecture, Hierarchy for Swing components, Containers ,JFrame, JApplet, JWindow, JDialog, JPanel, A simple swing application, Overview of several swing components, Layout management - Layout manager types – border, grid, flow, box.

UNIT II

HTML: Common Tags: List, Tables, images, forms, Frames, Cascading Style Sheets;

Java Script: Introduction to Java Scripts, Objects in Java Script, Dynamic HTML with Java Script.

XML: Document type definition, XML Schemas, Document Object model, Presenting XML, Using XML Processors: DOM and SAX

UNIT III

JDBC: Introduction to JDBC – Connections – Internal Database Connections – Statements – Results Sets - Prepared Statements - Callable Statements.

Network Programming and RMI: why networked Java – Basic Network Concepts – looking up Internet Addresses – URLs and URIs – UDP Datagrams and Sockets – Remote Method Invocation.

Unit –IV

Web Servers and Servlets: Tomcat web server, Introduction to Servlets: Lifecycle of a Servlet, JSDK, The Servlet API, Thejavax.servlet Package, Reading Servlet parameters, Reading Initialization parameters. The javax.servlet HTTP package, Handling Http Request & Responses, Using Cookies-Session Tracking, Security Issues.

Introduction to JSP: The Problem with Servlet. The Anatomy of a JSP Page, JSP Processing. JSP Application Design with MVC Setting Up and JSP Environment: Installing the Java Software Development Kit, Tomcat Server & Testing Tomcat

Prescribed Textbooks

1. The Complete reference Java, Herbet Schildt, 7th Edition, McGraw Hill.
2. Java Programming with JDBC ;Donald Bales, O'Reilly
3. Web Technologies – a computer science perspective, Jeffrey C. Jackson, Pearson, 2007.

Reference Textbooks

1. Java Network Programming, elliotte Rusty Harold, 3rd Edition
2. Java Server Pages – Hans Bergsten, SPD O'Reilly
3. Robert W. Sebesta, "Programming the World Wide Web", Third Edition, Pearson Education (2007).
4. Anders Moller and Michael schwartzbach, "An Introduction to XML and Web Technologies", Addison Wesley (2006)
5. Chris Bates, "Web Programming–Building Internet Applications", Second Edition, Wiley (2007).

(203MC24)

M.C.A. DEGREE EXAMINATION, MODEL QUESTION PAPER
Second Semester
203MC24:Web Technologies

Time: 3 Hours **Max. Marks: 70**

SECTION-A

Answer Question No.1 Compulsory **2 Marks × 7 = 14 Marks**

1. a) Define Interface
b) Differentiate Swing and AWT
c) List out Table Tags
d) Explain DTD
e) Explain Result Sets
f) Explain Servlet API
g) Write steps for installing JDK

SECTION-B

Answer ONE Question from Each Unit **4 × 14 = 56 Marks**

UNIT – I

2. a) Differentiate abstract class and interface

b) Write short notes on dynamic binding

OR

a) What are events handler? Explain five event handlers

UNIT – II

3. a) How to place hyperlink on web page? Explain <A> tag in detail

OR

a) What are style sheets? Explain various types.

b) Write a java script program to find factorial of a given numbers.

UNIT – III

4. a) Explain JDBC architecture and different types of devices available

OR

a) Explain about RMI architecture

UNIT – IV

5. a) What is JSP? What are the advantages and Disadvantages of JSP?

b) Explain the anatomy of JSP

OR

a) Explain the life cycle of Servlet.

CONTENTS

LESSON- 1

JAVA CONCEPTS AND OBJECT-ORIENTED PROGRAMMING

Aim and Objectives:

- To understand and explain the key Java buzzwords such as simple, secure, platform-independent, robust, and multithreaded.
- To analyze how Java's design principles support the development of portable, efficient, and reliable applications.
- To review fundamental object-oriented programming (OOP) concepts including encapsulation, inheritance, polymorphism, and abstraction.
- To demonstrate the practical application of OOP principles in Java programming.
- To develop Java programs that apply both Java buzzword features and object-oriented techniques for real-world problem solving.

STRUCTURE:

1. 1 INTRODUCTION TO JAVA

Java is a high-level, class-based, object-oriented programming language developed by Sun Microsystems in 1995 (now owned by Oracle Corporation). It is designed to have as few implementation dependencies as possible, allowing developers to write code once and run it anywhere (WORA – *Write Once, Run Anywhere*), thanks to the Java Virtual Machine (JVM).

1.1.1Basic structure of a Java program:

```
// This is a simple Java program
public class HelloWorld
{       // 1. Class declaration
   public static void main(String[] args)
```

```
{   // 2. main method - program entry point
      System.out.println("Hello, World!");  // 3. Statement to print output
}
}
```

Output:

Hello World

1.1.2Explanation of the Basic Structure:

1. **Class Declaration:**
   Every Java program must have at least one class. The class name here is HelloWorld. The filename must match the class name (HelloWorld.java).
2. **main() Method:**
   This is the entry point of any Java application. The JVM looks for this method to start program execution. It must be declared exactly as public static void main(String[] args).
3. **Statements:**
   Inside the main method, you write statements that perform actions. For example, System.out.println() prints text to the console.
4. **Curly Braces {}:**
   These define the beginning and end of classes and methods.

1.1.3 Uses of Java:

- Web applications (using Spring, JSP, Servlets)
- Mobile applications (especially Android)
- Desktop GUI applications (JavaFX, Swing)
- Enterprise applications
- Embedded systems
- Scientific and research applications

1.2 Java buzzwords

Java buzzwords are key features or concepts that describe the design and philosophy of the Java programming language. These buzzwords highlight why Java has been popular and widely adopted. Here are the major Java buzzwords along with brief explanations and examples:

Java buzzwords are key features or concepts that describe the design and philosophy of the Java programming language. These buzzwords highlight why Java has been popular and widely adopted. Here are the major Java buzzwords along with brief explanations and

examples:

1. Simple
2. Secure

3. Portable
4. Object-Oriented
5. Robust
6. Architecture Neutral (or) Platform Independent
7. Multithreaded
8. Interpreted
9. High Performance
10. Distributed
11.Dynamic

1. Simple

Java is designed to be easy to learn and use. It eliminates complex features like pointers and operator overloading.

Example: Program to add two numbers (input/output)

```
import java.util.Scanner;
public class SimpleExample
{
public static void main(String[] args)
{
Scanner sc = new Scanner(System.in);
 System.out.print("Enter first number: ");
int a = sc.nextInt();
System.out.print("Enter second number: ");
int b = sc.nextInt();
int sum = a + b;
System.out.println("Sum is: " + sum);
sc.close();
    }
}
```

Input:

Enter first number: 5
Enter second number: 7

Output:

Sum is: 12

2. Secure

Java has built-in security features like the absence of pointers, bytecode verification, and the Security Manager.

Example: 1.A simple print statement showing security message.

```
public class SecureExample
{
public static void main(String[] args) {
System.out.println("Java ensures security with runtime checks.");
   }
}
```

Output:

Java ensures security with runtime checks.

Example:2.Applets run inside a restricted sandbox environment to prevent unauthorized access.

3. Portable

Java programs can move easily from one system to another because there are no platform-specific features.

Example: Same as above. The compiled .class file runs on Windows, Linux, Mac etc. with JVM.

```
public class PortableExample

{

 public static void main(String[] args)

{

System.out.println("Java program is portable across platforms.");

}

}
```

Output:

Java program is portable across platforms.

4. Object-Oriented

Everything in Java is treated as an object. It uses classes and objects to model real-world entities.

Example: Represent a Car and print its behavior

```
class Car
{
String color;

Car(String c)
{
      color = c;
   }
   void drive()
{
System.out.println("The " + color + " car is driving.");
 }
}
public class OOPExample
{
   public static void main(String[] args)
{
Car myCar = new Car("red");
myCar.drive();
 }
}
```

Output:

The red car is driving.

5. Robust

Java emphasizes error checking at compile time and runtime with strong memory management.

Example: Handling division by zero

```
import java.util.Scanner;

public class RobustExample
{
   public static void main(String[] args)
{
Scanner sc = new Scanner(System.in);
System.out.print("Enter numerator: ");
      int num = sc.nextInt();
      System.out.print("Enter denominator: ");
```

```
        int denom = sc.nextInt();
        try
 {
int result = num / denom;
            System.out.println("Result is: " + result);
        }
catch (ArithmeticException e)
{
            System.out.println("Error: Division by zero is not allowed.");
        }
sc.close();
    }
}
```

Input:

Enter numerator: 10
Enter denominator: 0

Output:

Error: Division by zero is not allowed.

6. Architecture Neutral

Java bytecode is not tied to any specific machine architecture, making it portable.

Example:

```
public class ArchitectureNeutral

{

public static void main(String[] args)

 {

System.out.println("This bytecode runs on any architecture with JVM.");

    }

}
```

Output:

This bytecode runs on any architecture with JVM.

(Or)

Platform Independent

Java code is compiled into bytecode, which can run on any machine with a Java Virtual Machine (JVM).

Example:

javac Hello.java # Compiles to Hello.class (bytecode)

java Hello # Runs on any OS with JVM

7. Multithreaded

Java supports multithreading—multiple threads of execution can run concurrently.

Example: Two threads printing messages

```
class MyThread extends Thread
{
   String name;

MyThread(String n)
 {
      name = n;
   }

   public void run() {
      for (int i = 1; i <= 3; i++)
 {
         System.out.println(name + " is running: " + i);
         try { Thread.sleep(500); } catch (InterruptedException e) {}
      }
   }
}

public class MultithreadExample
{
public static void main(String[] args)
{
MyThread t1 = new MyThread("Thread 1");
MyThread t2 = new MyThread("Thread 2");
      t1.start();
  t2.start();
   }
```

```
}
```

Possible Output:

```
Thread 1 is running: 1
Thread 2 is running: 1
Thread 1 is running: 2
Thread 2 is running: 2
Thread 1 is running: 3
Thread 2 is running: 3
```

(Note: Output may vary because threads run concurrently.)

8. Interpreted

Java enables the creation of cross-platform programs by compiling into an intermediate representation calledJava bytecode. The byte code is interpreted to any machine code so that it runs on the native machine.

Example : Simple HelloWorld (No input)

```
public class HelloWorld

{

  public static void main(String[] args)

 {

  System.out.println("Hello, Java is interpreted!");

}

}
```

Output:

```
Hello, Java is interpreted!
```

Note:

1. You write the Java source code.
2. You compile it once using javac to get bytecode (.class file).
3. The JVM interprets this bytecode to native machine instructions at runtime, on whatever platform you use.
4. So, the same **.class** file runs everywhere without needing recompilation — this is the interpreted nature of Java.

9. High Performance

Java performance is better than traditional interpreted languages due to Just-In-Time (JIT) compilation.

Example: Simple computation to show performance

```
public class HighPerformance
{
public static void main(String[] args)
{
  long start = System.currentTimeMillis();
 long sum = 0;
for (long i = 0; i < 1_000_000_000L; i++)
 {
 sum += i;
}
 long end = System.currentTimeMillis();
 System.out.println("Sum: " + sum);
 System.out.println("Time taken in milliseconds: " + (end - start));
  }
}
```

Output: (sample)

Sum: 499999999500000000

Time taken in milliseconds: 500

(The exact time depends on your machine, but JIT optimizes such code during runtime.)

10. Distributed

Java supports networking and remote method invocation (RMI) to build distributed applications.

Example:: Simple Client-Server using sockets.

Server Program:

```
import java.net.ServerSocket;

import java.net.Socket;

import java.io.PrintWriter;

import java.io.IOException;

public class SimpleServer

{

public static void main(String[] args)

 {

 try (ServerSocket server = new ServerSocket(5000))

{

System.out.println("Server started. Waiting for client...");

Socket client = server.accept();  // accept client connection

 PrintWriter out = new PrintWriter(client.getOutputStream(), true);

out.println("Hello from Server!");

client.close();

 }

catch (IOException e)

{

e.printStackTrace();

  }
```

```
}

}
```

Client Program:

```
import java.net.Socket;

        import java.util.Scanner;

import java.io.InputStreamReader;

import java.io.BufferedReader;

import java.io.IOException;

public class SimpleClient

{

public static void main(String[] args)

 {

try (Socket socket = new Socket("localhost", 5000))

 {

BufferedReader in = new BufferedReader(new

InputStreamReader(socket.getInputStream())

        String message = in.readLine();

System.out.println("Message from Server: " + message);

}

        catch (IOException e)

{

        e.printStackTrace();

}
```

```
}

}
```

How to run:

1. Run SimpleServer first; it waits for a client.

2. Run SimpleClient; it connects and receives the message.

Output on Client:

Message from Server: Hello from Server!

Output on Server:

Server started. Waiting for client...

11. Dynamic

Java can dynamically load classes at runtime and supports runtime reflection.

Example: Dynamically loading a class

```
public class DynamicExample
 {
   public static void main(String[] args)
 {
Try
 {
        Class<?> cls = Class.forName("java.util.ArrayList");
        System.out.println("Class loaded dynamically: " + cls.getName());
      }
catch (ClassNotFoundException e)
{
        System.out.println("Class not found.");
      }
   }
}
```

Output:

Class loaded dynamically: java.util.ArrayList

1.3 REVIEW OF OOP CONCEPTS

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects which contain data and methods. Java is an object-oriented language that uses the following key concepts:

1. Class and Object

- **Class:** Blueprint or template for creating objects. It defines properties (fields) and behaviors (methods).
- **Object:** An instance of a class.

Example:

```
class Dog
{
   String name;
   int age;

   void bark()
{
      System.out.println(name + " is barking.");
   }
}

public class TestDog
{
   public static void main(String[] args)
 {
      Dog myDog = new Dog();          // Create object
      myDog.name = "Buddy";           // Set properties
      myDog.age = 3;
      myDog.bark();                   // Call method
   }
}
```

Output:

```
Buddy is barking.
```

2. Encapsulation

- Wrapping data (variables) and code (methods) together as a single unit.
- Use of **private** fields and **public** getter/setter methods to control access.

Example:

```
class Person
 {
   private String name;       // private variable

   public void setName(String n) {    // setter method
      name = n;
}
   public String getName()
{          // getter method
      return name;
   }
}

public class TestPerson
{
   public static void main(String[] args)
{
      Person p = new Person();
p.setName("Alice");
      System.out.println("Name is: " + p.getName());
   }
}
```

Output:

Name is: Alice

3. Inheritance

- Mechanism where one class (subclass/child) inherits fields and methods from another (superclass/parent).
- Supports code reuse.

Example:

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

```
class Cat extends Animal
{
   void sound()
 {
      System.out.println("Cat meows");
   }
}

public class TestInheritance
 {
   public static void main(String[] args)
 {
      Cat c = new Cat();
      c.sound();   // calls overridden method
   }
}
```

Output:

Cat meows

4. Polymorphism

- Ability to take many forms.
- Method overriding: Subclass provides specific implementation.
- Method overloading: Same method name, different parameters.

Example (Method Overriding):

```
class Bird
{
   void sound()
{
      System.out.println("Bird chirps");
   }
}

class Parrot extends Bird
 {
   void sound()
{
      System.out.println("Parrot talks");
   }
```

```
}

public class TestPolymorphism
{
   public static void main(String[] args)
 {
      Bird b = new Parrot();
b.sound();  // calls Parrot's method (runtime polymorphism)
   }
}
```

Output: Parrot talks

5. Abstraction

- Hiding complex implementation details and showing only the essentials.
- Can be achieved using abstract classes or interfaces.

Example (Abstract Class):

```
abstract class Shape
{
   abstract void draw();}

class Circle extends Shape
 {
   void draw()
 {
      System.out.println("Drawing Circle"); }}

public class TestAbstraction {
   public static void main(String[] args) {
      Shape s = new Circle();
s.draw();
   }
}
```

Output: Drawing Circle

1.4 SUMMARY

Java is a high-level, object-oriented programming language developed by Sun Microsystems in 1995, now owned by Oracle. It follows the "Write Once, Run Anywhere" philosophy, made possible through the Java Virtual Machine (JVM). A basic Java program includes class declarations, the main() method, and output statements. Java is widely used in web, mobile,

desktop, and enterprise applications. It is known for its key features—called Java buzzwords—such as simplicity, object-orientation, platform independence, security, robustness, multithreading, portability, high performance, distributed computing, and dynamic class loading. Each of these features is demonstrated through small, functional code examples. Java is also rooted in Object-Oriented Programming (OOP) principles including class and object creation, encapsulation, inheritance, polymorphism, and abstraction. These concepts help in modeling real-world systems, promoting code reusability, and building secure, maintainable applications. Through its rich feature set and OOP foundation, Java remains a powerful and versatile language for modern software development.

1.5 KEY TERMS

JVM (Java Virtual Machine)**,** WORA (Write Once, Run Anywhere) **,**Class **,**Object**,** EncapsulationInheritance**,** Abstraction **,**Platform Independent**,** Multithreading, Robust **,**Secure **,**Dynamic**.**

1.6 SELF-ASSESSMENT QUESTIONS

1. What is the role of the main() method in a Java program, and why must it be declared as public static void main(String[] args)?
2. Explain any three Java buzzwords and illustrate their meaning with an example.
3. How does Java achieve platform independence? Mention the role of the Java Virtual Machine (JVM) in this context.
4. Differentiate between encapsulation and abstraction in Java with suitable code examples.
5. What is polymorphism in Java? Provide a code example showing method overriding to demonstrate runtime polymorphism.

1.7 FURTHER READINGS

1. The Complete reference Java, Herbet Schildt, 7th Edition, McGraw Hill.
2. Java for Programmers, P.J.Deitel and H.M.Deitel, PEA (or) Java: How to Program , P.J.Deiteland H.M.Deitel,PHI
3. ObjectOrientedProgrammingthroughJava,P.RadhaKrishna,UniversitiesPress.

Dr. Kampa Lavanya

LESSON- 2

ADVANCED JAVA CONCEPTS: BINDING, ABSTRACTION, AND INTERFACES

Aim and Objectives:

- Understand the concept of dynamic binding and its role in achieving runtime polymorphism in Java programs through method overriding.
- Learn the purpose and usage of abstract classes and methods in designing flexible and partially implemented classes for future extension.
- Explore the use of interfaces to achieve full abstraction and enable multiple inheritance in Java, promoting contract-based design.
- Differentiate clearly between abstract classes and interfaces, and identify appropriate scenarios for their use in software development.
- Gain practical knowledge on creating and using packages to group related classes and interfaces, improve modularity, and manage namespace efficiently in Java applications.

STRUCTURE:

2.1UNDERSTANDING BINDING IN JAVA

Before exploring static and dynamic binding, it is important to grasp what binding means in the context of Java programming. Binding is the process of associating a method call with its actual method definition. In simpler terms, it determines which method gets executed when a method is invoked in the code.

Binding plays a crucial role in polymorphism, one of the core principles of object-oriented programming in Java. Polymorphism allows objects of different classes to be treated as objects of a common superclass. This is typically achieved through inheritance and method overriding, where a subclass can provide its own version of a method already defined in its parent class.

The type of binding—whether static or dynamic—depends on when the method resolution takes place and the type of reference used to call the method.

2.1.1 Types of Binding in Java

Java supports two primary types of method binding:

1. **Static Binding (Early Binding)**
   - Happens at compile time.
   - Used with private, static, and final methods as well as method overloading.
2.
3. **Dynamic Binding (Late Binding)**
   - Occurs at runtime.
   - Used with method overriding and supports runtime polymorphism.

2.2. WHAT IS DYNAMIC BINDING IN JAVA?

Dynamic Binding, also known as **late binding**, is a fundamental concept in object-oriented programming, particularly in Java. It refers to the process where method calls are resolved **at**

runtime, rather than during compilation.
This mechanism is key to achieving **runtime polymorphism**. When a method is overridden in a subclass, and the method is called through a reference of the superclass type, Java defers the method resolution until the program runs. This allows the JVM to determine the actual object type at runtime and invoke the correct method implementation.

2.2.1 Difference from Static Binding

In contrast, Static Binding (or early binding) occurs at compile time, where the method call is resolved based on the declared type of the reference variable. It is typically used for:

- private, static, or final methods
- Overloaded methods

Static binding leads to faster execution, but lacks the flexibility offered by dynamic binding.

2.2.2 Importance of Dynamic Binding

- Enables runtime polymorphism
- Supports method overriding
- Enhances flexibility and extensibility of code
- Promotes code reusability by treating subclass objects as superclass types
- Allows seamless addition of new functionality without altering existing code

2.2.3 Basics of Binding in Java

In Java**,** binding refers to the process of linking a method call to the actual method implementation. This process can take place at different stages of program execution and is primarily categorized into two types:

1. Static Binding (Early Binding)

- Binding occurs at compile time**.**
- The compiler determines the method to be invoked based on the reference type.
- Commonly applied to methods that are private, static, or final, since such methods cannot be overridden.
- Used in method overloading, where multiple methods share the same name but differ in parameters.

Example:

```
public class StaticBindingExample {
   public static voiddisplay() {
      System.out.println("Static method called");
   }
   public static void main(String[] args) {
      StaticBindingExample.display();  // Method resolved at compile time
   }
}
```

Output:

Static method called

2. Dynamic Binding (Late Binding)

- Binding takes place at runtime**.**
- The Java Virtual Machine (JVM)determines the method to be executed based on theactual object type, not the reference type.
- It is primarily used with overridden methods in inheritance to achieve runtime polymorphism**.**
- Enables more flexible and extensible code through dynamic method dispatch.

Example:

```
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 DynamicBindingExample {
   public static void main(String[] args) {
      Animal a = new Dog();  // Reference is of type Animal, object is of type Dog
a.sound();           // Method resolved at runtime (dynamic binding)
   }
}
```

Output:

Dog barks

Characteristics of Static Binding in Java

- The method call is determined at compile time.
- Applies to private, static, and final methods.
- Method overriding is not involved.

Characteristics of Dynamic Binding in Java

- The method call is resolved at runtime, depending on the actual object type.
- Supports method overriding and enables runtime polymorphism.
- Applicable to non-static and non-final methods.

Feature	Static Binding	Dynamic Binding
Binding Time	Occurs at compile time	Occurs at runtime
Method Types	Applies to private, static, and final methods	Applies to non-static and non-final methods
Method Overriding	Not involved	Involves method overriding
Polymorphism	Does not support runtime polymorphism	Enables runtime polymorphism
Performance	Faster, as resolution happens at compile time	Slightly slower, due to runtime method resolution
Decision Based On	Based on reference type	Based on object type at runtime
Example	static void show()	void show() (overridden in subclass)

2.2.4 Java Dynamic Binding: Using the super Keyword

Dynamic Binding in Java refers to method calls being resolved at **runtime** based on the actual object type. It allows **method overriding** and supports **runtime polymorphism**.

The super keyword is used within a subclass to refer to its **immediate superclass**. When used in the context of dynamic binding, super can help invoke the **overridden method** from the parent class, even if the subclass has provided its own implementation.

Key Points:

- super is used to access the superclass version of an overridden method.
- It bypasses dynamic binding for that specific method call.
- It is useful when the subclass wants to **extend or modify** the behavior of the superclass method.

Example Program: Dynamic Binding with super

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}
class Dog extends Animal {
   void sound() {
      System.out.println("Dog barks");
   }
   void display() {
sound();        // Calls Dog's overridden method (dynamic binding)
super.sound();  // Calls Animal's method using super
   }
}
public class Main {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.display();
   }
}
```

Output:

```
Dog barks
Animal makes a sound
```

Explanation:

- sound() is overridden in Dog.
- d.sound() calls the method dynamically based on object type (Dog).
- super.sound() calls the superclass (Animal) method explicitly, bypassing dynamic binding for that call.

How Dynamic Binding Works in Java

Dynamic Binding (also called late binding) is a process where the method call is resolved at runtime, not at compile time**.** It enables runtime polymorphism, allowing a program to decide which method implementation to execute based on the actual object type, not the reference type.

How It Works – Step by Step:

1. **Inheritance**: A subclass overrides a method defined in a superclass.
2. **Reference Variable**: A superclass reference is used to refer to a subclass object.
3. **Method Call**: When a method is called on this reference, Java checks the actual object typeatruntime**.**
4. **Runtime Resolution**: The JVM dynamically binds the method call to the overridden method in the subclass.

Example:

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}
class Cat extends Animal {
   void sound() {
      System.out.println("Cat meows");
   }
}
public class Main {
   public static void main(String[] args) {
      Animal a = new Cat();  // Superclass reference, subclass object
a.sound();            // Dynamic Binding: calls Cat's sound()
   }
}
```

Output:
Cat meows

Why It Matters:

- Supports runtime polymorphism**.**

- Increases flexibility and scalability of the code.
- Allows programs to be more modular and maintainable**.**

Key Points about Dynamic Binding

1. **Occurs at Runtime**
   - Method calls are resolved during program execution, not at compile time.
2. **Supports Polymorphism**
   - Enables runtime polymorphism, allowing different behaviors through a common interface or superclass.
3. **Involves Method Overriding**
   - Works when a subclass overrides a method from the superclass.
4. **Reference Type vs Object Type**
   - The reference type determines which methods are accessible, but the actual object type determines which overridden method is called.
5. **Used with Non-static Methods**
   - Applies only to non-static, non-final, and non-private methods (as these can be overridden).
6. **Enhances Flexibility**
   - Makes the code more modular, scalable, and maintainable.
7. **Handled by JVM**
   - The Java Virtual Machine determines the correct method to invoke at runtime.
8. **Improves Code Reusability**
   - Same method call can result in different behaviors depending on the object, enabling code reuse through inheritance.

Advantages of Java Dynamic Binding

1. **Enables Runtime Polymorphism**
   - Allows the same method call to behave differently based on the actual object type at runtime.
2. **Supports Method Overriding**
   - Lets subclasses provide specific implementations of methods defined in the superclass, enhancing flexibility.
3. **Improves Code Flexibility and Maintainability**
   - You can write more general code using superclass or interface references and extend it later without modifying existing code.
4. **Reduces Code Complexity**
   - Avoids the need for complex if-else or switch statements by using overridden methods in subclass objects.
5. **Promotes Loose Coupling**
   - Objects interact through abstract classes or interfaces rather than concrete implementations, promoting better design.
6. **Supports Extensibility**
   - New subclasses with different behaviors can be added without affecting the base class or existing code.
7. **Encourages Reusability**
   - Superclass code can be reused, while subclasses override only the necessary parts, minimizing duplication.

8. **Foundation for Design Patterns**
    - Essential for implementing many object-oriented design patterns like Strategy, Command, and Template Method.
9. **Enables Dynamic Behavior**
    - Behavior of the program can be changed at runtime, which is especially useful in frameworks and plug-in architectures.

Note: Dynamic binding in Java is the process where method calls are resolved at runtime, not at compile time, allowing for polymorphism and dynamic method invocation. This mechanism empowers developers to write flexible, extensible, and maintainable code by leveraging core principles of object-oriented programming. The examples provided illustrate how overridden methods are dynamically selected based on the actual object type at runtime, highlighting the importance and practical value of dynamic binding in real-world Java applications.

2.3 Abstract Classes and Methods

What is an Abstract Class?

An abstract class is a class that is declared with the abstractkeyword. It may or may not contain abstract methods (methods without a body), but it cannot be instantiated directly.
It acts as a base class that other classes extend to provide specific implementations.

Example:

```
abstract class Shape {
   abstract void draw();  // abstract method
   void display() {
      System.out.println("Displaying shape");}}
```

What is an Abstract Method?

An abstract method is a method declared without an implementation in the abstract class.
Subclasses must override this method.

Syntax:

abstract returnType methodName();
Abstract methods are used when you want to force subclasses to implement specific behavior**.**

2.3.1 Key Characteristics of Abstract Classes

Feature	Description
abstract keyword	Used to define an abstract class or method
Can contain abstract methods	Methods declared without a body
Can contain concrete methods	Fully implemented methods
Can have constructors	To initialize fields when subclass objects are created
Can have instance variables	Like normal classes
Can extend another class	Abstract or concrete
Can be extended by another class	Which must implement abstract methods

Example: Abstract Class with Method Implementation

```
abstract class Animal {
   abstract void makeSound();  // abstract method

   void breathe() {
      System.out.println("Animal breathes");
   }
}

class Cat extends Animal {
   void makeSound() {
      System.out.println("Cat meows");
   }
}
public class Main {
   public static void main(String[] args) {
      Animal a = new Cat();  // Polymorphism
a.makeSound();        // Calls Cat's method
a.breathe();          // Calls inherited method
   }
}
```

Why Use Abstract Classes?

Advantages:

- Provides a common interface for all subclasses.
- Helps implement partial abstraction.
- Promotes code reuse by defining common behavior in the abstract class.
- Ideal when multiple classes share common methods but also require specific implementations.

Rules for Abstract Classes and Methods

- A class must be declared abstract if it contains any abstract methods.
- Abstract classes cannot be instantiated**.**
- Subclasses must override all abstract methods, unless the subclass is also abstract.
- Abstract methods cannot be static**,** private**,** or final**.**
- Constructors in abstract classes can be used by subclasses through super().

Abstract Class vs Interface (Key Differences)

Feature	Abstract Class	Interface
Keywords	Abstract	interface
Method Implementation	Can have both abstract and concrete methods	Java 8+ allows default and static methods
Multiple Inheritance	Not supported	Supported (a class can implement multiple interfaces)
Constructors	Can have constructors	Cannot have constructors
Access Modifiers	Can use public, protected, etc.	All methods are public by default
Use Case	When classes share common base behavior	When unrelated classes share behavior

Real-World Use Cases

- **Shape hierarchy**: Shape (abstract) → Circle, Rectangle, etc.
- **Employee hierarchy**: Employee (abstract) → FullTimeEmployee, PartTimeEmployee
- **Game development**: Character (abstract) → Hero, Villain

Note:

- Use abstract classesto define a common structure and partial implementation.
- Use abstract methods to force specific behavior in subclasses.
- Abstract classes play a crucial role in polymorphism**,** inheritance, and code organization**.**

2.4 INTERFACES

What is an Interface?

An interfacein Java is a reference type, similar to a class that can contain only abstract methods(until Java 7) and constant variables. Since Java 8 and later, interfaces can also have default methods (methods with a body), static methods**,** and private methods (since Java 9).

- Interfaces define acontract that implementing classes must fulfill.
- Interfaces specify what a class should do, but not how it does it.

Syntax of Interface

```
interface Animal {
   void sound();        // Abstract method (implicitly public and abstract)
   int LEGS = 4;        // Constant variable (implicitly public, static, final)

   default void sleep() {   // Default method with implementation (Java 8+)
      System.out.println("Animal is sleeping");
   }
   static void info() {      // Static method (Java 8+)
      System.out.println("Animals interface");
   }
}
```

2.4.1 Implementing an Interface

A class implements an interface using the implements keyword and must provide implementations for all abstract methods.

```
class Dog implements Animal {
   public void sound() {
      System.out.println("Dog barks");
   }
}
```

Key Features of Interfaces

Feature	Description
All methods are abstract by default (before Java 8)	No method bodies unless default or static methods
Variables are public static final by default	Constants only, no instance variables
Supports multiple inheritance	A class can implement multiple interfaces
Cannot be instantiated	Interfaces only define behavior, not state
Default and static methods allowed (Java 8+)	Added flexibility in interface design
Private methods allowed (Java 9+)	To share code within interface methods

Why Use Interfaces?

- **Achieve Multiple Inheritance:** Java does not support multiple inheritance with classes but allows it through interfaces.
- **Define Contracts:** Specify methods that must be implemented, ensuring consistency across classes.
- **Decouple Code:** Promote loose coupling by programming to interfaces rather than implementations.
- **Support Polymorphism:** Objects can be referred to by their interface types, enabling flexible code.

Example: Multiple Interfaces

```
interface Printable {
   void print();
}
interface Showable {
   void show();
}
class Document implements Printable, Showable {
   public void print() {
      System.out.println("Printing document");
   }
   public void show() {
      System.out.println("Showing document");
   }
}
```

Rules and Restrictions

- A class must implement all abstract methods of an interface or be declared abstract.
- Interface methods are implicitly public.

- Interfaces cannot have constructors**.**
- Variables in interfaces are public static final by default.
- From Java 8 onwards, interfaces can have default and static methods.
- Interfaces can extend multiple interfaces**.**

Interface vs Abstract Class

Aspect	Interface	Abstract Class
Inheritance	Supports multiple inheritance	Single inheritance only
Methods	Only abstract, default, static methods	Abstract and concrete methods
Variables	Constants (public static final)	Instance variables allowed
Constructors	Not allowed	Allowed
Access Modifiers	All methods are public	Can have protected, private, etc.
Use Case	Define capabilities (e.g., Runnable)	Define base class with partial implementation

Real-world Uses

- **Runnable interface**: For defining tasks to be run by threads.
- **Comparable interface**: To provide sorting behavior.
- **Collection framework interfaces**: Like List, Set, Map.

Note:

- Interfaces are essential for designing flexible and extensible Java applications.
- They provide a way to achieve multiple inheritance.
- Promote loose coupling**,** code reusability, and polymorphism**.**
- Evolved to support default**,** static, and privatemethods for more powerful abstractions.

2.5 Packages

What is a Package?

A package in Java is a namespace that organizes classes, interfaces, enumerations, and sub-packages into a single group. It helps avoid naming conflicts and controls access.
Think of packages as folders/directories on your computer, grouping related files together.

2.5.1 Why Use Packages?

- **Namespace management**: Prevents class name conflicts by grouping classes logically.
- **Access control**: Classes in the same package can access each other's package-private members.
- **Code organization**: Makes large codebases manageable by grouping related classes.
- **Reusability**: Facilitates easier distribution and reuse of related classes.
- **Modularity**: Enables modular programming and better maintenance.

2.5.2 Types of Packages

Type	Description	Example
Built-in	Predefined packages provided by Java SDK	java.util, java.io, java.lang
User-defined	Created by developers to organize code	com.example.myapp, org.company.project

How to Declare a Package?

The package statement should be the first line(except comments) in a Java source file.

```
package com.example.myapp;

public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello from my package!");
   }
}
```

Importing Packages and Classes

To use classes from other packages, use the import statement.

```
import java.util.Scanner;   // Imports Scanner class only
import java.util.*;         // Imports all classes from java.util package
```

- Classes in the same package do not require import.
- Classes in java.lang (like String, System) are imported automatically.

Package Naming Conventions

- Use lowercase letters.
- Use your organization's internet domain name in reverse as a prefix to avoid name conflicts.
  Example: com.companyname.project
- Separate words with dots (.).
- Example package names:
  - com.google.maps
  - org.apache.commons

Access Control and Packages

Access Modifier	Same Class	Same Package	Subclass (any package)	Other Packages
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
default (no modifier)	Yes	Yes	No	No
Private	Yes	No	No	No

- Classes with default(no modifier) access are visible only within the same package.

2.5.3 Creating and Using Packages: Example

Folder structure:

```
src/
 └── com/
    └── example/
       ├── app/
       │   └── Main.java
       └── utils/
           └── Helper.java
```

Helper.java

```
package com.example.utils;
public class Helper {
   public static void greet() {
      System.out.println("Hello from Helper!");}}
```

Main.java

```
package com.example.app;
import com.example.utils.Helper;
public class Main {
   public static void main(String[] args) {
      Helper.greet();
}
}
```

Subpackages

- Packages can have subpackages to create a hierarchical structure.
- Subpackages are notconsidered part of the parent package.
- You must import subpackages explicitly.

2.5.4 Benefits of Using Packages

- Avoid class name collisions in large projects.
- Easier to maintain and locate files.
- Improves modularity and encapsulation.
- Facilitates sharing and reuse of code libraries.

Note:

- Packages are fundamental for organizing Java code.

- Provide a namespace mechanism and access control.
- Follow naming conventions based on reverse domain names.
- Essential for scalable and maintainable software development.

2.6 Summary

Binding in Java is the process of linking a method call to its actual method implementation. There are two types: static binding (early binding) and dynamic binding (late binding). Static binding happens at compile time and is used for private, static, and final methods, as well as method overloading, where the compiler determines which method to call based on the reference type. In contrast, dynamic binding occurs at runtime and is essential for method overriding and runtime polymorphism. It allows Java to determine the actual object's type during execution and call the appropriate overridden method, providing greater flexibility and extensibility. Dynamic binding supports polymorphism, making code more modular and maintainable, while static binding offers faster execution but less flexibility. The super keyword can be used in dynamic binding to explicitly call a superclass method, bypassing the dynamic dispatch. Dynamic binding enables the same method call to behave differently depending on the actual object, promoting code reuse, loose coupling, and scalable design. This feature is fundamental for many design patterns and frameworks, allowing behavior to change dynamically at runtime. Overall, dynamic binding enhances Java's object-oriented capabilities by supporting flexible, extensible, and reusable code

2.7 Key Terms

Binding**,** Static Binding (Early Binding)**,** Dynamic Binding (Late Binding) Polymorphism**,** Method Overriding**,** Method Overloading**,** Runtime Polymorphism**,** Compile Time**,** Runtime**,** super Keyword**,** Abstract Class**,** Abstract Method**,** Interface**,** Package**,** Access Modifiers.

2.8 Self-Assessment Questions

1. What is binding in Java?
2. What is the difference between static binding and dynamic binding?
3. Which types of methods use static binding?
4. How does dynamic binding support polymorphism?
5. What role does the super keyword play in dynamic binding?
6. Why can't abstract classes be instantiated?
7. What is the purpose of an abstract method?
8. How do interfaces differ from abstract classes?
9. What is the benefit of using packages in Java?
10. How does Java control access to classes and members within packages?

2.9 Further Readings

1. The Complete reference Java, Herbet Schildt, 7th Edition, McGraw Hill.
2. Java for Programmers, P.J.Deitel and H.M.Deitel, PEA (or) Java: How to Program, P.J.Deiteland H.M.Deitel,PHI
3. ObjectOrientedProgrammingthroughJava,P.RadhaKrishna,UniversitiesPress.

Dr. Kampa Lavanya

LESSON- 3

GUI DEVELOPMENT IN JAVA – EVENT HANDLING AND APPLETS

Aim and Objectives:

- To understand the structure and components of Java GUI using AWT and Swing.
- To learn and implement event handling using various listener interfaces.
- To develop interactive GUI applications responding to user actions.
- To understand the lifecycle and usage of Java Applets.
- To design and integrate GUI elements within applets for web-based Java applications.

STRUCTURE:

3.1.What is an Event in Java?

An **event** in Java is an object that describes a change in the state of a source. Events are generated when a user interacts with GUI components such as buttons, text fields, or windows.

Definition:

An **event** is an occurrence triggered by the user (like a mouse click or key press) or by the system (like a window closing), which can be handled by event listeners to execute some specific action.

Examples of Events:

User Action	Corresponding Event
Clicking a button	ActionEvent
Moving or clicking a mouse	MouseEvent
Pressing a key	KeyEvent
Selecting an item from a list	ItemEvent
Resizing a window	ComponentEvent

3.1.1 Event Class Hierarchy:

All event classes are part of the java.awt.event package and are subclasses of java.util.EventObject.

```
java.util.EventObject
  └── java.awt.AWTEvent
      ├── ActionEvent
      ├── MouseEvent
      ├── KeyEvent
      ├── WindowEvent
      └── ... and more
```

3.1.2 How Events Work in Java:

Java uses an **Event Delegation Model**, which involves three key components:

1. **Event Source** – The GUI component (e.g., button) that generates the event.
2. **Event Object** – Contains information about the event.
3. **Event Listener** – Interface that receives and handles the event.

Simple Example:

```
JButton button = new JButton("Click Me");

button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvente) {
System.out.println("Button was clicked!");
   }
});
```

In this example:

- The **button** is the event source.
- ActionEvent is the event object.
- ActionListener is the event listener handling the action.

What Happens at Runtime?

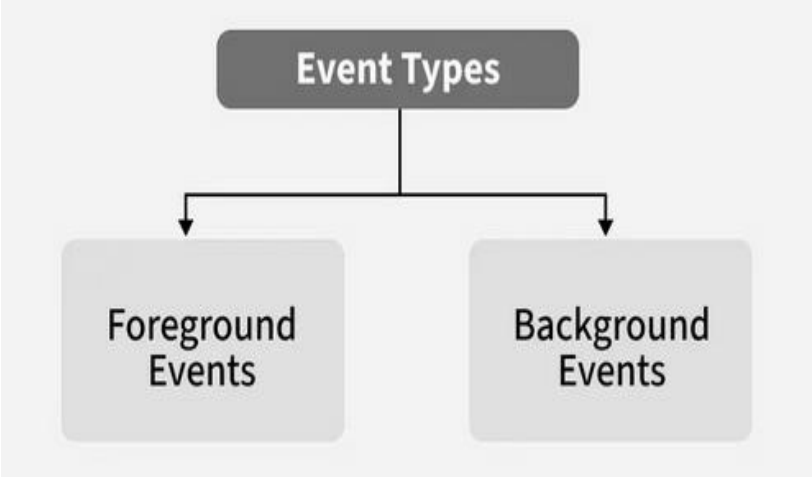
- A button labeled **"Click Me"** is created.

- An ActionListener is added to the button.
- When the user **clicks the button**, the actionPerformed() method is executed.

Console Output (After Button Click):

Button was clicked!

3.1.3 Classification of Events



Events in Java can be broadly classified into two categories based on how they are generated:

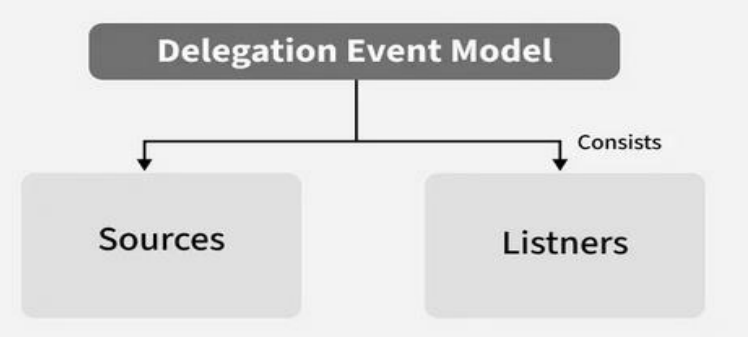
1. **Foreground Events:** Foreground events are the events that require user interaction to generate. Examples of these events include Button clicks, Scrolling the scrollbar, Moving the cursor, etc.

2. **Background Events:** Events that don't require interactions of users to generate are known as background events. Examples of these events are operating system failures/interrupts, operation completion, etc.

3.1.4.Event Handling Mechanism in Java

Event handling in Java enables programs to manage and respond to events effectively by specifying actions to be performed when an event occurs. Java employs the **Delegation Event Model** to handle these interactions, which is based on a clear separation of responsibilities between two key components:

1. **Source**: The source is the object that generates an event. Common event sources include GUI components such as buttons, checkboxes, lists, menu items, choices, scrollbars, text fields, and windows.
2. **Listener**: A listener is an object that waits for and responds to events generated by the source. Listeners are defined through interfaces, and each type of event has a corresponding listener interface responsible for handling it (e.g., ActionListener, ItemListener, MouseListener, etc.).



Registering the Source with a Listener

In Java, to enable event handling, the event source must be explicitly registered with an appropriate listener. This establishes a connection so that when the event occurs, the listener is notified and can respond accordingly.
Java provides specific methods for registering listeners depending on the type of event:

Syntax:

add<Type>Listener(listenerInstance)

Examples:

- addKeyListener() – used to register a listener for keyboard events (KeyEvent)
- addActionListener() – used to register a listener for action events like button clicks (ActionEvent)

3.1.5 Event Classes and Listener Interfaces

- Java offers a rich set of event classes and their corresponding listener interfaces as part of the java.awt.event and javax.swing.event packages. These classes and interfaces enable developers to handle different types of user and system-generated events effectively.
- The table below lists some of the most commonly used event classes along with their associated listener interfaces:
-

Event Class	Listener Interface	Description
ActionEvent	ActionListener	Handles action events like button clicks
ItemEvent	ItemListener	Handles item selection events in components like checkboxes and choices
KeyEvent	KeyListener	Handles keyboard key press, release, and type events
MouseEvent	MouseListener, MouseMotionListener	Handles mouse click, press, release, enter, exit, move, and drag events
MouseWheelEvent	MouseWheelListener	Handles mouse wheel rotation
WindowEvent	WindowListener	Handles window events like open, close, minimize, etc.

Event Class	Listener Interface	Description
FocusEvent	FocusListener	Handles focus gained or lost by a component
ComponentEvent	ComponentListener	Handles changes to a component's size, position, or visibility
AdjustmentEvent	AdjustmentListener	Handles scrollbar adjustments
TextEvent	TextListener	Handles changes in text components

3.1.6. Methods in Listener Interfaces

Each listener interface in Java defines one or more methods that must be implemented to handle specific types of events. These methods are automatically invoked when the corresponding event occurs, allowing developers to define custom behavior in response.
Below is a summary of some common listener interfaces and their associated methods:

Listener Interface	Methods	Purpose
ActionListener	actionPerformed(ActionEvent e)	Invoked when an action event occurs (e.g., button click)
ItemListener	itemStateChanged(ItemEvent e)	Invoked when an item is selected or deselected
KeyListener	keyPressed(KeyEvente)keyReleased(KeyEvente)keyTyped(KeyEvent e)	Handle keyboard key actions (press, release, and typing)
MouseListener	mouseClicked(MouseEvente)mouseEntered(MouseEvente)mouseExited(MouseEvente)mousePressed(MouseEvente)mouseReleased(MouseEvent e)	Handle mouse click and basic interactions
MouseMotionListener	mouseDragged(MouseEvente)mouseMoved(MouseEvent e)	Handle mouse movement and dragging
MouseWheelListener	mouseWheelMoved(MouseWheelEvent e)	Handles mouse wheel rotation
WindowListener	windowOpened(WindowEvente)windowClosing(WindowEvente)windowClosed(WindowEvente)windowIconified(WindowEvente)windowDeiconified(WindowEvente)windowActivated(WindowEvente)windowDeactivated(WindowEvent e)	Handle window state changes
FocusListener	focusGained(FocusEvente)focusLost(FocusEvent e)	Handle focus gained or lost by components
ComponentListener	componentResized(ComponentEvente)componentMoved(ComponentEvente)componentShown(ComponentEvente)componentHidden(ComponentEvent e)	Handle changes in component visibility, size, or position

3.1.7 Flow of Event Handling in Java

Event handling in Java follows a systematic process to manage user interactions. The typical flow involves the following steps:

1. **User Interaction**: An event is triggered when the user interacts with a GUI component (e.g., clicking a button or pressing a key).
2. **Event Object Creation**: After the event is generated, an object of the corresponding event class is automatically created. This object contains detailed information about the event and its source.
3. **Event Dispatch to Listener**: The event object is then passed to the appropriate method of the listener interface that has been registered to handle that event.
4. **Listener Method Execution**: The listener method processes the event and executes the defined actions or responses.

3.1.8 Approaches for Event Handling in Java

Java provides several approaches to handle events, giving developers flexibility in structuring their code. The most commonly used approaches are:

1. **Implementing Listener Interface in a Separate Class**
   - A dedicated class is created to implement the listener interface.
   - Keeps code modular and reusable.
   - Best suited for handling events from multiple components.

   Example:

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;

class MyEventHandler implements ActionListener {

   public void actionPerformed(ActionEvente) {

System.out.println("Button clicked from separate class!");

   }

}

public class SeparateClassExample {

   public static void main(String[] args) {

JFrame frame = new JFrame("Event Handling - Separate Class");

JButton button = new JButton("Click Me");

button.addActionListener(new MyEventHandler());  // Register listener
```

```
        frame.add(button);
        frame.setSize(300, 100);
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Input: Click the button

Output:

Button clicked from separate class!

2. Implementing Listener Interface in the Main Class
Example:

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

public class MainClassListenerExample extends JFrame implements ActionListener {
   JButton button;

MainClassListenerExample() {
      button = new JButton("Click Me");
button.addActionListener(this);  // Register listener

      add(button);
setSize(300, 100);
setLayout(new FlowLayout());
```

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
    }

    public void actionPerformed(ActionEvente) {
System.out.println("Button clicked from main class!");
    }

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

Input: Click the button

Output:

Button clicked from main class!

3. Using Anonymous Inner Class

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class AnonymousClassExample {
    public static void main(String[] args) {
JFrame frame = new JFrame("Event Handling - Anonymous Class");
JButton button = new JButton("Click Me");
button.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvente) {
System.out.println("Button clicked from anonymous class!");
        }
      });
frame.add(button);
frame.setSize(300, 100);
frame.setLayout(new FlowLayout());
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
   }
}
```

Input: Click the button

Output:

Button clicked from anonymous class!

4. Using Lambda Expression (Java 8+)

Example:

```
import java.awt.*;
import javax.swing.*;
public class LambdaExpressionExample {
   public static void main(String[] args) {
JFrame frame = new JFrame("Event Handling - Lambda Expression");
JButton button = new JButton("Click Me");
button.addActionListener(e ->System.out.println("Button clicked using lambda!"));
frame.add(button);
frame.setSize(300, 100);
```

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

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setVisible(true);

   }

}
```

Input: Click the button

Output:

Button clicked using lambda!

Table:

Approach	Pros	Cons
Separate Class	Clean separation of concerns	Requires more boilerplate code
Main Class	Easy to implement for small applications	Can clutter the main class
Anonymous Inner Class	Concise for one-time use	Less reusable
Lambda Expression (Java 8+)	Most concise and readable	Only works with functional interfaces

3.2.Applets

Java Applets

Java Applets were once a popular feature in early web applications. These small Java programs could run inside a web browser, providing interactive and dynamic content. Although applets are now obsolete, learning about them helps us understand the evolution of Java, especially its approach to graphics and GUI development.

Note: The java.applet package was deprecated starting from **Java 9**, as modern web technologies have replaced the need for applets.

3.2.1 What is a Java Applet?

A **Java Applet** is a program written in Java that is designed to be embedded within a web page and executed by a browser. Applets are included in HTML using the <applet> or <object> tags. They were commonly used to enhance web pages with multimedia, animations, and interactive features.

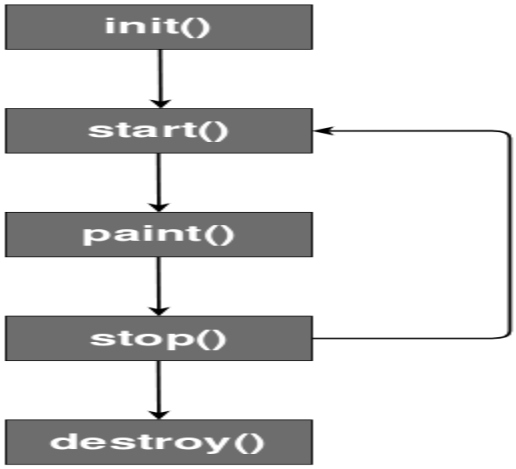
Applets operate in a **sandboxed environment** to ensure security, meaning they have restricted access to the local system.

Key Points about Applets

- **Applet Basics**: Every applet is a subclass of the java.applet.Applet class.
- **Not Standalone**: Applets cannot run independently like regular Java applications. They require a **web browser** or a **special tool called the Applet Viewer**, which is included with the Java Development Kit (JDK).
- **No main() Method**: Applets do not use the main() method to start execution. Instead, they rely on lifecycle methods such as init(), start(), and paint().
- **Displaying Output**: Unlike traditional programs that use System.out.println(), applets display content using graphical methods from the **Abstract Window Toolkit (AWT)**, such as drawString().

3.2.2.Java Applet Life Cycle

The below diagram demonstrates the life cycle of Java Applet:



3.2.3 Understanding the Lifecycle of an Applet

It is essential to understand the order in which the lifecycle methods of an applet are invoked.

When an applet starts, the following methods are called in sequence:

1. **init()** – This method is called once when the applet is first loaded. It is used for initialization, such as setting up user interface components or loading resources.
2. **start()** – Called after init(), and also each time the applet becomes active (e.g., when returning to the page containing the applet).
3. **paint(Graphics g)** – Invoked after start() to render the applet's output on the screen.

When an applet is terminated or no longer visible, the following methods are executed in order:

1. **stop()** – Called when the applet becomes inactive, such as when the user navigates away from the page.
2. **destroy()** – Invoked just before the applet is removed from memory. It is used to release resources and perform cleanup operations.

Key Packages for Java Applets

- java.applet.Applet: Base class for applets.
- java.awt.Graphics: Used for drawing on the applet screen.
- java.awt: Provides GUI components and event-handling mechanisms.

3.2.4 Creating Hello World Applet

To understand how applets work, let's create a simple **"Hello World"** applet. This example demonstrates the basic structure and the use of applet lifecycle methods.

Steps to Create a Hello World Applet:

1. **Import Required Packages**
   Import java.applet.Applet and java.awt.Graphics packages.
2. **Extend the Applet Class**
   Create a class that extends the Applet class.
3. **Override the paint() Method**
   Use the paint(Graphics g) method to display the output on the applet window.

Example Code:

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

/*
<applet code="HelloWorldApplet.class" width="300" height="100">
</applet>
*/

public class HelloWorldApplet extends Applet {

public voidpaint(Graphics g) {
g.drawString("Hello, World!", 100, 50);
   }
}
```

Explanation:

- The drawString() method of the Graphics class is used to display the text **"Hello, World!"** at the coordinates (100, 50).
- The HTML-style comment is used to embed the applet into a web page using the <applet> tag.

How to Run:

1. Save the file as HelloWorldApplet.java.
2. Compile it using:
3. javac HelloWorldApplet.java
4. Run it using the appletviewer tool:
5. appletviewer HelloWorldApplet.java

Note: Modern browsers no longer support applets. Applet-based programs are mainly run using the appletviewer tool or within Java-enabled development environments.

1. Using a Java-Enabled Web Browser

In the past, Java applets could be run directly within web browsers that had Java plugin support. To run a **Hello World Applet** this way, follow these steps:

Step 1: Create the Applet Source Code

Create a file named HelloWorldApplet.java:

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

public class HelloWorldApplet extends Applet {
public voidpaint(Graphics g) {
g.drawString("Hello, World!", 100, 50);
   }
}
```

Step 2: Compile the Applet

Open the command prompt or terminal and compile the Java file:

```
javac HelloWorldApplet.java
```

This will generate a HelloWorldApplet.class file.

Step 3: Create an HTML File

Create a file named HelloApplet.html and include the following content:

```
<html>
<head>
<title>Hello Applet</title>
</head>
<body>
<applet code="HelloWorldApplet.class" width="300" height="100">
      Your browser does not support Java Applets.
</applet>
</body>
</html>
```

Step 4: Open the HTML File in a Java-Enabled Browser

- Open the HelloApplet.html file in a Java-enabled browser (e.g., older versions of Internet Explorer, Netscape Navigator, or using a special plugin).
- The applet will be loaded, and the text **"Hello, World!"** will appear.

Important Note:
Most modern web browsers (like Chrome, Firefox, and Edge) have **removed support for Java Applets** due to security concerns. Java Applets are now considered outdated, and the recommended way to run them is using the **appletviewer** tool included with the JDK for testing and educational purposes.

2. Using appletviewer Tool

The appletviewer is a command-line utility provided by the JDK to run and test Java applets without needing a web browser.

Step 1: Write the Applet Code

Create a Java file named HelloWorldApplet.java:

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

public class HelloWorldApplet extends Applet {
public voidpaint(Graphics g) {
g.drawString("Hello, World!", 100, 50);
   }
}
```

Step 2: Add an HTML Applet Tag in Comments

Add the <applet> tag inside a block comment in the same .java file, like this:

```
/*
<applet code="HelloWorldApplet.class" width="300" height="100">
</applet>
*/
```

This allows the appletviewer tool to detect and use the tag directly from the source file.

Step 3: Compile the Applet

Open the terminal or command prompt and run:

```
javac HelloWorldApplet.java
```

This creates the compiled file HelloWorldApplet.class.

Step 4: Run the Applet Using appletviewer

Now run the applet using:

appletviewer HelloWorldApplet.java

The appletviewer will launch a window and display **"Hello, World!"** at the specified position.

Advantages of Using appletviewer

- No need for a web browser.
- Quick testing of applets during development.
- Works even though browsers have discontinued support for applets.

Note: Make sure you have the JDK installed and that the bin directory is included in your system's PATH variable to use javac and appletviewer.

3.3 Introduction to GUI in Java

A **Graphical User Interface (GUI)** in Java allows users to interact with programs visually through elements like buttons, text fields, menus, and windows rather than typing commands in a console.

Java provides two main libraries for creating GUIs:

1. **AWT (Abstract Window Toolkit)**
    - Part of Java's original GUI library (from JDK 1.0)
    - Platform-dependent (uses native OS components)
    - Basic GUI components like Button, Label, TextField, etc.
2. **Swing (javax.swing package)**
    - Built on top of AWT (introduced in JDK 1.2)
    - Platform-independent (pure Java implementation)
    - Provides rich, flexible GUI components like JButton, JTextField, JLabel, JTable, etc.
    - Supports pluggable look-and-feel, lightweight components, and more control over GUI design.

3.3.1 Key Features of Java GUI:

- **Event-driven**: Components respond to user actions (click, type, move).
- **Component-based**: GUI built using reusable components.
- **Customizable**: Layout managers allow flexible positioning of elements.

- **Thread-safe**: GUI updates should occur on the Event Dispatch Thread (EDT).

Common GUI Components in Swing:

Component	Description
JFrame	Main window
JButton	Button
JLabel	Display text or image
JTextField	Input single line text
JTextArea	Multi-line text input
JCheckBox, JRadioButton	Selection options

Example: Basic GUI with JFrame, JLabel, JTextField, and JButton

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

public class SimpleGUI {
   public static void main(String[] args) {
      // Create a frame
JFrame frame = new JFrame("Simple GUI Example");
frame.setSize(300, 200);
frame.setLayout(null);  // Using no layout manager
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a label
JLabel label = new JLabel("Enter your name:");
label.setBounds(20, 20, 120, 25);
frame.add(label);

      // Create a text field
JTextFieldtextField = new JTextField();
textField.setBounds(150, 20, 100, 25);
frame.add(textField);

      // Create a button
      JButton button = new JButton("Greet");
button.setBounds(100, 70, 80, 30);
frame.add(button);

      // Create a label to show the result
JLabelresultLabel = new JLabel("");
```

```
resultLabel.setBounds(20, 120, 250, 25);
frame.add(resultLabel);

      // Add event listener to button
button.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvente) {
             String name = textField.getText();
resultLabel.setText("Hello, " + name + "!");
          }
       });

      // Make frame visible
frame.setVisible(true);
   }
}
```

Expected Output:

When you run the program:

1. A window titled **"Simple GUI Example"** opens.
2. It contains:
    - A label: "Enter your name:"
    - A text field to input your name.
    - A button labeled "Greet".
    - An empty label where the greeting will appear.

When you **type a name** in the text field (e.g., Alice) and **click the "Greet" button**, the label at the bottom updates to:

Hello, Alice!

3.4 Summary

In Java, **event handling** is a mechanism that enables programs to respond to user interactions like mouse clicks, key presses, or window actions through a well-structured Event Delegation Model. Events are generated by sources such as buttons or text fields and are handled using listener interfaces like ActionListener, MouseListener, or KeyListener. Java provides several approaches to implement event handling, including separate listener classes, the main class, anonymous inner classes, and lambda expressions (Java 8+), each with its own advantages. Events are categorized into foreground (user-driven) and background (system-driven), and are managed using classes in the java.awt.event and javax.swing.event packages. In contrast, **Java applets** are small Java programs designed to run within a web browser, historically used to create interactive web content. Applets follow a defined lifecycle involving methods

like init(), start(), paint(), stop(), and destroy(), and rely on the java.applet and java.awt packages. Though now deprecated and unsupported in modern browsers, applets represent an important part of Java's evolution in GUI and web-based development. Java GUI allows users to interact visually using components like buttons and text fields, created using AWT or Swing libraries. Swing provides a rich, platform-independent set of components, supporting event-driven and customizable interfaces.

3.5 Key Terms

Event,Event Listener,ActionEvent,MouseEvent,KeyEvent,Event Delegation Model,Foreground Events,Background Events,addActionListener(),Anonymous Inner Class,Lambda Expression,AWT (Abstract Window Toolkit),Applet,paint(Graphics g),Applet Life Cycle,Swing, Event-driven programming.

3.6 Self-Assessment Questions

1. What is an event in Java, and how is it triggered?
2. Explain the Event Delegation Model in Java. What are its main components?
3. What is the purpose of an event listener in Java?
4. Differentiate between foreground and background events with examples.
5. What are the steps involved in the event handling flow in Java?
6. List any three event classes and their corresponding listener interfaces.
7. What are the different approaches to implement event handling in Java?
8. How does a lambda expression simplify event handling in Java 8 and above?
9. What is a Java Applet and how is it different from a standalone Java application?
10. Describe the life cycle methods of a Java Applet and their execution order.
11. What does it mean for a Java GUI application to be event-driven?

3.7 Further Readings

1. The Complete reference Java, Herbet Schildt, 7th Edition, McGraw Hill.
2. Java for Programmers, P.J.Deitel and H.M.Deitel, PEA (or) Java: How to Program ,P.J.DeitelandH.M.Deitel,PHI
3. ObjectOrientedProgrammingthroughJava,P.RadhaKrishna,UniversitiesPress.

Dr. Kampa Lavanya

LESSON- 4

SWING FRAMEWORK AND LAYOUT MANAGEMENT

Aim and Objectives:

- Understand the fundamentals of Swing and how it differs from AWT in terms of architecture, performance, and flexibility.
- Explore the MVC (Model-View-Controller) architecture followed by Swing and how it separates data, UI, and control logic.
- Identify the hierarchy of Swing components and comprehend the role of various top-level containers like JFrame, JApplet, JWindow, and JDialog.
- Gain practical knowledge of container classes such as JPanel and how they help in organizing Swing components.
- Develop a simple Swing-based GUI application using standard components like buttons, labels, text fields, and message dialogs.
- Learn various layout managers—BorderLayout, GridLayout, FlowLayout, and BoxLayout—for effective component arrangement in Swing interfaces.

STRUCTURE:

4.1What is Swing in Java?

Swing is a Graphical User Interface (GUI) toolkit for Java that is part of the Java Foundation Classes (JFC). It is used to create window-based applications and provides a rich set of lightweight, platform-independent components.

Swing is built on top of the Abstract Window Toolkit (AWT) but offers more powerful and flexible components such as buttons, tables, lists, menus, and more. It also supports features like pluggable look-and-feel, double buffering, and drag-and-drop functionality.

4.2 Introduction to Swing

- **Swing** was introduced by Sun Microsystems to address the limitations of AWT and offer a more robust GUI development framework.
- It is defined in the javax.swing package.
- Swing components are written entirely in Java, making them platform-independent and lightweight (they do not rely on native OS peers).
- Swing supports the Model-View-Controller (MVC) architecture, which separates the logic (model), the interface (view), and the user interaction (controller).
- It allows developers to create customized, consistent, and interactive user interfaces.
- Example components: JFrame, JButton, JLabel, JTextField, JPanel, JTable, JTree, etc.

Advantages of Swing

- Platform independent
- Rich set of customizable GUI components
- Consistent look-and-feel across platforms
- Fully object-oriented
- Supports MVC architecture
- Integrated drag-and-drop support
- **Disadvantages of Swing**
- Slower than native GUI frameworks
- Can be more memory-intensive
- Learning curve for advanced components

4.3 Swing vs. AWT

Swing vs. AWT in Java

Feature	AWT (Abstract Window Toolkit)	Swing
Package	java.awt	javax.swing
Component Type	Heavyweight (uses native OS peers)	Lightweight (pure Java components)
Look and Feel	OS-dependent	Pluggable look and feel
Custom Components	Hard to customize	Easy to customize
Advanced Components	Limited (no JTable, JTree, etc.)	Rich set (JTable, JTree, JTabbedPane, etc.)
MVC Architecture	Not strictly followed	Follows MVC architecture
Thread Safety	Partially thread-safe	More thread-safe
Performance	Faster on native components	Slightly slower due to more features

Example Programs

1. AWT Example Program

```
import java.awt.*;
import java.awt.event.*;
public class AWTExample {
   public static void main(String[] args) {
      Frame frame = new Frame("AWT Example");
      Label label = new Label("Enter your name:");
      label.setBounds(50, 50, 120, 30);
      TextField textField = new TextField();
      textField.setBounds(180, 50, 150, 30);
      Button button = new Button("Greet");
      button.setBounds(100, 100, 100, 30);
      Label output = new Label();
      output.setBounds(50, 150, 250, 30);
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            output.setText("Hello, " + textField.getText());
         }
      });
      frame.add(label);
      frame.add(textField);
```

```
        frame.add(button);
        frame.add(output);
        frame.setSize(400, 250);
        frame.setLayout(null);
        frame.setVisible(true);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                frame.dispose();
            }
        });
    }
}
```

Input:

- User types John in the text field
- Clicks the **Greet** button

Output:

- Label displays: Hello, John

2. Swing Example Program

```
import javax.swing.*;
import java.awt.event.*;
public class SwingExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Example");

        JLabel label = new JLabel("Enter your name:");
        label.setBounds(50, 50, 120, 30);

        JTextField textField = new JTextField();
        textField.setBounds(180, 50, 150, 30);

        JButton button = new JButton("Greet");
        button.setBounds(100, 100, 100, 30);

        JLabel output = new JLabel();
```

```
        output.setBounds(50, 150, 250, 30);

        button.addActionListener(e -> {
            output.setText("Hello, " + textField.getText());
        });

        frame.add(label);
        frame.add(textField);
        frame.add(button);
        frame.add(output);


        frame.setSize(400, 250);
        frame.setLayout(null);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Input:

- User types Anita in the text field
- Clicks the Greet button

Output:

- Label displays: Hello, Anita

Note:

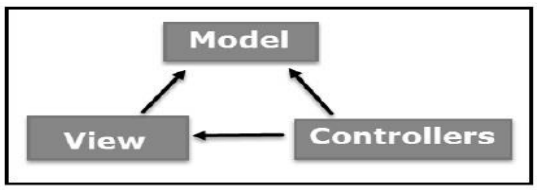
- **AWT** is simpler but less flexible, uses native GUI components, and is platform-dependent.
- **Swing** provides a modern and consistent UI, with more controls and features suitable for real-world applications.

4.4 MVC architecture

MVC stands for Model-View-Controller**.** It is a design pattern that separates the application logic into three interconnected components. This separation helps manage complexity and promotes organized and scalable code in GUI applications like those built using Java Swing**.**

MVC Components

Following are the components of MVC



4.4.1 Components of MVC:

1. Model

- **Definition**: The Model represents the data and the business logic of the application.
- **Responsibilities**:
  - Manages application state and data.
  - Performs calculations and database interactions.
  - Notifies the View of data changes.

2. View

- **Definition**: The View is the **UI (User Interface)** component that displays the data.
- **Responsibilities**:
  - Renders the data from the model to the user.
  - Provides interface elements like buttons, labels, and text fields.
  - Updates when the model changes.

3. Controller

- **Definition**: The Controller handles user inputs and controls application flow.
- **Responsibilities**:
  - Interprets user actions (e.g., button clicks).
  - Updates the model or changes the view accordingly.

Flow of MVC Architecture:

1. **User interacts with the View** (e.g., clicks a button).
2. **Controller handles the input** and makes changes to the Model.
3. **Model updates its state** and notifies the View.
4. **View refreshes** to display updated data.

Advantages of MVC:

- **Separation of concerns**: Keeps UI, data, and logic separate.
- **Maintainability**: Easier to manage and update code.
- **Reusability**: Components can be reused independently.
- **Scalability**: Suitable for complex applications.

Disadvantages of MVC Architecture

- The structure can be hard to understand and modify, making it challenging to read, test, and reuse the code effectively.

- Not ideal for small-scale applications, as the overhead of implementing MVC may outweigh its benefits.
- The View layer has limited direct access to data, which can reduce efficiency in data display and interaction.
- Navigation within the framework can be complex, due to multiple layers and the need to understand MVC's separation of concerns.
- The architecture adds additional complexity, which can lead to inefficiencies, especially during data updates and application scaling.

4.4.2 Working of the MVC framework with Example

The **Model-View-Controller (MVC)** framework is a design pattern that separates an application into three interconnected components to organize code efficiently, especially in large applications.

Step-by-Step Working of MVC

1. **User Interaction with the View**
   - The user interacts with the UI (e.g., clicks a button, enters text).
   - These interactions are captured by the View.
2. **Controller Handles User Input**
   - The Controller receives the input event from the View.
   - It interprets the action and determines what needs to change in the application.
3. **Controller Updates the Model**
   - The Controller modifies the Model based on the user input.
   - The Model performs business logic or data processing (like updating a value or retrieving data from a database).
4. **Model Changes the State**
   - The Model updates its internal state or data.
   - It may notify the View about the change (if an observer or event mechanism is used).
5. **View Updates the Interface**
   - The View reads the updated data from the Model.
   - It then refreshes the user interface to reflect the new state.

Summary of Responsibilities

Component	Role
Model	Manages data and business logic
View	Displays data and interacts with the user
Controller	Handles user actions and updates Model & View

Example Use Case (Counter App):

- User clicks “Increment” (View)

- Controller receives the click event and calls increment() on Model
- Model updates count value
- View reads new value from Model and displays updated count.

Popular MVC Frameworks

The Model-View-Controller (MVC) architecture is widely adopted in web and application development. Below are some of the most commonly used and well-known MVC frameworks:

- **Ruby on Rails** – A powerful framework written in Ruby, known for its simplicity and speed.
- **Django** – A high-level Python framework that promotes rapid development and clean design.
- **CherryPy** – A minimalist Python framework that allows developers to build web applications quickly.
- **Spring MVC** – A robust, enterprise-level Java framework, part of the Spring ecosystem.
- **Catalyst** – A Perl-based MVC framework that is flexible and scalable.
- **Rails** – Often synonymous with Ruby on Rails, it's a popular choice for rapid web application development.
- **Zend Framework** – A PHP-based framework designed for building secure and modern web applications.
- **FuelPHP** – A simple, flexible PHP framework with support for HMVC (Hierarchical MVC).
- **Laravel** – A widely-used PHP framework that emphasizes elegance, simplicity, and readability.
- **Symfony** – A mature PHP framework known for its reusable components and large developer community.

4.5 Hierarchy for Swing components

Swing components in Java are built upon the Java Foundation Classes (JFC) and follow a class hierarchy that extends from AWT (Abstract Window Toolkit). Swing provides a richer set of GUI components than AWT.

Top-Level Hierarchy

```
java.lang.Object
  └── java.awt.Component
      └── java.awt.Container
          └── javax.swing.JComponent
```

Main Categories of Swing Components

1. **Top-Level Containers** – The root containers for any Swing GUI.
    - JFrame – Main window
    - JDialog – Popup window

- JWindow – Undecorated window
- JApplet – Applet container

2. **Intermediate Containers** – Used to hold and organize other components.
   - JPanel – Generic lightweight container
   - JScrollPane, JSplitPane, JTabbedPane – Specialized containers
3. **Atomic Components (Controls)** – Basic UI elements.
   - **Labels and Text:**
     - JLabel, JTextField, JTextArea, JPasswordField
   - **Buttons:**
     - JButton, JToggleButton, JRadioButton, JCheckBox
   - **Menus:**
     - JMenuBar, JMenu, JMenuItem
   - **Lists and Tables:**
     - JList, JComboBox, JTable, JTree
   - **Spinners and Sliders:**
     - JSpinner, JSlider
   - **Others:**
     - JProgressBar, JToolBar, JSeparator, JFileChooser

Hierarchy Diagram (Simplified)

```
Object
 └── Component (AWT)
    └── Container (AWT)
       └── JComponent (Swing)
          ├── JPanel
          ├── JLabel
          ├── JButton
          ├── JTextField
          ├── JTable
          ├── JComboBox
          └── etc.
```

Top-Level Containers (not under JComponent but under Container):

```
Object
 └── Component
    └── Container
       ├── Window
       │   ├── Frame → JFrame
       │   ├── Dialog → JDialog
       │   └── Window → JWindow
       └── Applet → JApplet
```

Key Points

- All Swing components are lightweight and extend from JComponent.
- Swing is built on top of AWT but provides a pluggable look-and-feel.
- The hierarchy enables reusability and flexibility of components.
- Containers can hold other components and manage their layout using layout managers.

4.6 Containers

Containers in Swing

A **container** is a special component in Java that can hold and organize other components (like buttons, labels, text fields) in a graphical user interface.

4.6.1 Types of Containers: Swing provides several types of containers, divided into two categories:

1. Top-Level Containers

These are the root containers for any Swing application. Every Swing GUI must start with a top-level container.

🔷 JFrame

- Represents a main window with title bar, borders, buttons, etc.
- Commonly used in desktop applications.

🔷 JDialog

- A pop-up window, typically used for alerts, messages, or input dialogs.
- Can be modal or non-modal.

🔷 JWindow

- A borderless window used for splash screens or tool tips.
- No title bar or controls.

🔷 JApplet (Deprecated)

- Used for embedding Java applications in web browsers (not commonly used now).

2. Intermediate (Lightweight) Containers

These are used to group and arrange components inside top-level containers.

🔶 JPanel

- A general-purpose lightweight container.

- Often used to group related components.
- Can be nested and combined with layout managers.

◆ JScrollPane

- Provides a scrollable view of another component.

◆ JSplitPane

- Divides two components horizontally or vertically.

◆ JTabbedPane

- Lets users switch between tabs, each containing a different component.

◆ JLayeredPane, JDesktopPane

- Used for more advanced UI structures like internal frames.

Why Use Containers?

- Containers organize components using layout managers.
- They enable nesting and structuring of complex GUIs.
- Containers make the UI modular and reusable.

Example: Using a Container (JFrame + JPanel)

```
import javax.swing.*;
public class MyFrame {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Container Example");
      JPanel panel = new JPanel();
      JButton button = new JButton("Click Me");
      panel.add(button);
      frame.add(panel);
      frame.setSize(300, 150);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Output: A window (JFrame) with a panel (JPanel) and a button inside.

4.7 JFrame

- JFrame is a top-level container used to create a main application window.
- It includes features like a title bar, close/minimize/maximize buttons, and menu bars.

- Commonly used in desktop GUI applications.

Example: JFrame frame = new JFrame("My Window");

4.8 JApplet(Deprecated)

- JApplet is a top-level container used to embed Java applications in web browsers.
- Extends Applet and provides support for Swing components.
- Rarely used now, as applets are outdated and no longer supported in modern browsers.

Example: public class MyApplet extends JApplet { ... }

4.9 JWindow

- JWindow is a borderless top-level window with no title bar or control buttons.
- Useful for splash screens, tooltips, or popups.
- Cannot be resized or closed by the user.

Example: JWindow window = new JWindow();

4.10 JDialog

- JDialog is a pop-up window used for short interactions like messages, warnings, or form inputs.
- Can be modal (blocks other windows until closed) or non-modal.

Example: JDialog dialog = new JDialog(frame, "Message", true);

4.11 JPanel

- JPanel is a lightweight container used to group components inside a window.
- Often used with layout managers to organize buttons, labels, and other UI elements.
- Cannot exist independently—must be added to a top-level container like JFrame.

Example: JPanel panel = new JPanel();

4.12 A simple swing application

A basic Swing application can display a "Hello World" message inside a window. This example shows how to create a JFrame as the main window, add a JLabel with the message, and make the window visible.

```
import javax.swing.JFrame;

import javax.swing.JLabel;
public class SimpleSwingApp {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Simple Swing App");
```

```
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 200);
    JLabel label = new JLabel("Hello World");
    frame.add(label);
    frame.setVisible(true);
  }}
```

Explanation:

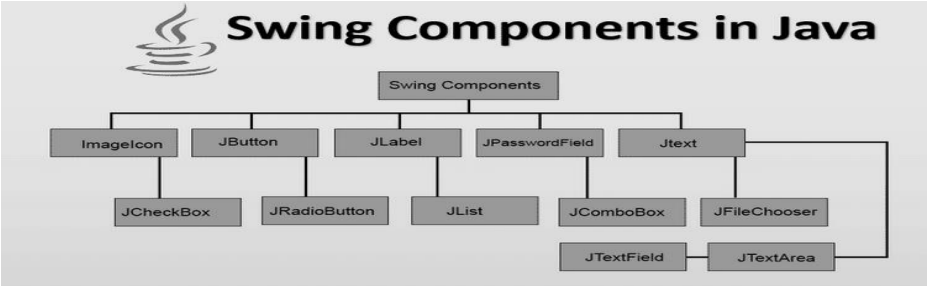
1. **Import necessary classes:**
   javax.swing.JFrame is used to create the main application window, and javax.swing.JLabel is used to display text.
2. **Create a JFrame:**
   This is the main window of the application.
3. **Set window properties:**
   - setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) ensures the application closes when the window's close button is clicked.
   - setSize(300, 200) sets the window dimensions.
4. **Create a JLabel:**
   This component holds and displays the "Hello World" text.
5. **Add the label to the frame:**
   The label is added to the frame's content area.
6. **Make the frame visible:**
   Calling setVisible(true) displays the window on the screen.

This simple program serves as a foundation for creating more complex Swing applications by adding additional components and event handling.

4.13 Overview of several swing components

Swing components form the core elements of Java-based graphical user interfaces (GUIs). They include interactive elements like buttons, text fields, labels, and more, which are essential for building user-friendly applications. Compared to the older AWT (Abstract Window Toolkit), Swing offers a more versatile and feature-rich set of components with greater flexibility and customization.

Below is a brief overview of commonly used Swing components:



Swing Components

Swing provides a comprehensive set of components for building interactive and visually appealing Java GUI applications. These components are categorized into interactive and non-interactive elements and are supported by powerful features like customization, lightweight design, and event handling.

Interactive Components

- **JButton:** Triggers an action when clicked.
- **JCheckBox:** Allows users to select or deselect one or more independent options.
- **JRadioButton:** Lets users choose one option from a group (mutually exclusive).
- **JTextField / JTextArea:** Enables users to input and display single-line or multi-line text.
- **JList:** Displays a scrollable list of selectable items.
- **JComboBox:** Offers a dropdown list from which users can choose a single item.

Non-Interactive Components

- **JLabel:** Displays static text or images.
- **ImageIcon:** Loads and shows images within the GUI.
- **JPanel:** Acts as a container to group and organize components within the interface.

Container Components

- **JFrame:** The main top-level window in a Swing application.
- **JScrollPane:** Adds scroll bars to a component when its content exceeds the visible area.
- **JSplitPane:** Splits a window into two resizable sections (horizontal or vertical).

Key Features of Swing Components

- **Pluggable Look and Feel:** Components can adopt different styles to match various operating systems or custom themes.
- **Lightweight Components:** Pure Java implementation ensures platform independence and better performance.
- **Event-Driven Programming:** Components can respond dynamically to user actions such as clicks, typing, and mouse movements.
- **MVC Architecture:** Swing follows the Model-View-Controller pattern, promoting separation of data, UI, and control logic.

Swing empowers developers to create flexible, efficient, and modern desktop applications by combining these components with powerful features and customization options.

4.14 Layout management

Layout management in Swing refers to the way components (like buttons, labels, and text fields) are arranged within a container (like JFrame or JPanel). Swing provides several layout managers to control the positioning and sizing of components automatically, making the interface more flexible and adaptable across different screen sizes and resolutions.

Advantages:

- Automatically adjusts components for different screen sizes.
- Reduces manual effort in positioning components.
- Makes GUI development easier and more organized.

Note:

If more control is needed, developers can also set layout to null and use absolute positioning with setBounds(), but this is generally discouraged for real-world applications.

4.15 Layout manager types

The Layout managers enable us to control the way in which visual components are arranged in the GUI forms by determining the size and position of components within the containers.

4.15.1 Border Layout

Description: Divides the container into five regions: NORTH, SOUTH, EAST, WEST, and CENTER.

Default for: JFrame

Use case: Organizing content by position.

Constructors of BorderLayout class:

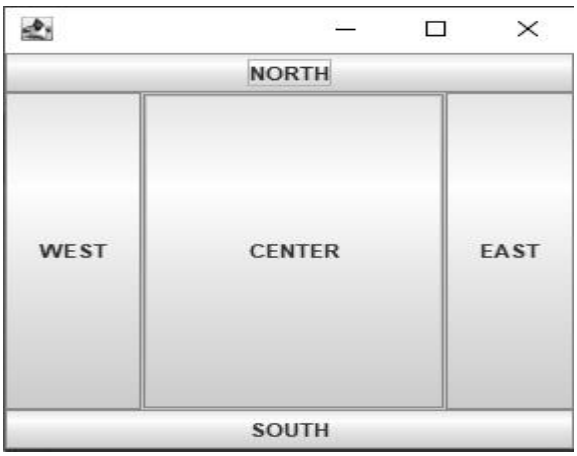
- **BorderLayout():** creates a border layout but with no gaps between the components.
- **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components

Example:

```
import java.awt.*;

import javax.swing.*;

public class Border

{

   JFrame f;

   Border()

   {

      f = new JFrame();

      // creating buttons
```

```
        JButton b1 = new JButton("NORTH");;
        JButton b2 = new JButton("SOUTH");;
        JButton b3 = new JButton("EAST");;
        JButton b4 = new JButton("WEST");;
        JButton b5 = new JButton("CENTER");;
        f.add(b1, BorderLayout.NORTH);
        f.add(b2, BorderLayout.SOUTH);
        f.add(b3, BorderLayout.EAST);
        f.add(b4, BorderLayout.WEST);
        f.add(b5, BorderLayout.CENTER);
        f.setSize(300, 300);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new Border();
    }
}
```

Output:



4.15.2 Grid Layout

Description: Arranges components in a grid of equally sized rows and columns.

Use case: Uniformly spaced component layout.

Constructors of GridLayout class

1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

Example:

```
import java.awt.*;

import javax.swing.*;

public class GridLayoutExample

{

   JFrame frameObj;

   GridLayoutExample()

   {

      frameObj = new JFrame();

      JButton btn1 = new JButton("1");

      JButton btn2 = new JButton("2");

      JButton btn3 = new JButton("3");

      JButton btn4 = new JButton("4");

      JButton btn5 = new JButton("5");

      JButton btn6 = new JButton("6");

      JButton btn7 = new JButton("7");

      JButton btn8 = new JButton("8");

      JButton btn9 = new JButton("9");
```

```
        frameObj.add(btn1); frameObj.add(btn2); frameObj.add(btn3);
        frameObj.add(btn4); frameObj.add(btn5); frameObj.add(btn6);
        frameObj.add(btn7); frameObj.add(btn8); frameObj.add(btn9);
        frameObj.setLayout(new GridLayout());
        frameObj.setSize(300, 300);
        frameObj.setVisible(true);
    }
    public static void main(String argvs[])
    {
        new GridLayoutExample();
    }
}
```

Output:

4.15.3 Flow Layout

Description: Arranges components in a single row, left to right. Wraps to the next line when necessary.

Default for: JPanel

Use case: Simple horizontal or vertical layouts.

Fields of FlowLayout class

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER

4. public static final int LEADING
5. public static final int TRAILING

Constructors of FlowLayout class

1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example:

```
import java.awt.*;

import javax.swing.*;

public class FlowLayoutExample

{

   JFrame frameObj;

   FlowLayoutExample()

   {

      frameObj = new JFrame();

      JButton b1 = new JButton("1");

      JButton b2 = new JButton("2");

      JButton b3 = new JButton("3");

      JButton b4 = new JButton("4");

      JButton b5 = new JButton("5");

      JButton b6 = new JButton("6");

      JButton b7 = new JButton("7");

      JButton b8 = new JButton("8");

      JButton b9 = new JButton("9");

      JButton b10 = new JButton("10");
```

```
        frameObj.add(b1); frameObj.add(b2); frameObj.add(b3); frameObj.add(b4);
        frameObj.add(b5); frameObj.add(b6);  frameObj.add(b7);  frameObj.add(b8);
        frameObj.add(b9);  frameObj.add(b10);
        frameObj.setLayout(new FlowLayout());
        frameObj.setSize(300, 300);
        frameObj.setVisible(true);
    }
    public static void main(String argvs[])
    {
        new FlowLayoutExample();
    }
}
```

Output:

4.15.4 Box Layout

Description: Arranges components either **vertically** (Y-axis) or **horizontally** (X-axis).

Use case: Stacking components in a single direction.

For this purpose, the BoxLayout class provides four constants. They are as follows:

Fields of BoxLayout Class

1. public static final int X_AXIS: Alignment of the components are horizontal from left to right.

2. public static final int Y_AXIS: Alignment of the components are vertical from top to bottom.
3. public static final int LINE_AXIS: Alignment of the components is similar to the way words are aligned in a line,
4. public static final int PAGE_AXIS: Alignment of the components is similar to the way text lines are put on a page,

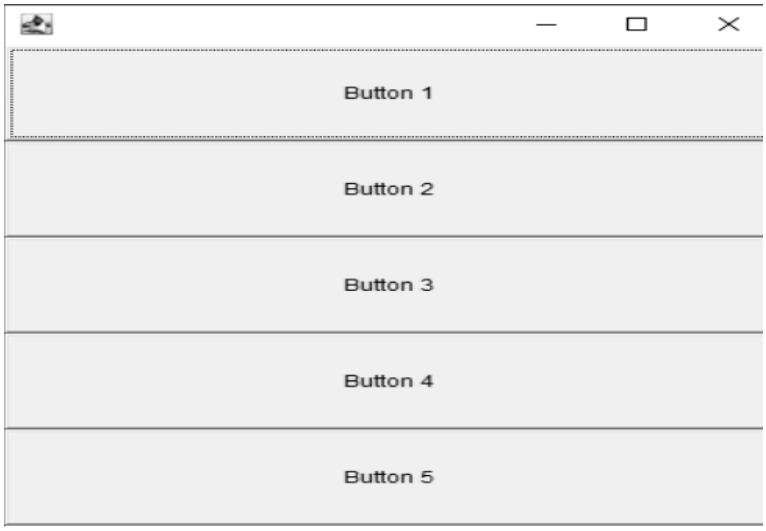
Constructor of BoxLayout class

1. **BoxLayout(Container c, int axis):** creates a box layout that arranges the components with the given axis.

Example:

```
import java.awt.*;
import javax.swing.*;
class BoxLayoutExample1 extends Frame {
   Button buttons[];
   public BoxLayoutExample1 () {
      buttons = new Button [5];
      for (int i = 0;i<5;i++) {
         buttons[i] = new Button ("Button " + (i + 1));
         add (buttons[i]);
      }
      setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
      setSize(400,400);
      setVisible(true);
   }
   public static void main(String args[]){
      BoxLayoutExample1 b=new BoxLayoutExample1();
   }
}
```

Output:



4.16 SUMMARY

Swing is a powerful part of Java's GUI toolkit called Java Foundation Classes (JFC), used to create rich window-based applications. It is built on top of AWT but offers more components and greater flexibility. Unlike AWT, which uses native GUI elements, Swing is fully written in Java, making it lightweight and platform-independent. Swing follows the Model-View-Controller (MVC) architecture, separating data, user interface, and user interaction logic for better structure and maintenance. The javax.swing package includes various useful classes like JFrame, JButton, JLabel, and JTextField. Swing provides a uniform look and feel across different platforms and allows for customized appearances through pluggable look-and-feel settings. It supports advanced features like drag-and-drop and rich component behavior. Although Swing can be slower than native toolkits, it is more customizable and flexible than AWT. Its component hierarchy starts with java.awt.Component and extends through javax.swing.JComponent. Swing components fall into top-level, intermediate, and atomic categories, such as JFrame, JPanel, and JButton. Containers like JPanel and JDialog are crucial for layout and component organization. Swing also integrates layout managers to control component placement. The MVC pattern, while powerful, can be complex for small applications. Despite being older, Swing is still widely used for desktop apps. Its extensive features, object-oriented design, and platform independence make it suitable for building sophisticated user interfaces.

4.17 Key Terms

Swing**,** AWT**,** JFC (Java Foundation Classes)**,** javax.swing**,** Lightweight components**,** Pluggable Look-and-Feel**,**MVC(Model-View-Controller)**,**JFrame**,**JPanel**,**JComponent**,**Top-Level Container, Atomic Components**,** Layout Managers**,** Event Handling**,** Swing vs. AWT

4.18 Self-Assessment Questions

1. What is Swing in Java?
2. How does Swing differ from AWT?
3. What is the role of JFC in Swing?
4. Which package contains all Swing classes?
5. What are lightweight components in Swing?
6. What is a pluggable look-and-feel in Swing?
7. Explain the MVC architecture used in Swing.
8. What is the purpose of JFrame?
9. Name two top-level containers in Swing.
10. What is the function of layout managers in Swing?

4.19 Further Readings

1. The Complete reference Java, Herbet Schildt, 7th Edition, McGraw Hill.
2. Java for Programmers, P.J.Deitel and H.M.Deitel, PEA (or) Java: How to Program , P.J.Deiteland H.M.Deitel,PHI
3. ObjectOrientedProgrammingthroughJava,P.RadhaKrishna,UniversitiesPress.
4. "Java: The Complete Reference" by Herbert Schildt
   (Latest Edition, McGraw-Hill Education) – Chapter on Swing Components and GUI Design

Dr. Kampa Lavanya

LESSON- 5

BUILDING WEB PAGES WITH HTML AND CSS

Aim and Objectives:

- Understand the basic structure and purpose of HTML for web page development.
- Identify and apply common HTML tags to create lists, tables, and insert images.
- Design and implement user input forms using appropriate HTML form elements.
- Utilize HTML frames to organize content into multiple sections within a browser window.
- Apply Cascading Style Sheets (CSS) to enhance the presentation and layout of web pages.

STRUCTURE:

5.1 HTML

HTML Overview and Evolution

HTML (HyperText Markup Language) structures a document by dividing its content into elements. These elements are generally categorized into two types:

1. Elements that control how the content in the BODY of the document is displayed by the web browser.

2. Elements that provide metadata about the document, such as its title or links to other resources.

The rules and syntax of HTML (including the names and usage of tags) are based on a more complex language known as SGML (Standard Generalized Markup Language). SGML was originally designed for managing large document collections and is quite sophisticated. Thankfully, HTML is a much simpler subset that suits the needs of web developers.

Despite its simplicity, HTML lacks some powerful features of SGML. To bridge this gap, XML (eXtensible Markup Language) was introduced—retaining the simplicity of HTML while incorporating the flexibility of SGML.

5.1.1 History of HTML

HTML was first created by Tim Berners-Lee at CERN and gained popularity through the **Mosaic** browser developed at NCSA. As the Web grew rapidly in the 1990s, HTML evolved significantly with various enhancements and updates.

To maintain consistency and compatibility among web browsers and developers, standardized versions of HTML were introduced.

- **HTML 2.0** (released in November 1995) standardized the common usage practices of the time.
- **HTML 3.0** (1995) aimed to support more complex web design features but was not widely adopted due to implementation challenges.

Maintaining compatibility across browsers helps reduce development costs and prevents the fragmentation of the Web into proprietary, incompatible systems—ensuring a more unified and accessible experience for all users.

HTML's Future Vision

HTML has always been developed with a vision of universal access. It is designed to support a wide range of devices and platforms—ranging from PCs with high-resolution displays to mobile phones, voice-based devices, and even low-bandwidth systems—ensuring that web content remains accessible and functional across diverse environments.

Advantages of HTML:

1. **Widely Adopted:** HTML is a universally accepted standard for creating web pages, making it highly reliable and popular.
2. **Cross-Browser Support:** All major web browsers support HTML, ensuring that websites display consistently across platforms.
3. **User-Friendly:** HTML is easy to learn, write, and understand, even for beginners in web development.
4. **No Cost or Special Software:** HTML editing requires only a basic text editor and a web browser, both of which are typically pre-installed on most systems.

Disadvantages of HTML:

1. **Static Content Only:** HTML is limited to creating static pages; it cannot handle dynamic functionality without additional technologies like JavaScript or server-side scripting.

2. **Verbose Coding:** Even simple designs may require writing large amounts of repetitive code.
3. **Weak Security:** HTML lacks built-in security mechanisms, making it dependent on other technologies for secure web applications.
4. **Code Complexity for Larger Pages:** Managing and organizing lengthy HTML code can become complex and error-prone as the size of the webpage increases.

Important Points about HTML:

- HTML tags are enclosed within angle brackets (e.g., <tag>).
- Tags are not case-sensitive; for example, <head>, <HEAD>, and <Head> are treated the same.
- If a browser does not recognize a tag, it typically ignores it without error.
- Certain special characters must be represented using escape sequences (e.g., < for <, > for >).
- Whitespace, tabs, and newlines are generally ignored by the browser when rendering the content.

Structure of an HTML Document

All HTML documents follow a standard structure. The root element is <html>, which contains two main sections: <head> and <body>.

- The <head> tag includes metadata and control information for the browser.
- The <body> tag holds the actual content that is displayed to the user.

Below is a basic example of an HTML document:

```
<html>
  <head>
    <title>Basic HTML Document</title>
  </head>
  <body>
    <h1>Welcome to the World of Web Technologies</h1>
    <p>A sample HTML program</p>
  </body>
</html>
```

In this example:

- The <title> tag (inside <head>) specifies the text shown in the browser's title bar.
- The <h1> tag is used for main headings. It renders the text in bold and a larger font size.
- The <p> tag is used to define a paragraph of text.

HTML Comments

Comments in HTML start with <!-- and end with -->.
They can span multiple lines, and are ignored by the browser. For example:

<!-- This is a comment
that spans multiple lines -->

Avoid using double hyphens (--) inside the comment text, as it may cause errors in some browsers.

5.2 Common Tags

Common HTML Tags:

1. <html> – Root element that defines the entire HTML document.
2. <head> – Contains metadata and links to external resources like stylesheets and scripts.
3. <title> – Specifies the title of the web page (appears in the browser tab).
4. <body> – Contains the main content of the webpage that is visible to users.
5. <h1> **to** <h6> – Heading tags, where <h1> is the largest and <h6> is the smallest.
6. <p> – Defines a paragraph.
7.
 – Inserts a line break.
8. <hr> – Inserts a horizontal line (thematic break).
9. <a> – Creates a hyperlink. Example: Visit
10. <img> – Embeds an image. Example: <img src="image.jpg" alt="Image">
11. – Creates an unordered (bulleted) list.
12. – Creates an ordered (numbered) list.
13. – List item (used inside <ul> or <ol>).
14. <table> – Defines a table.
15. <tr> – Defines a table row.
16. <td> – Defines a cell in a table row.
17. <th> – Defines a header cell in a table.
18. <form> – Creates a form for user input.
19. <input> – Accepts user input inside a form.
20. <button> – Defines a clickable button.

5.3 List

HTML List Tags

HTML provides **three main types of lists**:

Type	Tag	Description
Ordered		Displays list items in a specific order (numbered: 1, 2, 3… or i, ii, iii…).
Unordered		Displays items with bullet points.
Description	<dl>	Used for name/value pairs, like in dictionaries or glossaries.

Ordered List (<ol>)

Displays items in **numbered order**.


```
  <li>Wake up</li>
  <li>Brush teeth</li>
  <li>Eat breakfast</li>
</ol>
```

Output:

1. Wake up
2. Brush teeth
3. Eat breakfast

Attributes:

- type: Specifies the numbering type (1, A, a, I, i)
- start: Specifies the start number
- reversed: Displays the list in reverse order

Unordered List (<ul>)

Displays items with **bullet points**.

```
<ul>
  <li>Milk</li>
  <li>Eggs</li>
  <li>Bread</li>
</ul>
```

🔷 Output:

- Milk
- Eggs
- Bread

Custom bullets can be styled using CSS (list-style-type).

Description List (<dl>)

Used for **terms and their descriptions**.

```
<dl>
  <dt>HTML</dt>
  <dd>HyperText Markup Language</dd>

  <dt>CSS</dt>
  <dd>Cascading Style Sheets</dd>
</dl>
```

- <dt>: Defines the term (name)
- <dd>: Defines the description (value)

Nesting Lists

You can nest lists within each other:

```
<ul>
  <li>Fruits
    <ul>
      <li>Apple</li>
      <li>Banana</li>
    </ul>
  </li>
  <li>Vegetables</li>
</ul>
```

Note:

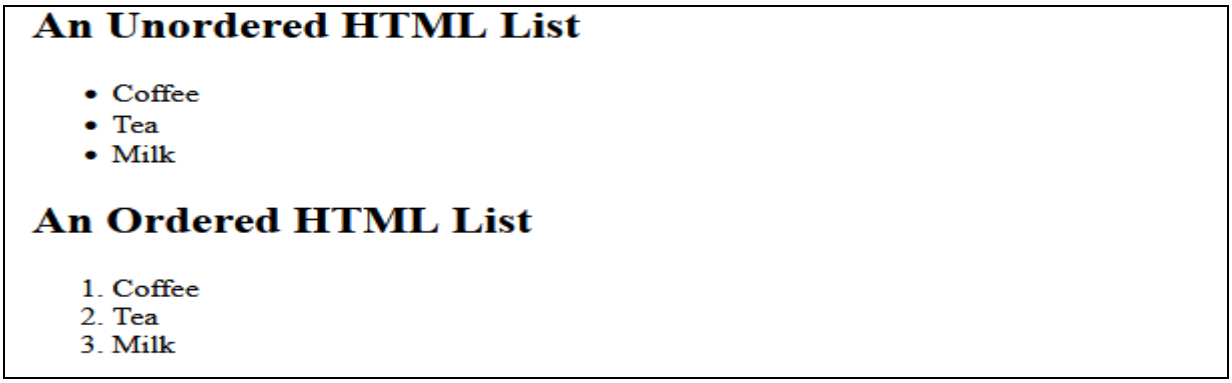
Tag	Purpose
	Ordered list (numbered)
	Unordered list (bulleted)
	List item for both and
<dl>	Description list
<dt>	Term (used in <dl>)
<dd>	Definition (used in <dl>)

Example program:

```
<!DOCTYPE html>
<html>
<body>
<h2>An Unordered HTML List</h2>
<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ul>
<h2>An Ordered HTML List</h2>
<ol>
  <li>Coffee</li>
```

```
  <li>Tea</li>
  <li>Milk</li>
</ol>
</body>
</html>
```

Output:



5.4 Tables

Detailed Information About the <table> Tag in HTML

The <table> tag is used to create tables in HTML, which organize data into rows and columns for easy presentation.

Core Elements of an HTML Table

- **<table>**
  This is the container element that wraps the entire table structure.
- **<tr> (Table Row)**
  Defines a single row within the table. Each <tr> contains one or more <th> or <td> elements.
- **<th> (Table Header Cell)**
  Represents a header cell in a table row. Header cells are usually displayed in bold and centered by default.
- **<td> (Table Data Cell)**
  Represents a standard data cell in a table row.

Additional Table Elements

- **<caption>**
  Provides a title or caption for the table. It is usually displayed above the table.
- **<thead>**
  Groups the header content in the table. This can help with styling and makes tables more accessible.

- **<tbody>**
  Groups the main body content of the table. Useful for applying styles or scripting to the body separately.
- **<tfoot>**
  Groups the footer content of the table, often used for summaries or totals. This section is typically displayed after the table body but can be useful for browsers and assistive technologies to know the footer's role.
- **<colgroup> and <col>**
  Used to group and define attributes for columns, such as width or background color.

Attributes Commonly Used with <table>

- **border**
  Adds a border around the table and cells (deprecated in HTML5; CSS recommended instead).
- **cellpadding**
  Defines the space between the cell content and its border (use CSS padding nowadays).
- **cellspacing**
  Defines the space between cells (use CSS border-spacing).
- **width**
  Sets the width of the table.

Accessibility and Best Practices

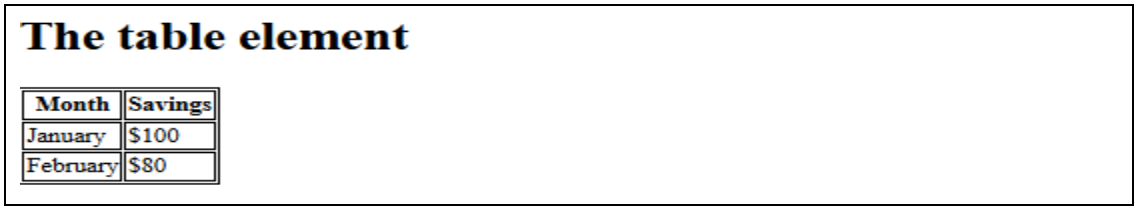
- Always use <th> for header cells to improve readability and accessibility, especially for screen readers.
- Use <caption> to describe the table's purpose.
- Use semantic grouping elements like <thead>, <tbody>, and <tfoot> for better structure.
- Avoid using deprecated attributes like border, cellpadding, and cellspacing. Instead, use CSS for styling.

  Example program:

```
<!DOCTYPE html>

<html>

<head>

<style>

table, th, td {

  border: 1px solid black;

}

</style>
```

```
</head>
<body>

<h1>The table element</h1>

<table>
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$80</td>
  </tr>
</table>

</body>
</html>
```

Output:



5.5. IMAGES

<img> Tag in HTML

The <img> tag is used to embed images into a web page. Unlike most tags, <img> is self-closing—it does not have a closing tag.

Basic Syntax

<img src="image.jpg" alt="Description of image">

Common Attributes of <img>

Attribute	Description
src	**(Required)** The path or URL to the image file.
alt	**(Required for accessibility)** Descriptive text shown if the image cannot be displayed. Useful for screen readers.
width	Sets the width of the image (in pixels or %).
height	Sets the height of the image.
title	Tooltip text shown when you hover over the image.
loading	Determines image loading behavior (lazy or eager).
style	Allows inline CSS styling for the image.

Example

<img src="flowers.jpg" alt="A bouquet of flowers" width="300" height="200" title="Spring Flowers">

Types of Image Sources

- **Local file**:
- <img src="images/photo.jpg" alt="Local image">
- **External URL**:
- <img src="https://example.com/logo.png" alt="Logo">
- **Data URI (embedded image in base64)**:
- <img src="..." alt="Embedded Image">

Best Practices

- Always include the alt attribute for accessibility and SEO.
- Use appropriate image formats:
    - .jpg or .jpeg for photographs
    - .png for images with transparency
    - .gif for animations
    - .svg for vector graphics
- Use width and height to avoid layout shifts (good for performance).

- For responsive design, use CSS (e.g., max-width: 100%) or the picture element.

Advanced Usage with <picture> Element

To provide different images based on screen size or format:

```
<picture>
  <source srcset="image.webp" type="image/webp">
  <source srcset="image.jpg" type="image/jpeg">
  <img src="image.jpg" alt="Example image">
</picture>
```

Example program:

```
<!DOCTYPE html>

<html>

<body>

<h1>The img element</h1>


<img src="img_girl.jpg" alt="Girl in a jacket" width="500" height="600">

</body>

</html>
```

Output:

5.6 FORMS

Forms are essential for adding interactivity to web pages. They allow users to input data and send it to the server, commonly used in user registration, search, and feedback functionalities. Forms can also help navigate complex websites more efficiently.

Basic Syntax of a Form

```
<form action="URL" method="post|get">
  ...
</form>
```

- action: Specifies the destination URL where the form data will be sent.
- method:
  - get: Appends data to the URL; visible in the address bar; suitable for small data submissions.
  - post: Sends data in the request body; more secure; used for larger and sensitive data.

Common Input Tags Inside Forms

<input type="..." name="..." value="..." size="...">

Input Type	Description
text	Creates a single-line text field. The size attribute controls the width, and value sets the default input.
password	Similar to text, but user input is masked with asterisks (*).
radio	Creates a radio button. All radio buttons in the same group must share the same name but have different values. Only one can be selected at a time.
checkbox	Allows multiple selections. Each checkbox can have a different value.
submit	Creates a button that submits the form data to the server. The button text is taken from the value attribute.

Select Dropdown

```
<select name="...">
  <option value="..." selected>Option Label</option>
  ...
</select>
```

- select: Creates a drop-down list.
- option: Specifies the individual options. The selected attribute makes it the default choice.

Textarea

<textarea name="..." rows="..." cols="...">Default Text</textarea>

- Allows multi-line plain text input.
- rows and cols define the visible size.
- Supports vertical scrolling if the content overflows.

HTML forms allow data collection and interaction using various controls like:

- Text fields

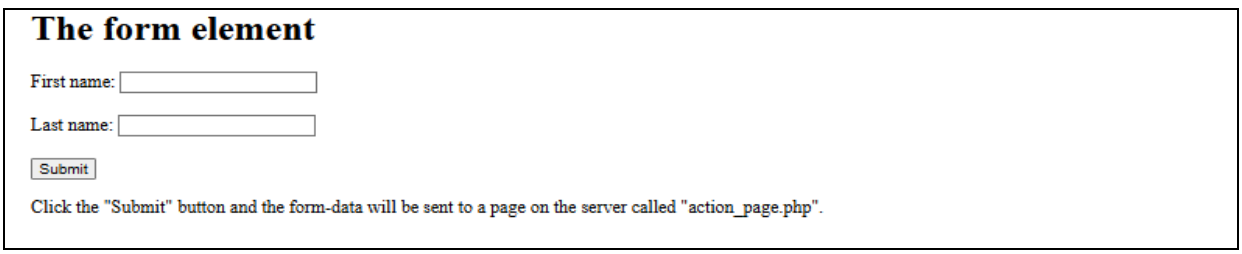
- Password inputs
- Radio buttons
- Checkboxes
- Drop-down lists
- Submit buttons
- Multi-line text areas

These elements help create powerful and interactive web applications.

Example program

```
<!DOCTYPE html>
<html>
<body>
<h1>The form element</h1>
<form action="/action_page.php">
  <label for="fname">First name:</label>
  <input type="text" id="fname" name="fname"><br><br>
  <label for="lname">Last name:</label>
  <input type="text" id="lname" name="lname"><br><br>
  <input type="submit" value="Submit">
</form>
<p>Click the "Submit" button and the form-data will be sent to a page on the
server called "action_page.php".</p>
</body>
</html>
```

Output



5.7 FRAMES

Frames in HTML are used to divide the web browser window into multiple sections, where each section can load a separate HTML document. This allows for the display of multiple

web pages within a single browser window. Frames were commonly used to create navigation menus that stay static while content changes in another section.

Key Concepts of Frames

1. **Frameset**:
   - Replaces the <body> tag in a frames-based page.
   - Defines how the screen is split — vertically (cols) or horizontally (rows).
   - Syntax:
   - <frameset cols="30%, 70%">
   - <frame src="menu.html">
   - <frame src="content.html">
   - </frameset>
2. **Frame**:
   - Defines each individual frame within a frameset.
   - Uses attributes like src, name, scrolling, and frameborder.
   - Example:
   - <frame name="menu" src="menu.html" scrolling="yes" frameborder="0">
3. **NoFrames**:
   - Provides alternative content for browsers that do not support frames.
   - Syntax:
   - <noframes>
   - Your browser does not support frames.
   - </noframes>

Attributes of <frame> Tag

Attribute	Description
src	Specifies the HTML file to display in the frame.
name	Gives the frame a name to be targeted by links.
scrolling	Sets scrollbar visibility: yes, no, or auto.
frameborder	Controls border visibility: 0 (no border), 1 (with border).

Advantages of Using Frames

- Allows independent scrolling in different parts of the screen.
- Useful for persistent navigation menus.
- Reduces the need to reload the entire page.

Disadvantages of Using Frames

- **Deprecated in HTML5** — modern web development discourages frames.
- Not SEO-friendly; search engines may not index all content properly.
- Bookmarking specific frames is difficult.
- Navigation issues (e.g., back button behavior) may confuse users.

Note: While frames were once popular for building complex web layouts, they have been largely replaced by modern techniques using CSS, JavaScript, and responsive design principles. Frames are no longer supported in HTML5, so developers are encouraged to use <iframe> or layout techniques like Flexbox and Grid.

Example

Let's put above example as follows, here we replaced rows attribute by cols and changed their

width. This will create all the three frames vertically:

```
<!DOCTYPE html>
<html>
<head>
<title>HTML Frames</title>
</head>
<frameset cols="25%,50%,25%">
<frame name="left" src="/html/top_frame.htm" />
<frame name="center" src="/html/main_frame.htm" />
<frame name="right" src="/html/bottom_frame.htm" />
<noframes>
<body>
Your browser does not support frames.
</body>
</noframes>
</frameset>
</html>
```

This will produce following result:

5.8 CASCADING STYLE SHEETS (CSS)

Definition:

CSS (Cascading Style Sheets) is a style sheet language used for describing the look and formatting of a document written in HTML or XML. It controls the layout of multiple web pages all at once and separates content from design.

Objectives of CSS:

- To enhance the visual appearance of web pages.
- To separate structure (HTML) from presentation (CSS).
- To allow consistent styling across multiple pages.
- To make web development and maintenance easier and faster.

Types of CSS:

5.8.1. Inline CSS

- Applied directly within an HTML tag using the style attribute.
- Affects only the specific element.
- **Example:**
- This is inline styled text.

5.8.2. Internal CSS

- Defined within a <style> tag in the <head> section of the HTML document.
- Affects all elements on that page.
- **Example:**
- <head>
- <style>
- h1 {
- color: green;
- text-align: center;
- }
- </style>
- </head>

5.8.3. External CSS

- Written in a separate .css file and linked to HTML using the <link> tag.
- Can be used across multiple HTML pages.
- **Example (in HTML):**
- <link rel="stylesheet" type="text/css" href="style.css">

Example (style.css):

```
body {
 background-color: #f0f0f0;
 font-family: Arial;
```

```
}
```

Cascading Order (Priority)

When multiple styles apply to the same element, CSS follows a specific order of precedence:

1. **Inline CSS** (Highest priority)
2. **Internal CSS**
3. **External CSS**
4. **Browser default styles** (Lowest priority)

The term *"cascading"* refers to this order of priority.

CSS Syntax

```
selector {
  property: value;
  property: value;
}
```

- **Selector**: The HTML element you want to style.
- **Property**: The style attribute you want to change.
- **Value**: The value you want to assign to that property.

Example:

```
p {
  color: red;
  font-size: 14px;
}
```

5.8.4 Common CSS Properties

Property	Description	Example
color	Text color	color: blue;
background-color	Background color	background-color: yellow;
font-size	Size of the text	font-size: 18px;
font-family	Font type	font-family: Arial;
text-align	Alignment of text	text-align: center;
margin	Space outside the element	margin: 20px;
padding	Space inside the element	padding: 10px;
border	Border style	border: 1px solid black;
width/height	Size of an element	width: 100px;

CSS Selectors

CSS selectors are used to "select" the HTML elements you want to style.

1. Universal Selector:

```
* {
  margin: 0;
}
```

2. Element Selector:

```
h1 {
  color: blue;
}
```

3. Class Selector:

```
.myclass {
  font-size: 18px;
}
```

Used in HTML like: Text

4. ID Selector:

```
#myid {
  color: green;
}
```

Used in HTML like: Box

5. Group Selector:

```
h1, h2, p {
  color: red;
}
```

6. Descendant Selector:

```
div p {
  color: orange;
}
```

Box Model in CSS

Every HTML element is treated as a box with four components:

1. **Content** – The actual text or image.
2. **Padding** – Space between content and border.
3. **Border** – Surrounds the padding (if any) and content.
4. **Margin** – Space outside the border.

Box Model Example:

```
div {
  padding: 10px;
  border: 1px solid black;
  margin: 20px;
}
```

Advanced CSS Features

Pseudo-classes (e.g., :hover, :first-child)

```
a:hover {
  color: red;
}
```

Pseudo-elements (e.g., ::before, ::after)

```
p::first-letter {
  font-size: 30px;
}
```

Media Queries (for responsive design)

```
@media screen and (max-width: 600px) {
  body {
    background-color: lightblue;
  }
}
```

Animations and Transitions

```
div {
  transition: all 0.3s ease-in-out;
}
```

Flexbox and Grid Layouts (modern layout techniques)

Advantages of CSS

- Enhances presentation and user experience.
- Reduces repetition and saves time.
- Makes maintenance easier.
- Supports responsive web design.
- Allows separation of concerns (structure vs. presentation).

Disadvantages of CSS

- Browser compatibility issues may occur.
- Complex projects may require a CSS preprocessor (like SCSS/SASS).
- Overriding styles can be difficult in large projects.

Note:CSS is an essential part of modern web development. It provides the tools needed to make web pages visually appealing, consistent, and responsive. With advanced features like Flexbox, Grid, animations, and media queries, CSS goes far beyond just changing colors and fonts—it shapes the entire user experience.

Example

```
<html>

<head>

<title>My Web Page</title>

<style type="text/css">

h1{font-family:mssanserif;font-size:30;font-style:italic;fontweight:

bold;color:red;background-color:blue;border:thin groove}

.m{border-width:thick;border-color:red;border-style:dashed}

.mid{font-family:BankGothicLtBT;text-decoration:link;texttransformation:uppercase;text-
indentation:60%}

</style>

</head>

<body class="m">

<h1> ANUCS</h1>

<p class="mid">Acharya  Nagarjuna  University Guntur</p>

</div>

</body>

</html>
```

Output

5.9 SUMMARY

HTML (HyperText Markup Language) is the foundational language for creating web pages, allowing content to be structured using predefined tags. It evolved from SGML and was simplified for web development needs. XML later extended HTML's functionality while retaining its simplicity. HTML was created by Tim Berners-Lee and gained momentum through the Mosaic browser. To standardize web content, versions like HTML 2.0 and HTML 3.0 were introduced, improving compatibility across browsers. HTML's vision emphasizes universal access across devices, from desktops to mobile and low-bandwidth systems. It is widely used due to its simplicity, cross-browser support, and no-cost development. However, it is limited to static content, lacks security features, and can become complex for large pages. HTML documents follow a defined structure with <html>, <head>, and <body> elements. Common tags include headings (<h1> to <h6>), paragraphs (<p>), links (<a>), images (<img>), and forms. HTML lists include ordered (<ol>), unordered (<ul>), and description (<dl>) types, while tables use <table>, <tr>, <td>, and <th> to organize data. The <img> tag embeds images, with src and alt attributes being essential. HTML encourages semantic structure and accessibility best practices through appropriate tag usage.

5.10 KEY TERMS

HTML (HyperText Markup Language),SGML (Standard Generalized Markup Language),XML (eXtensible Markup Language),Tags, Paragraph, Anchor/hyperlink, Image, Lists, Table, Form, Frame, CSS (Cascading Style Sheets).

5.11 Self-Assessment Questions

1. What is the purpose of a <form> tag in HTML?
2. What does the action attribute in a form specify?
3. How does the method="get" work in a form?
4. When should you use the method="post"?
5. What tag is used to take user input in a form?
6. What is the difference between <input type="text"> and <textarea>?
7. How do you create a drop-down list in a form?
8. What is the purpose of the <label> tag?
9. How do radio buttons differ from checkboxes?
10. What happens when a form is submitted with the GET method?
11. Can you send a file using a form? How?
12. What attribute is used to group form controls?
13. What is the use of the name attribute in an input field?
14. How can you prefill values in form elements?

5.12 Further Readings

1. Web Technologies – a computer science perspective, Jeffrey C. Jackson, Pearson, 2007.
2. Web Programming, building internet applications, Chris Bates 2nd edition, WILEY Dreamtech.
3. Internet and World Wide Web – How to program by Dietel and Nieto PHI/Pearson Education Asia.
4. An Introduction to web Design and Programming –Wang-Thomson.
5. Web Applications Technologies Concepts-Knuckles, John Wiley.

Dr. Vasantha Rudramalla

LESSON- 6

INTRODUCTION TO JAVASCRIPT AND CLIENT-SIDE SCRIPTING

Aims and Objectives:

- Understand the basic syntax rules of JavaScript and how to declare and use variables effectively.
- Learn to define and call functions to organize reusable blocks of code.
- Explore how to handle user interactions through JavaScript events like clicks and form submissions.
- Embed JavaScript into HTML documents using <script> tags in various ways (inline, internal, external).
- Implement basic client-side form validation using JavaScript to ensure correct user input before submission.

STRUCTURE:

6.1 JAVASCRIPT

1. Introduction

JavaScript is a high-level, interpreted programming language used primarily for creating interactive and dynamic content on web pages. It allows developers to implement features like image sliders, form validation, dropdown menus, modal windows, and more.

2. History

- **Created by**: Brendan Eich in 1995 at Netscape.
- **Initially called**: Mocha, then LiveScript, and finally renamed JavaScript.
- **Standardized as**: ECMAScript (by ECMA International) in 1997.

3. Features of JavaScript

- JavaScript was designed to add interactivity to HTML pages
- **Lightweight and interpreted**: No need for compilation.
- **Client-side scripting**: Runs in the user's browser.
- **Dynamic typing**: Variables do not require type declaration.
- **Prototype-based OOP**: Supports object-oriented programming.
- **Event-driven**: Reacts to user actions like clicks and input.
- **Cross-platform**: Works on all modern browsers and devices.
- **Asynchronous programming**: Supports callbacks, promises, and async/await.

What Can JavaScript Do?

JavaScript provides HTML designers with a powerful programming tool. While HTML authors may not always be trained programmers, JavaScript is a lightweight scripting language with **simple and readable syntax**, making it accessible to beginners. With just small code snippets, JavaScript can greatly enhance a webpage's functionality.

6.1.1Key Capabilities of JavaScript

a) **Add Dynamic Content to Web Pages**
   - JavaScript can insert dynamic text into HTML using variables.
   - **Example:**
   - document.write("<h1>" + name + "</h1>");
     This writes the value of the variable name directly into the HTML page.

b) **React to User Events**
   - JavaScript can execute specific code in response to events such as:
     - Page loading
     - Mouse clicks
     - Keyboard inputs

- This interactivity improves the user experience.

c) **Read and Modify HTML Elements**
   - JavaScript can access and manipulate the content of HTML elements using the **DOM (Document Object Model)**.
   - It can change text, attributes, styles, and structure of the webpage dynamically.

d) **Validate Form Data**
   - JavaScript can check user inputs before the form is submitted to the server.
   - This reduces unnecessary server load and improves performance and user feedback.
   - Example checks: Empty fields, valid email format, password strength.

e) **Detect Browser Information**
   - JavaScript can detect the user's browser and operating system.
   - This is useful for customizing content or redirecting users to a browser-compatible version of the site.

f) **Create and Manage Cookies**
   - JavaScript can store, retrieve, and delete cookies on the user's computer.
   - Cookies are useful for saving user preferences, login sessions, and other personalized data.

4. JavaScript Syntax Basics

```
// Variable declaration
let name = "Alice"; // ES6 syntax
var age = 25;       // ES5 syntax
const pi = 3.14;    // Constant

// Function
function greet() {
alert("Hello, " + name);
}

// Conditional
if (age >18) {
  console.log("Adult");
} else {
  console.log("Minor");
}
```

6.2 . Variables in JavaScript

What is a Variable?

A **variable** in JavaScript is a named container used to store data that can be referenced and manipulated in a program. Variables allow developers to reuse values and manage dynamic data.

Declaring Variables

JavaScript provides three keywords to declare variables:

Keyword	Scope	Reassignment	Hoisting	Block Scoped
var	Function	Yes	Yes	No
let	Block	Yes	No	Yes
const	Block	No	No	Yes

Syntax

```
var x = 10;
let name = "Alice";
const PI = 3.14;
```

- var is function-scoped and can be redeclared.
- let is block-scoped and cannot be redeclared in the same scope.
- const is block-scoped and must be initialized during declaration. Its value cannot be changed (immutable binding).

Variable Naming Rules

- Names must start with a letter, underscore _, or dollar sign $.
- Cannot start with a digit.
- Are case-sensitive (Name and name are different).
- Should not use reserved JavaScript keywords (like for, if, etc.)

Valid Names:

```
let userName;
let $price;
let _totalAmount;
```

Invalid Names:

```
let 2value;     // Invalid: starts with a number
let if;         // Invalid: reserved keyword
```

Data Types Stored in Variables

Variables can hold different types of data:

```
let age = 25;              // Number
let name = "John";         // String
let isActive = true;       // Boolean
let person = {name:"Ava"}; // Object
let items = [1, 2, 3];     // Array
let result = null;         // Null
```

```
let score;              // Undefined
```

Variable Scope

1. **Global Scope** – Declared outside any function or block.
2. **Function Scope** – Declared inside a function using var.
3. **Block Scope** – Declared using let or const inside {} like loops, if statements.

```
function testScope() {
  var a = 10; // function scoped
  let b = 20; // block scoped
  if (true) {
    let c = 30;
    console.log(c); // 30
  }
  // console.log(c); // Error: c is not defined
}
```

Variable Hoisting

- var declarations are **hoisted** to the top of their scope but not initialized.
- let and const are hoisted but stay in the temporal dead zone (TDZ**)** until declared.

```
console.log(x); // undefined (due to hoisting)
var x = 5;

// console.log(y); // Error: Cannot access 'y' before initialization
let y = 10;
```

Best Practices

- Always prefer let or const over var.
- Use const when the value should not change.
- Declare variables at the top of their scope.
- Use meaningful names (e.g., userAge instead of x).

Example Program

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Variable Example</h2>
<p id="demo"></p>

<script>
  const firstName = "John";
  let age = 30;
  var message = firstName + " is " + age + " years old.";
```

```
document.getElementById("demo").innerHTML = message;
</script>

</body>
</html>
```

Output:
John is 30 years old.

6.3 Functions

What is a Function?

A **function** is a block of reusable code designed to perform a specific task. Functions help organize code, reduce repetition, and improve maintainability.

Why Use Functions?

- To **reuse** code.
- To **break down** complex problems into smaller, manageable parts.
- To improve **readability** and **maintainability**.
- To execute code only when **invoked or called**.

Function Declaration (Function Statement)

```
function greet() {
console.log("Hello, World!");
}
greet(); // Output: Hello, World!
```

Function Parameters and Arguments

Functions can accept inputs called parameters, and these values passed are called arguments.

```
function greetUser(name) {
console.log("Hello, " + name + "!");
}
greetUser("Alice"); // Output: Hello, Alice!
```

Return Statement

Functions can return a value using the return keyword.

```
function add(a, b) {
  return a + b;
}
let result = add(5, 3); // result is 8
```

Function Expressions

A function can also be defined as an expression and stored in a variable.

```
const multiply = function(x, y) {
  return x * y;
};
console.log(multiply(4, 5)); // Output: 20
```

Arrow Functions (ES6+)

Arrow functions provide a shorter syntax for writing functions.

```
const square = (n) => {
  return n * n;
};
console.log(square(6)); // Output: 36
```

Simplified Version (one-liner):

```
const square = n => n * n;
```

Anonymous Functions

Functions without a name are called anonymousfunctions. Often used in event handlers or passed as arguments.

```
setTimeout(function() {
console.log("Executed after 2 seconds");
}, 2000);
```

Immediately Invoked Function Expression (IIFE)

A function that runs immediately after it is defined.

```
(function() {
console.log("IIFE executed!");
})();
```

Nested Functions

Functions can be defined inside other functions and have access to variables in the parent function.

```
function outer() {
  let outerVar = "I'm outer!";

  function inner() {
    console.log(outerVar); // Can access outer variable
  }

inner();
```

```
}
outer();
```

Function Scope

- Variables declared inside a function are local to that function.
- Functions have access to global variables and to variables in their parent functions (closures).

Rest Parameters

Allows a function to accept an indefinite number of arguments as an array.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

Default Parameters

Set default values for parameters.

```
function greet(name = "Guest") {
console.log("Hello, " + name);
}
greet();       // Output: Hello, Guest
greet("Ravi"); // Output: Hello, Ravi
```

Example: Function in HTML

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Function Example</h2>

<p>Click the button to see a message.</p>

<button onclick="showMessage()">Click Me</button>

<script>
  function showMessage() {
alert("Hello! You clicked the button.");
  }
</script>

</body>
</html>
```

Best Practices

- Use descriptive function names.
- Keep functions small and focused on a single task.
- Avoid global variables inside functions.
- Use arrow functions for shorter syntax when appropriate.
- Use const or let when assigning functions to variables.

Common Use Cases

- Input validation
- Performing calculations
- Event handling (e.g., button clicks)
- API calls
- Data manipulation

6.4 Events in JavaScript

6.4.1. Introduction to Events in JavaScript

Events in JavaScript are actions or occurrences that happen in the browser, which JavaScript can respond to. Examples of events include:

- A user clicking a button
- A web page loading
- A form being submitted
- A key being pressed

JavaScript allows developers to create dynamic and interactive web applications by reacting to these events.

6.4.2. Common Types of JavaScript Events

Event	Description
click	Triggered when an element is clicked
dblclick	Triggered when an element is double-clicked
mouseover	Triggered when the mouse pointer moves over an element
mouseout	Triggered when the mouse pointer moves out of an element
keydown	Triggered when a key is pressed
keyup	Triggered when a key is released
load	Triggered when a page or image is fully loaded
submit	Triggered when a form is submitted
change	Triggered when the value of a form element changes
focus	Triggered when an element gains focus
blur	Triggered when an element loses focus

6.4.3. Adding Event Handlers in JavaScript

JavaScript provides multiple ways to attach event handlers to elements:

a) Inline HTML Event Handling
<button onclick="alert('Button clicked!')">Click Me</button>

b) Using DOM Properties
```
let btn = document.getElementById("myBtn");
btn.onclick = function () {
alert("Button clicked!");
};
```

c) Using addEventListener()
```
let btn = document.getElementById("myBtn");
btn.addEventListener("click", function () {
alert("Button clicked using addEventListener!");
});
```

addEventListener() is the preferred method as it allows attaching multiple handlers and supports event bubbling and capturing.

6.4.4. Event Object

When an event occurs, an event object is automatically passed to the event handler. It contains useful information like:

- type – the type of the event
- target – the element on which the event occurred
- preventDefault() – prevents the default action
- stopPropagation() – stops the event from bubbling up

```
document.getElementById("myForm").addEventListener("submit", function (event) {
event.preventDefault(); // Prevents form from submitting
alert("Form submission prevented!");
});
```

6.4.5. Event Propagation

JavaScript events follow a three-phase propagation model:

1. **Capturing Phase** (trickle down)
2. **Target Phase** (event reaches the target)
3. **Bubbling Phase** (bubble up to ancestors)

```
element.addEventListener("click", handler, true); // Capturing
element.addEventListener("click", handler, false); // Bubbling (default)
```

6.4.6. Event Delegation

Instead of adding event listeners to individual elements, event delegation allows attaching a single event listener to a parent element that handles events for its child elements.

```
document.getElementById("parent").addEventListener("click", function (e) {
  if (e.target&&e.target.matches("button.classname")) {
alert("Button inside parent clicked");
  }
});
```

6.4.7. Removing Event Listeners

You can remove an event listener using removeEventListener().

```
function greet() {
  alert("Hello");
}
btn.addEventListener("click", greet);

// To remove
btn.removeEventListener("click", greet);
```

Note: The function reference must be the same to remove it.

6.4.8. Keyboard and Mouse Events

- **Keyboard Events**: keydown, keypress, keyup

```
document.addEventListener("keydown", function (e) {
console.log("Key pressed:", e.key);
});
```

- **Mouse Events**: click, dblclick, mousedown, mouseup, mousemove, mouseenter, mouseleave

```
document.addEventListener("mousemove", function (e) {
console.log("Mouse X:", e.clientX, "Mouse Y:", e.clientY);
});
```

6.4.9. Form Events

Form controls support events like:

- submit
- change
- focus
- blur

```
document.getElementById("name").addEventListener("change", function () {
alert("Name changed!");
});
```

6.4.10. Best Practices for Using Events

- Prefer addEventListener() over inline handlers
- Use event delegation for dynamic or multiple elements
- Always remove unused event listeners to avoid memory leaks
- Avoid anonymous functions if the event needs to be removed
- Use preventDefault() and stopPropagation() wisely

6.4.11. Example: Simple Event Handler

```
<!DOCTYPE html>
<html>
<head>
<title>Event Example</title>
</head>
<body>
<button id="btn">Click Me</button>

<script>
document.getElementById("btn").addEventListener("click", function () {
alert("Button was clicked!");
   });
</script>
</body>
</html>
```

Conclusion

Events in JavaScript are crucial for creating interactive and dynamic web applications. Mastery of events and their proper handling ensures a responsive and user-friendly experience. Understanding concepts like event propagation, delegation, and proper use of listeners is essential for efficient JavaScript programming.

6.5 Embedding JavaScript in HTML

JavaScript can be embedded in an HTML document to add dynamic functionality to web pages. This is done using the <script> tag. JavaScript can be embedded in several ways: directly within the HTML file (inline or internal), or through an external JavaScript file.

6.5.1. Using the <script> Tag

The <script> tag is used to embed JavaScript code in HTML. It can be placed in the <head> or <body> section of an HTML document.

```
<script>
  // JavaScript code goes here
</script>
```

6.5.2. Types of Embedding JavaScript

There are **three main methods** of embedding JavaScript into HTML:

a) Inline JavaScript

JavaScript code can be placed directly within an HTML element's attribute, such as the onclick attribute of a button.

Example:

```
<button onclick="alert('Hello, World!')">Click Me</button>
```

Use Case: Best for simple tasks and demonstrations. Not recommended for complex logic or production code.

b) Internal JavaScript (Embedded in HTML)

You can write JavaScript within a <script> tag inside the HTML file, typically in the <head> or at the end of the <body>.

Example:

```
<!DOCTYPE html>
<html>
<head>
<title>Internal JavaScript</title>
<script>
   function greetUser() {
alert("Welcome to my website!");
   }
</script>
</head>
<body>
<button onclick="greetUser()">Say Hello</button>
</body>
</html>
```

c) External JavaScript

JavaScript can be written in a separate .js file and linked to the HTML document using the src attribute of the <script> tag.

Example:

1. HTML File (index.html)

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>External JavaScript</title>
<script src="script.js"></script>
</head>
<body>
<button onclick="greetUser()">Click Me</button>
</body>
</html>
```

2. JavaScript File (script.js)

```
function greetUser() {
alert("Hello from external JavaScript!");
}
```

Benefits:

- Cleaner HTML code
- Reusability of scripts across pages
- Easier maintenance and debugging

Script Placement in HTML

Placement	Description
<head>	Executes before the content loads. May delay rendering.
End of <body>	Preferred for performance. Loads content first, then script.
defer	Attribute used to defer script execution until after HTML parsing.
async	Attribute used to download and execute script asynchronously.

Example with defer:

```
<script src="script.js" defer></script>
```

Security Note

Avoid embedding user-generated content directly into <script> tags to prevent Cross-Site Scripting (XSS) attacks.

Table

Method	Where Used	Use Case	Example
Inline	HTML attributes	Quick tests, simple interactions	<button onclick="alert('Hi')">
Internal	<script> tag	Page-specific logic	<script>function(){...}</script>
External	Linked .js file	Reusable and maintainable code	<script src="script.js"></script>

Best Practices

- Always place scripts just before </body> unless required in <head>.

- Use external JavaScript for modularity and reuse.
- Avoid inline JavaScript in production.
- Use defer or async for performance optimization.

6.6 Form Validation Basics

Form validation is the process of checking the data entered into a form to ensure it meets certain rules before it is submitted to a server. It helps in maintaining data quality, enhancing user experience, and improving security.

Types of Form Validation

1. **Client-Side Validation**
   - Happens in the user's browser (before data is sent to the server).
   - Uses HTML5 attributes or JavaScript.
   - Provides immediate feedback.
2. **Server-Side Validation**
   - Happens on the server after the form is submitted.
   - Essential for security (users can bypass client-side validation).
   - Typically done using languages like PHP, Python, Java, etc.

Common Validation Rules

- **Required fields**: Ensures the user has entered something.
- **Data type checks**: Ensures values are numbers, emails, URLs, etc.
- **Length checks**: Limits input to a specific number of characters.
- **Pattern matching**: Validates format using regular expressions (e.g., phone number, postal code).
- **Matching fields**: Confirms that values in two fields are the same (e.g., password and confirm password).

Client-Side Validation Techniques

1. HTML5 Attributes

- required: Field must be filled.
- type="email": Must be a valid email format.
- min, max, maxlength: Numeric or text length limits.
- pattern: Regular expression validation.

```
<form>
<input type="text" name="name" required>
<input type="email" name="email" required>
<input type="password" name="pwd" minlength="6" required>
<input type="submit">
</form>
```

Example

```
<!DOCTYPE html>
```

```
<html>
<head>
<script>
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
alert("Name must be filled out");
    return false;
  }
}
</script>
</head>
<body>
<h2>JavaScript Validation</h2>
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()"
method="post">
  Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Output:

Common Use Cases

- Form validation
- Image sliders
- Interactive maps
- Popups/modals
- Real-time updates (chat, notifications)
- Games
- AJAX requests (load data without refreshing page)

Advantages

- Enhances user experience
- Reduces server load
- Fast client-side processing
- Wide community and support

Limitations

- Runs in the browser – can't access server files directly
- Code visibility (can be viewed and modified by users)
- May behave differently across browsers if not standardized

Modern JavaScript Tools and Frameworks

- **Libraries**: jQuery, Axios
- **Frameworks**: React, Angular, Vue.js
- **Runtimes**: Node.js (for server-side JS)
- **Build tools**: Webpack, Babel, ESLint

Note:

JavaScript is an essential language for web development. It bridges the gap between static HTML/CSS and dynamic user interaction, making websites more functional, responsive, and user-friendly.

Example: Change Text on Button Click

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript Example</title>
</head>
<body>
<p id="myParagraph">This is the initial text.</p>
<button onclick="changeText()">Click Me</button>
<script>
   function changeText() {
document.getElementById("myParagraph").innerHTML = "This text has been changed!";
   }
</script>
</body>
</html>
```

Explanation of Code

1. **HTML Structure**:
   - A paragraph element is defined with the ID myParagraph.
   - A button element is provided to trigger the JavaScript function.
2. **JavaScript Functionality**:
   - The changeText() function is defined inside a <script> tag.
   - It uses document.getElementById("myParagraph") to access the paragraph.
   - The innerHTML property is used to change the paragraph's content.
3. **Event Handling**:
   - The onclick attribute of the button is used to call the changeText() function when the button is clicked.

4. **Output**:
   - Initially, the paragraph displays: "This is the initial text."
   - After clicking the button, the paragraph updates to: "This text has been changed!"

Key JavaScript Concepts Demonstrated

- **HTML Structure**: Defines the elements to be manipulated.
- **JavaScript Behavior**: Provides interactivity and control logic.
- **Event Handling**: The onclick event responds to user actions.
- **DOM Manipulation**: document.getElementById() and innerHTML are used to modify webpage content.
- **Functions**: Encapsulate reusable code logic that executes when triggered.

6.7.DATA TYPES

- **Primitive**: String, Number, Boolean, Null, Undefined, Symbol, BigInt.
- **Non-Primitive**: Object, Array, Function.

Strings - are a series of letters and numbers enclosed in quotation marks. JavaScript uses the string literally; it doesn't process it. You'll use strings for text you want displayed or values you want passed along.

Numbers - are values that can be processed and calculated. You don't enclose them in quotation marks. The numbers can be either positive or negative.

Boolean (true/false) - lets you evaluate whether a condition meets or does not meet specified criteria.

Null - is an empty value. null is not the same as 0 -- 0 is a real, calculable number, whereas null is the absence of any value.

Data Types

TYPE	EXAMPLE
Numbers	Any number, such as 17, 21, or 54e7
Strings	"Greetings!" or "Fun"
Boolean	Either true or false
Null	A special keyword for exactly that – the null value (that is, nothing)

6.8 SUMMARY

JavaScript is a high-level, interpreted scripting language used to add interactivity to web pages. Created by Brendan Eich in 1995, it was standardized as ECMAScript in 1997. JavaScript supports dynamic typing, prototype-based OOP, event-driven programming, and asynchronous operations. It can modify HTML elements, validate forms, handle browser detection, and manage cookies. Variables are declared using var, let, or const, each with different scopes and behaviors. JavaScript supports various data types and has well-defined

rules for naming variables. Functions in JavaScript allow code reuse and come in multiple forms, including function declarations, expressions, and arrow functions. Events are central to JavaScript, enabling responses to user interactions like clicks and keypresses. Event handling can be done inline, via DOM properties, or using addEventListener(). Developers can manage event propagation and delegation for efficient interaction handling. JavaScript can be embedded into HTML using <script> tags—inline, internal, or external. External scripts promote modularity and cleaner code. The defer and async attributes optimize script loading. Proper use of events and functions enhances web app responsiveness. JavaScript's versatility makes it essential for modern web development.

6.9 KEY TERMS

JavaScript,ECMAScript,Variable (var, let, const),Data Types,Function,Arrow Function,Event Handling,DOM (Document Object Model),Form Validation,addEventListener(),Event Delegation,Event Propagation,Script Tag (**<script>**),Inline/Internal/External JavaScript,Hoisting.

6.10 SELF-ASSESSMENT QUESTIONS

1. What is JavaScript and where is it primarily used?
2. What is the difference between var, let, and const in JavaScript?
3. List some of the basic data types in JavaScript.
4. What is the purpose of functions in JavaScript?
5. How do arrow functions differ from regular functions in JavaScript?
6. What is the DOM in JavaScript?
7. How can JavaScript be added to an HTML document?
8. What is form validation and why is it important in JavaScript?
9. Explain the use of addEventListener() in event handling.
10. What is the difference between inline, internal, and external JavaScript?

6.11 FURTHER READINGS

1. JavaScript: The Definitive Guide, Seventh Editionby David Flanagan. O'Reilly Media.
2. Eloquent JavaScript: A Modern Introduction to Programming, Third Editionby Marijn Haverbeke. No Starch Press.
3. Beginning JavaScript, Fifth Editionby Jeremy McPeak. Wrox (Wiley).
4. Learning JavaScript Design Patterns, First Editionby Addy Osmani. O'Reilly Media.
5. The complete Reference Java 2 Fifth Edition by Patrick Naughton and Herbert Schildt. TMH

Dr. Vasantha Rudramalla

LESSON- 7

OBJECTS IN JAVA SCRIPT AND DYNAMIC HTML (DHTML)

Aims and Objectives:

- Understand the concept and structure of JavaScript objects, including properties and methods.
- Learn how to create, access, modify, and delete object properties using different approaches.
- Explore how to use JavaScript to dynamically change HTML content, attributes, and styles.
- Apply the this keyword and methods within objects to build real-world functionalities.
- Demonstrate the use of JavaScript in manipulating DOM elements for creating interactive web pages.

STRUCTURE:

7.1 Objects in JavaScript

In JavaScript, objects are collections of related data and functionality. These can include built-in objects provided by the browser (like window, document, etc.) or user-defined objects. Objects contain properties (values) and methods (functions).

1. Window Object

The **window object** is the top-level object in the browser's JavaScript environment. It represents the browser window or tab that displays the HTML document. Every global variable, function, or object created in JavaScript is automatically a member of the window object.

Key Features:

- It is automatically created by the browser when a web page loads.
- Acts as the global object in browsers.
- Contains properties like location, history, navigator, etc.

- Provides methods like alert(), prompt(), confirm(), setTimeout(), and setInterval().
- **Note:**
- The window object is global, so you usually don't need to type window. explicitly.
- alert("Hello!"); // Equivalent to window.alert("Hello!");
- console.log(window.innerHeight); // Displays window height

Common Properties:

Property	Description
window.innerWidth	Width of the content area of the window
window.innerHeight	Height of the content area
window.location	Provides URL and navigation functions
window.document	Refers to the current page's document
window.navigator	Gives browser information
window.history	Controls the session history

Common Methods:

Method	Description
alert(message)	Displays a popup alert box
confirm(message)	Displays a popup with OK and Cancel
prompt(message)	Displays a popup with a text input field
setTimeout(fn, time)	Executes a function after a delay
setInterval(fn, time)	Repeats function execution at intervals
open(url)	Opens a new browser window
close()	Closes the current window

Example Usage:

```
// Display an alert box
window.alert("Welcome to JavaScript!");
// Set a timeout
window.setTimeout(function() {
  alert("This alert appears after 3 seconds.");
}, 3000);
// Log the window dimensions
```

```
console.log("Width: " + window.innerWidth);
console.log("Height: " + window.innerHeight);
// Redirect to another URL
window.location.href = "https://www.example.com";
```

2. Navigator Object

The Navigator Object is a built-in object in JavaScript that contains information about the web browser being used by the client (user). It is a property of the window object, and it provides useful metadata about the browser's name, version, platform, and capabilities.

The navigator object is often used for browser detection and for controlling or accessing certain browser features.

Syntax:

```
window.navigator
// or simply
navigator
```

Key Properties of Navigator Object:

Property	Description
appName	Returns the name of the browser (mostly returns "Netscape" for compatibility reasons)
appCodeName	Returns the code name of the browser (usually "Mozilla")
appVersion	Returns the version information of the browser
userAgent	Returns the user-agent header sent by the browser to the server
platform	Returns the platform on which the browser is running (e.g., "Win32", "Linux x86_64")
language or languages	Returns the language preference of the browser
cookieEnabled	Returns true if cookies are enabled in the browser
onLine	Returns true if the browser is online
javaEnabled()	Returns true if Java is enabled in the browser
plugins	Returns a list of all plugins installed in the browser
mimeTypes	Returns a list of all MIME types supported by the browser
hardwareConcurrency	Returns the number of logical processors available to run threads
maxTouchPoints	Returns the maximum number of simultaneous touch points supported
webdriver	Returns true if the browser is controlled by automation (e.g., Selenium)

Example: Using Navigator Object

```
<!DOCTYPE html>
<html>
<head>
  <title>Navigator Object Example</title>
</head>
<body>
  <h2>Browser Information</h2>
  <script>
    document.write("<b>Browser Name:</b> " + navigator.appName + "<br>");
    document.write("<b>Browser Code Name:</b> " + navigator.appCodeName + "<br>");
    document.write("<b>Browser Version:</b> " + navigator.appVersion + "<br>");
    document.write("<b>Platform:</b> " + navigator.platform + "<br>");
    document.write("<b>User Agent:</b> " + navigator.userAgent + "<br>");
    document.write("<b>Cookies Enabled:</b> " + navigator.cookieEnabled + "<br>");
    document.write("<b>Online Status:</b> " + navigator.onLine + "<br>");
  </script>
</body>
</html>
```

Common Uses of Navigator Object

1. **Browser Detection:**
   To perform actions depending on which browser is being used (though feature detection is preferred over browser detection).
2. **Language Preferences:**
   To customize content based on the user's preferred language.
3. **Online/Offline Detection:**
   To check whether the user is connected to the internet.
4. **Feature Availability:**
   Determine if Java, cookies, or other plugins are enabled or available.

Note:

Avoid relying solely on browser detection using the navigator object for functionality. Instead, use feature detection (via libraries like Modernizr) to ensure consistent behavior across different environments.

Table

Feature	Description
Object Name	navigator
Purpose	Provides information about the browser and system
Important Properties	userAgent, platform, language, cookieEnabled, onLine
Common Use	Browser info, feature support, user customization

Key Properties:

- navigator.appName: Browser name
- navigator.appVersion: Browser version
- navigator.platform: OS platform
- navigator.language: Language setting
- navigator.onLine: Checks if the browser is online

Example:

```
console.log("Browser Name: " + navigator.appName);
console.log("Online: " + navigator.onLine);
```

3. Document Object

The document **object** is a part of the Browser Object Model (BOM) and more specifically, it is the core of the DOM (Document Object Model)**.** It represents the webpag**e** loaded in the browser and acts as an entry point to access and manipulate HTML content, structure, and styles through JavaScript.

Syntax:

```
window.document
// or simply
document
```

Purpose of Document Object

- To access elements in the HTML document.
- To modify the structure, style, and content of the webpage.
- To handle events.
- To dynamically create or delete HTML elements.

Important Properties of Document Object

Property	Description
document.title	Gets or sets the title of the document
document.URL	Returns the complete URL of the document
document.domain	Gets the domain name of the server
document.body	Represents the <body> element of the document
document.head	Represents the <head> element of the document
document.forms	Returns a collection of all <form> elements
document.images	Returns a collection of all <img> elements
document.links	Returns a collection of all <a> elements with an href
document.cookie	Gets or sets cookies for the current page
document.readyState	Returns the loading state of the document (e.g., "loading", "complete")
document.documentElement	Returns the root <html> element

Commonly Used Methods of Document Object

Method	Description
getElementById(id)	Returns the element with the specified ID
getElementsByTagName(tag)	Returns a collection of elements with the given tag name
getElementsByClassName(class)	Returns a collection of elements with the specified class name
querySelector(selector)	Returns the first element that matches the CSS selector
querySelectorAll(selector)	Returns all elements that match the CSS selector
createElement(tag)	Creates a new HTML element
createTextNode(text)	Creates a new text node
appendChild(node)	Adds a node to the end of a list of children of a specified parent
removeChild(node)	Removes a child node from the document
write(text)	Writes HTML expressions or JavaScript code to the document
open()	Opens a document for writing
close()	Closes a document stream opened with document.open()

Example: Access and Modify Elements

<!DOCTYPE html>

```
<html>
<head>
   <title>Document Object Example</title>
</head>
<body>
   <h1 id="heading">Hello World</h1>
   <p class="message">This is a paragraph.</p>
   <script>
      // Access and modify content using document object
      document.getElementById("heading").innerHTML = "Welcome to JavaScript!";
      document.querySelector(".message").style.color = "blue";
      alert("Page Title: " + document.title);
   </script>
</body>
</html>
```

Use Cases of the Document Object

- Reading or changing the page content dynamically.
- Adding or removing HTML elements using JavaScript.
- Validating form input.
- Responding to user interactions like clicks and keypresses.
- Manipulating CSS styles and classes.

Important Notes

- Changes made using the document object reflect immediately on the webpage.
- document.write() should be avoided after the page has loaded as it can **overwrite** the entire document.

Table

Feature	Description
Object Name	document
Part of	DOM (Document Object Model)
Purpose	Interact with and modify the HTML structure
Common Methods	getElementById(), querySelector(), createElement(), write()
Common Properties	title, URL, body, head, forms, images
Usage	Dynamic HTML manipulation, event handling, content modification

Common Methods:

- getElementById()
- getElementsByTagName()
- querySelector()
- createElement(), appendChild()

Example:

document.getElementById("demo").innerHTML = "Text changed!";

4. Form Object

Form Object in JavaScript

The Form object in JavaScript represents an HTML <form> element. It allows JavaScript to access**,** manipulate**,** validate, and submit form data dynamically.

Each form in an HTML document becomes a part of the document.forms collection. You can access a form either by its index or name.

Syntax:

// Access form by index

document.forms[0]

// Access form by name (name attribute in HTML)

document.forms["formName"]

Purpose of the Form Object

- To retrieve form input values.
- To set or modify form fields.
- To validate form inputs before submission.
- To handle form submission via JavaScript.

Important Properties of the Form Object

Property	Description
elements	Collection of all form elements (inputs, selects, buttons, etc.)
length	Number of elements in the form
name	The name of the form (from the name attribute)
action	URL where the form data is sent (from the action attribute)
method	HTTP method used when submitting the form (GET or POST)
target	Specifies where to display the response (like _blank, _self)

Property	Description
enctype	Encoding type for submitted form data (e.g., application/x-www-form-urlencoded)
acceptCharset	Character encodings the server can handle
autocomplete	Indicates whether form input fields can be auto-completed (on or off)
noValidate	If present, form will not be validated on submit

Common Methods of the Form Object

Method	Description
submit()	Submits the form programmatically
reset()	Resets all form fields to their default values
checkValidity()	Returns true if the form is valid
reportValidity()	Reports validity and displays error messages if invalid

Example: Accessing and Using Form Object

```
<!DOCTYPE html>
<html>
<head>
   <title>Form Object Example</title>
</head>
<body>
<form name="myForm" action="/submit" method="post">
   Name: <input type="text" name="username"><br>
   Email: <input type="email" name="email"><br>
   <input type="button" value="Submit" onclick="submitForm()">
</form>
<script>
   function submitForm() {
      var form = document.forms["myForm"];
      var name = form["username"].value;
      var email = form["email"].value;
      if (name === "" || email === "") {
         alert("All fields are required!");
```

```
        } else {
            alert("Form submitted with:\nName: " + name + "\nEmail: " + email);
            form.submit(); // optional: submit programmatically
        }
    }
</script>
</body>
</html>
```

Form Validation Example

```
function validateForm() {
    var form = document.forms["myForm"];
    if (!form.checkValidity()) {
        alert("Form is invalid!");
    } else {
        alert("Form is valid and ready to submit.");
    }
}
```

Accessing Form Elements

You can access individual form fields using:

```
document.forms["myForm"]["username"].value;
```

Or loop through all form elements:

```
var form = document.forms[0];
for (var i = 0; i < form.length; i++) {
    console.log(form.elements[i].name + ": " + form.elements[i].value);
}
```

Table

Feature	Details
Object	Form
Part of	DOM (document.forms)
Used for	Accessing and manipulating form fields
Properties	elements, action, method, target, length, name
Methods	submit(), reset(), checkValidity(), reportValidity()
Use Cases	Input validation, programmatic form submission, dynamic data handling

5. Date Object

Date Object in JavaScript

The Date **object** is a built-in JavaScript object used to work with dates and times. It allows you to create, retrieve, and manipulate date and time values such as the current date, day, month, year, hour, minute, second, and millisecond.

Syntax:

```
let date = new Date();           // Current date and time
let date = new Date(milliseconds); // Date based on milliseconds since Jan 1, 1970
let date = new Date(dateString);   // Date from a string (e.g., "2025-06-11")
let date = new Date(year, month, day, hours, minutes, seconds, ms);
```

Note: In JavaScript, months are zero-indexed (January = 0, December = 11).

Creating Date Objects

1. Current Date and Time

```
let now = new Date();
```

2. From String

```
let d = new Date("2025-06-11");
```

3. From Components

```
let d = new Date(2025, 5, 11, 10, 30, 0);  // June 11, 2025 10:30:00
```

4. From Milliseconds

```
let d = new Date(0);  // January 1, 1970 (Unix Epoch Time)
```

Important Date Methods

Getter Methods (to retrieve parts of the date)

Method	Description
getFullYear()	Returns the 4-digit year (e.g., 2025)
getMonth()	Returns the month (0–11)
getDate()	Returns the day of the month (1–31)
getDay()	Returns the day of the week (0–6, where 0 = Sunday)
getHours()	Returns the hour (0–23)
getMinutes()	Returns the minutes (0–59)
getSeconds()	Returns the seconds (0–59)
getMilliseconds()	Returns the milliseconds (0–999)
getTime()	Returns milliseconds since Jan 1, 1970
getTimezoneOffset()	Returns difference from UTC in minutes

Setter Methods (to set parts of the date)

Method	Description
setFullYear(year)	Sets the full year
setMonth(month)	Sets the month (0–11)
setDate(day)	Sets the day of the month
setHours(hour)	Sets the hour
setMinutes(min)	Sets the minutes
setSeconds(sec)	Sets the seconds
setMilliseconds(ms)	Sets the milliseconds
setTime(ms)	Sets the date based on milliseconds since 1970

Conversion Methods

Method	Description
toDateString()	Returns date as a readable string (e.g., "Wed Jun 11 2025")
toTimeString()	Returns time as a readable string (e.g., "10:30:00 GMT+0530")
toISOString()	Returns ISO format string (e.g., "2025-06-11T05:00:00.000Z")
toLocaleDateString()	Returns date as a localized string
toLocaleTimeString()	Returns time as a localized string
toUTCString()	Returns UTC date string

Examples

Example 1: Get Current Date and Time

```
let now = new Date();
console.log(now.toString());
```

Example 2: Extract Date Components

```
let today = new Date();
console.log("Year: " + today.getFullYear());
console.log("Month: " + today.getMonth()); // 0 = January
console.log("Date: " + today.getDate());
console.log("Day: " + today.getDay());     // 0 = Sunday
```

Example 3: Set Custom Date

```
let customDate = new Date();
customDate.setFullYear(2026);
customDate.setMonth(11); // December
```

```
customDate.setDate(25);
console.log(customDate.toDateString());  // "Fri Dec 25 2026"
```

Example 4: Compare Two Dates

```
let d1 = new Date("2025-06-01");
let d2 = new Date("2025-06-11");
if (d1 < d2) {
   console.log("d1 is earlier than d2");
}
```

Use Cases of Date Object

- Displaying current date and time
- Calculating date differences
- Creating countdowns or clocks
- Validating date input in forms
- Generating timestamps for logs

Important Notes

- Months are **0-**indexed: January is 0, December is 11.
- Always consider time zone differences when working with global users.
- Prefer using toISOString() or toLocaleString() for consistent formatting.

Table

Feature	Description
Object	Date
Purpose	Work with dates and times
Common Getters	getFullYear(), getMonth(), getDate(), getHours()
Common Setters	setFullYear(), setMonth(), setDate()
Formats	toString(), toISOString(), toLocaleString()
Use Cases	Clocks, timestamps, reminders, form validations

6. String Object

The **String object** in JavaScript is a built-in object that allows you to create, manipulate, and work with text (string values). Strings are sequences of characters used for storing and manipulating text.

What is a String?

A **string** is a sequence of characters enclosed in:

- single quotes (')

- double quotes ("")
- backticks (``) — for template literals

```
let str1 = 'Hello';
let str2 = "World";
let str3 = `Hello, ${str2}`; // Template literal
```

Creating String Objects

1. String Literal

```
let name = "JavaScript";
```

2. String Object Using Constructor

```
let nameObj = new String("JavaScript");
```

📌 **Note:** Using string objects (new String(...)) is not recommended for regular usage. Use string **literals** for simplicity and performance.

Common String Properties

Property	Description
length	Returns the number of characters in the string

```
let text = "Hello World";
console.log(text.length);  // Output: 11
```

Common String Methods

String Inspection Methods

Method	Description
charAt(index)	Returns the character at a specified index
charCodeAt(index)	Returns the Unicode of the character
includes(substring)	Checks if the string contains a substring
startsWith(substring)	Checks if the string starts with a substring
endsWith(substring)	Checks if the string ends with a substring
indexOf(substring)	Returns the index of the first occurrence
lastIndexOf(substring)	Returns the last index of the substring

String Manipulation Methods

Method	Description
concat(string2)	Combines two or more strings
replace(search, replace)	Replaces substring with another
replaceAll(search, replace)	Replaces all occurrences
slice(start, end)	Extracts part of a string
substring(start, end)	Similar to slice but does not accept negative indices
substr(start, length)	Deprecated; use slice instead
toLowerCase()	Converts string to lowercase
toUpperCase()	Converts string to uppercase
trim()	Removes whitespace from both ends
trimStart() / trimEnd()	Trims start or end spaces

String Splitting and Matching

Method	Description
split(separator)	Splits a string into an array
match(regex)	Matches string against a regular expression
matchAll(regex)	Returns all matches as an iterator
search(regex)	Searches for a match using a regular expression
includes(text)	Checks if text exists in the string

String Conversion Methods

Method	Description
toString()	Returns string representation
valueOf()	Returns the primitive string value

Examples

Example 1: Basic Usage

```
let msg = "Hello JavaScript";
console.log(msg.length);         // 17
console.log(msg.charAt(0));      // "H"
console.log(msg.toUpperCase());  // "HELLO JAVASCRIPT"
```

Example 2: Searching and Replacing

```
let txt = "The rain in Spain";
```

```
console.log(txt.includes("rain"));        // true
console.log(txt.indexOf("rain"));         // 4
console.log(txt.replace("rain", "sun"));  // "The sun in Spain"
```

Example 3: Extracting Parts of a String

```
let str = "Hello, world!";
console.log(str.slice(0, 5));    // "Hello"
console.log(str.substring(7));  // "world!"
```

Example 4: Using Template Literals

```
let name = "Alice";
let greeting = `Hello, ${name}!`;
console.log(greeting); // "Hello, Alice!"
```

Example 5: Splitting a String

```
let csv = "apple,banana,orange";
let fruits = csv.split(",");
console.log(fruits); // ["apple", "banana", "orange"]
```

Use Cases of String Object

- Displaying and formatting messages
- Validating form inputs
- Searching and filtering text
- Parsing and formatting data
- Creating dynamic content (e.g., templates)

Important Notes

- Strings are immutable: operations like replace() or concat() return a new string, not modify the original.
- Prefer using string literals instead of new String().
- For multi-line strings or embedded variables, use template literals (`) with ${}.

Table

Feature	Description
Object	String
Type	Wrapper object for text
Common Property	Length
Common Methods	charAt(), slice(), toUpperCase(), replace(), split()
Use Cases	Text display, manipulation, validation, and search
Mutability	Immutable – returns new string after changes

7. Array Object

The **Array object** in JavaScript is a built-in object used to store multiple values in a single variable**.** Arrays are versatile data structures that allow storing items like numbers, strings, objects, and even other arrays.

What is an Array?

An **array** is a collection of items stored in a single variable. Each item has a numeric index, starting from 0.

```
let fruits = ["Apple", "Banana", "Cherry"];
```

Creating Arrays

1. Array Literal (Recommended)

```
let colors = ["Red", "Green", "Blue"];
```

2. Using new Array() Constructor

```
let numbers = new Array(1, 2, 3, 4);
```

3. Creating an Empty Array

```
let emptyArr = [];
```

⚠ Avoid using new Array(size) unless you know what you are doing. It creates an empty array of a specific length but without elements.

Array Properties

Property	Description
length	Returns the number of elements in the array

```
let arr = [10, 20, 30];
console.log(arr.length); // Output: 3
```

Common Array Methods

Adding & Removing Elements

Method	Description
push(item)	Adds item to the end
pop()	Removes and returns the last item

Method	Description
unshift(item)	Adds item to the beginning
shift()	Removes and returns the first item
splice(start, deleteCount, items...)	Adds/removes elements at a specific index
slice(start, end)	Returns a shallow copy of a portion of the array

Searching & Testing

Method	Description
indexOf(item)	Returns the first index of the item
lastIndexOf(item)	Returns the last index of the item
includes(item)	Checks if the array contains the item
find(callback)	Returns the first element that satisfies the condition
findIndex(callback)	Returns the index of the first element that satisfies the condition
some(callback)	Returns true if at least one element passes the test
every(callback)	Returns true if all elements pass the test

Iteration and Transformation

Method	Description
forEach(callback)	Executes a function on each element
map(callback)	Creates a new array with transformed values
filter(callback)	Creates a new array with elements that pass the test
reduce(callback, initialValue)	Reduces array to a single value
flat(depth)	Flattens nested arrays
flatMap(callback)	Maps and flattens results into a new array

Sorting & Reversing

Method	Description
sort(compareFunction)	Sorts the array elements in-place
reverse()	Reverses the order of the elements

Joining & Converting

Method	Description
join(separator)	Joins all elements into a string
toString()	Converts array to a string (comma-separated)
Array.isArray()	Checks whether a variable is an array

Examples

Example 1: Creating and Accessing Elements

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[0]); // Output: "Apple"
console.log(fruits.length); // Output: 3
```

Example 2: Adding and Removing

```
fruits.push("Mango");
console.log(fruits); // ["Apple", "Banana", "Cherry", "Mango"]
fruits.pop();
console.log(fruits); // ["Apple", "Banana", "Cherry"]
```

Example 3: Iterating Over an Array

```
fruits.forEach(function(item, index) {
   console.log(index + ": " + item);
});
```

Example 4: Transforming with map()

```
let numbers = [1, 2, 3];
let squared = numbers.map(n => n * n);
console.log(squared); // [1, 4, 9]
```

Example 5: Filtering with filter()

```
let scores = [45, 67, 89, 30];
let passed = scores.filter(score => score >= 50);
console.log(passed); // [67, 89]
```

Example 6: Reducing with reduce()

```
let total = [10, 20, 30].reduce((acc, curr) => acc + curr, 0);
console.log(total); // 60
```

Use Cases of Array Object

- Storing lists of items (names, numbers, etc.)
- Iterating and transforming data
- Filtering and sorting information
- Storing results from APIs or form data
- Creating dynamic UI components

Important Notes

- JavaScript arrays are dynamic — they grow/shrink in size.
- Arrays can store mixed data types, but it's best to use uniform types.
- Use Array.isArray**()** to check if a variable is truly an array (as typeof returns "object").

Table

Feature	Description
Object	Array
Purpose	Store and manipulate ordered collections of data
Indexing	Starts from 0
Property	Length
Common Methods	push(), pop(), map(), filter(), reduce(), sort()
Use Cases	Lists, filtering, mapping, sorting, iteration, UI building
Type Check	Array.isArray(var)

Overview Table

Object	Purpose	Example Method / Property
window	Global browser context	alert(), setTimeout()
navigator	Browser information	navigator.appName
document	HTML document interaction	getElementById()
form	HTML form access and data	document.forms
Date	Dates and time manipulation	getDate(), getFullYear()
String	Text processing	toUpperCase(), replace()
Array	Collection of values	push(), length, sort()

7.2 Dynamic HTML with Java Script

What is Dynamic HTML (DHTML)?

Dynamic HTML (DHTML) is not a language itself, but a collection of technologies used together to create interactive and dynamic websites. DHTML allows web pages to change after they are loaded**,** without requiring a full page reload.

DHTML Combines:

- **HTML** – to define structure/content
- **CSS** – to style and position elements
- **JavaScript** – to add interactivity and control behavior
- **DOM (Document Object Model)** – to access and manipulate HTML elements dynamically

Purpose of DHTML

- Enhance user experience with interactive and responsive interfaces
- Create animations, dropdown menus, form validation, etc.
- Update content dynamically without reloading the entire page

Core Components of DHTML

Component	Role
HTML	Provides the basic structure and elements of the page
CSS	Styles elements dynamically (e.g., color, font, layout)
JavaScript	Adds functionality and manipulates page content
DOM	Allows access and modification of elements on the fly

7.2.1 How JavaScript Enables DHTML

JavaScript is the engine behind DHTML. Using JavaScript, you can:

- Access and modify HTML elements (via DOM)
- Change CSS styles dynamically
- Respond to user events (click, hover, input, etc.)
- Insert, remove, or replace content on the page

Example: Changing Content Dynamically

```
<!DOCTYPE html>
<html>
<head>
  <title>DHTML Example</title>
```

```
  <script>
    function changeContent() {
      document.getElementById("demo").innerHTML = "Content changed using DHTML!";
    }
  </script>
</head>
<body>
  <h2 id="demo">Original Content</h2>
  <button onclick="changeContent()">Click Me</button>
</body>
</html>
```

Explanation:

- JavaScript accesses the element using getElementById
- The content is updated dynamically without reloading the page

DHTML Features Enabled by JavaScript

Feature	Description
Event Handling	Responds to user actions like clicks, keypresses, hovers
DOM Manipulation	Modifies HTML structure in real-time
CSS Style Control	Changes the look of elements dynamically
Animations	Moves, fades, or transforms elements smoothly
Form Validation	Validates user input before submission
Real-Time Content Updates	Updates parts of the page based on conditions or input

7.2.2 DOM Methods in DHTML

JavaScript Method	Purpose
getElementById()	Access a specific element
getElementsByClassName()	Access multiple elements by class
innerHTML	Set or get the content inside an element
style.property	Change element style (e.g., style.color)
createElement()	Create new HTML elements
appendChild()	Add new elements to the DOM
removeChild()	Remove elements from the DOM

Dynamic Style Example

```
<script>
  function changeColor() {
    document.getElementById("text").style.color = "red";
  }
</script>
<p id="text">Hello, world!</p>
<button onclick="changeColor()">Change Color</button>
```

Event Handling Example

```
<script>
  function showMessage() {
    alert("You clicked the button!");
  }
</script>
<button onclick="showMessage()">Click Here</button>
```

Advantages of DHTML

- No page reload required for updates
- Better user interactivity and engagement
- Fast and responsive UI
- Enables advanced features like sliders, popups, drag-drop, etc.

Disadvantages of DHTML

- Complex for beginners to debug and maintain
- Not all older browsers may support advanced DHTML features
- Too much DHTML can affect page performance

Real-Life Use Cases

- Dynamic menus and tooltips
- Live form validation (email/password check)
- Content sliders and carousels
- Expandable/collapsible sections
- Interactive games and visualizations
- AJAX-powered web applications (e.g., Gmail, Facebook)

Table

Feature	Description
Full Form	Dynamic HTML
Key Technologies	HTML, CSS, JavaScript, DOM
Driven By	JavaScript
Main Purpose	Make web pages dynamic and interactive
Core Capabilities	Content change, style update, event handling
Use Cases	Menus, animations, form validation, pop-ups

Note: DHTML with JavaScript is a powerful combination that allows developers to build interactive, responsive, and dynamic web applications**.** It forms the foundation of modern web interactivity, and understanding how JavaScript works with HTML and CSS is key to mastering front-end development.

7.3 SUMMARY

JavaScript provides powerful built-in objects like Window**,** Navigator**,** Document**,** Form**,** Date**,** String**,** and Array that form the core of dynamic web development. The Window object serves as the global object and includes properties and methods like alert(), location, and setTimeout(). The Navigator object offers browser-specific details such as appName, userAgent, and onLine status. The Document object allows access and manipulation of HTML elements using methods like getElementById() and querySelector(). The Form object is used to access and validate form inputs dynamically. The Date object enables working with dates and times through methods like getFullYear() and setDate(). The String object allows text manipulation with methods such as toUpperCase(), replace(), and split(). Arrays, represented by the Array object, can hold multiple values and support powerful methods like map(), filter(), and reduce(). These objects enable developers to create responsive, data-driven interfaces. In addition, Dynamic HTML (DHTML) integrates HTML, CSS, JavaScript, and the DOM to create interactive and visually dynamic web pages. JavaScript is essential to DHTML, enabling real-time content updates, event handling, and style changes without reloading the page.

7.4 KEY TERMS

Window Object, Navigator Object**,** Document Object, Form Object, Date Object, String Object, Array Object, Dynamic HTML (DHTML), Document Object Model (DOM), Event Handling**.**

7.5 SELF-ASSESSMENT QUESTIONS

1. What is the role of the Window object in JavaScript?
2. How can the Navigator object be used to detect browser information?

3. How can you access and validate form data using the Form object in JavaScript?
4. What are some useful methods of the Date object to retrieve or set date values?
5. How do String methods like replace() and split() help in text manipulation?
6. What is Dynamic HTML (DHTML), and how does JavaScript contribute to it?

7. 6 FURTHER READINGS

1. JavaScript: The Definitive Guide, Seventh Edition by David Flanagan. O'Reilly Media.
2. Eloquent JavaScript: A Modern Introduction to Programming, Third Edition by Marijn Haverbeke. No Starch Press.
3. Beginning JavaScript, Fifth Edition by Jeremy McPeak. Wrox (Wiley).
4. The complete Reference Java 2 Fifth Edition by Patrick Naughton and Herbert Schildt. TMH.

Dr. Vasantha Rudramalla

LESSON- 8

XML BASICS AND DATA PROCESSING IN WEB APPLICATIONS

Aims and Objectives:

- Understand the purpose and syntax of Document Type Definition (DTD) for defining the structure of XML documents.
- Learn how to use XML Schemas (XSD) to enforce data types and validation rules in XML files.
- Explore the Document Object Model (DOM) to access, modify, and traverse XML documents programmatically.
- Gain knowledge of presenting XML datausing technologies like XSLT for transformation and display.
- Compare and utilize XML processors such as DOM and SAX for parsing and handling XML data efficiently.

STRUCTURE:

8.1 INTRODUCTION TO XML

XML (eXtensible Markup Language) is a simplified subset of SGML (Standard Generalized Markup Language), a powerful markup language adopted as a standard by the International Organization for Standardization (ISO). SGML was originally developed to add structure and formatting to data in a way that could be used across various applications.

Unlike other languages that focus on how data is displayed, XML is designed to describe what the data is. It emphasizes data structure rather than presentation. XML was introduced as a recommendation by the **World Wide Web Consortium (W3C)** to facilitate data sharing and transportation across systems in a platform-independent manner.

Markup and Tags

In XML, markup refers to the set of instructions (called **tags**) used to define elements within a document. These tags help structure the data but do not specify how it should be displayed. XML syntax closely resembles HTML, but its purpose is different: while HTML formats data for display, XML organizes data for storage and transport.

8.1.1 Basic XML Structure

Below is an example of a simple XML document:

```
<?xml version="1.0"?>
<college>
<studdetail>
<regno>05j0a1260</regno>
<name>
<firstname>karthik</firstname>
<lastname>btech</lastname>
</name>
<country name="india"/>
<branch>csit</branch>
</studdetail>
</college>
```

- The first line is a processing instruction that declares the file as an XML document and specifies the version.
- <college> is the root element, and all other elements are nested within it.

Well-Formed vs Valid XML

- A **well-formed** XML document adheres strictly to XML syntax rules:
  - All tags must be properly opened and closed.
  - Tags must not overlap.
  - Empty tags must be self-closed (e.g., <tag/>).
  - The document must include an XML declaration at the top.
- A **valid** XML document is not only well-formed but also follows a defined DTD(Document Type Definition**)** or XML Schema that specifies its structure and allowed elements. XML parsers can be used to check both well-formedness and validity.

XML Elements and Rules

XML documents are composed of:

- Elements (the core content)
- Controlinformation (like comments and declarations)
- Entities (reusable data)

Key characteristics of XML elements include:

- **Nesting Tags:** XML requires proper nesting. If one tag is opened inside another, it must be closed before the outer tag is closed.
- <parent>
- <child>Content</child>
- </parent>
- **Case Sensitivity:** XML is case-sensitive. <Name> and <name> are treated as different tags. It's a best practice to use lowercase for all tags.
- **Empty Tags:** Tags without content must be self-closed using a forward slash:
- <country name="india"/>
- **Attributes:** Elements can include attributes to hold additional information:
- <country name="india"/>

Attributes should not replace elements when the data is complex or needs further structure.

Control Information in XML

In XML, control information refers to special components that provide instructions and structure to the document. There are three main types of control information:

1. **Comments**
   Comments in XML are used to include notes or explanations within the document that are ignored by the parser.
   - Syntax: <!-- This is a comment -->
   - XML comments are similar to those in HTML.
2. **Processing Instructions (PIs)**
   These are special instructions intended for applications that process the XML file.
   - Example: <?xml version="1.0"?>
   - This declaration tells the XML processor which version of XML is being used.
3. **Document Type Declarations (DOCTYPE)**
   A Document Type Declaration links an XML document to a **DTD (Document Type Definition)**, which defines its structure and the rules for validation.
   - Syntax: <!DOCTYPE element SYSTEM "filename.dtd">
   - Example: <!DOCTYPE cust SYSTEM "customer.dtd">
   - The DTD can be either internal (within the XML) or external (in a separate file).

Entities in XML

Entities in XML are reusable content placeholders used to store small pieces of data that may be repeated throughout the document. These help in maintaining consistency and manageability.

For example, an XML document using control information and entities might look like this:

```
<?xml version="1.0"?>
<!DOCTYPE stud SYSTEM "student.dtd">
<college>
<studdetail>
<regno>mc20001</regno>
<name>
<firstname>feroz</firstname>
<lastname>pg</lastname>
</name>
<country name="india"/>
<branch>cse</branch>
</studdetail>
</college>
```

In this example:

- The <?xml version="1.0"?> is a processing instruction.
- The <!DOCTYPE stud SYSTEM "student.dtd"> links the XML file to an external DTD.
- Elements like <college> and <studdetail> are structured using rules defined in the DTD.

8.2 Document type definition

1. Introduction to DTD

Document Type Definition (DTD) defines the structure and the legal elements and attributes of an XML document. It acts as a blueprint or grammar that XML documents must follow to be considered **valid**.

While XML provides flexibility in data representation, DTD ensures data consistency, validity,and structure conformity across documents.

DTD can be written as part of the XML document (internalDTD) or in a separate file (externalDTD).

2. Purpose of DTD

- To validate the structure and content of XML documents.
- To ensure data integrity and uniformity across systems.
- To definerules for:
    - Elements and their hierarchy
    - Attributes of elements
    - Entities
    - Notations

3. Types of DTD

a. Internal DTD

Defined within the XML document itself, using the <!DOCTYPE> declaration.

Syntax:

```
<?xml version="1.0"?>
<!DOCTYPE root-element [
<!ELEMENT element-name (child-elements)>
<!ATTLIST element-name attribute-name attribute-type #default>
]>
<root-element>
  ...
</root-element>
```

Example:

```
<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
<note>
<to>Alice</to>
<from>Bob</from>
<heading>Reminder</heading>
<body>Meeting at 10 AM</body>
</note>
```

b. External DTD

Stored in a separate .dtd file and referenced from the XML document.

XML Document:

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
<to>Alice</to>
<from>Bob</from>
<heading>Reminder</heading>
<body>Meeting at 10 AM</body>
</note>
```

note.dtd (External DTD File):

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

4. DTD Components

a. Elements

Defines the allowable content of an element.

Syntax:

```
<!ELEMENT element-name (child-elements | #PCDATA | ANY | EMPTY)>
```

Examples:

```
<!ELEMENT title (#PCDATA)><!-- Text-only -->
<!ELEMENT book (title, author)><!-- Nested structure -->
<!ELEMENT page ANY><!-- Any content -->
<!ELEMENT img EMPTY><!-- Empty element -->
```

b. Attributes

Defines attributes for elements and their data types.

Syntax:

```
<!ATTLIST element-name attribute-name attribute-type default-value>
```

Attribute Types:

- CDATA: Character data
- ID: Unique identifier
- IDREF: Reference to another ID
- IDREFS: Multiple references
- NMTOKEN: Name token
- ENUMERATION: List of allowed values

Default Values:

- #REQUIRED
- #IMPLIED
- #FIXED "value"
- "default value"

Example:

<!ATTLIST book isbn CDATA #REQUIRED>
<!ATTLIST book category (fiction | nonfiction | reference) "fiction">

c. Entities

Used to define constants or placeholders that can be reused.

Syntax:

<!ENTITY entity-name "replacement-text">

Example:

<!ENTITY author "ABC">

Usage in XML:

<creator>&author;</creator>

d. Comments in DTD

<!-- This is a comment -->

5. DTD Element Content Types

Type	Description
#PCDATA	Parsed Character Data (text)
EMPTY	Element has no content
ANY	Element can contain any content
Child list	Defines specific child elements and their order

Modifiers:

- ? – zero or one
- * – zero or more
- + – one or more
- | – choice
- , – sequence

Example:

<!ELEMENT name (first, middle?, last)>
<!ELEMENT phone (home | mobile)>
<!ELEMENT address (street, city, state, zip)>

6. Advantages of DTD

- Simplicity and ease of use
- Useful for simple validation tasks
- Wide support by XML parsers
- Promotes consistency and reusability

7. Limitations of DTD

- No support for data types beyond text (e.g., integer, date)
- Limited namespace support
- Cannot enforce constraints like min/max values or string patterns
- Written in a different syntax (not XML-based)

Alternative: XML Schema (XSD) overcomes these limitations and is written in XML itself.

8. Validating XML with DTD

To validate an XML document:

- Use an XML parser (e.g., Xerces, DOM, SAX)
- Ensure the DOCTYPE declaration correctly references the DTD
- Check the document structure, elements, and attributes against the DTD rules

9. Table

Feature	Description
Full form	Document Type Definition
Purpose	Validates XML structure and content
Defined in	Internally in XML or externally as a .dtd file
Key components	Elements, attributes, entities, content models
Limitation	No support for data types or namespaces
Replacement	XML Schema (XSD) for advanced validation

Library XML with DTD:

library.xml

```
<?xml version="1.0"?>
<!DOCTYPE library SYSTEM "library.dtd">
<library>
<book isbn="12345">
<title>XML Basics</title>
<author>John Smith</author>
</book>
</library>
```

library.dtd

<!ELEMENT library (book+)>
<!ELEMENT book (title, author)>
<!ATTLIST book isbn CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>

8.3 XML Schemas

1. Introduction to XML Schema

XML Schema Definition (XSD) is a powerful way to define the structure, content, and semantics of XML documents. It is a recommendation by the World Wide Web Consortium (W3C)and is considered more powerful and expressive than Document Type Definition (DTD).

Purpose of XML Schema

- To define the structure of an XML document.
- To define the datatypes of elements and attributes.
- To validate whether an XML document adheres to a specific format.
- To support namespace and extensibility features.

2. Advantages of XML Schema over DTD

Feature	DTD	XML Schema (XSD)
Syntax	SGML-based	XML-based
Data types	Not supported	Strongly supported (e.g., string, int)
Namespaces	Not supported	Fully supported
Custom types	Not available	Available
Reuse of components	Limited	Extensive (via complex types, imports)

3. XML Schema Syntax

XML Schema documents are XML files that define:

- Elements
- Attributes
- Data types
- Element relationships

Basic Structure of XML Schema

<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- Definitions go here -->
</xs:schema>

4. Defining Elements and Attributes

Defining a Simple Element

```
<xs:element name="studentName" type="xs:string"/>
```

Defining an Attribute

```
<xs:attribute name="id" type="xs:integer"/>
```

5. Complex Types

Used to define elements that contain:

- Other elements
- Attributes

Example: Complex Type with Child Elements

```
<xs:element name="student">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

Example: Complex Type with Attributes

```
<xs:element name="book">
<xs:complexType>
<xs:attribute name="isbn" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
```

6. Data Types in XML Schema

XML Schema provides many built-in data types, categorized as:

Primitive Types

- xs:string
- xs:integer
- xs:boolean
- xs:decimal
- xs:date
- xs:time

Derived Types

- xs:positiveInteger
- xs:nonNegativeInteger
- xs:token
- xs:ID

7. Occurrence Constraints

To control the number of times an element can occur:

- **minOccurs** – Minimum number of occurrences
- **maxOccurs** – Maximum number of occurrences

Example

<xs:element name="phone" type="xs:string" minOccurs="0" maxOccurs="3"/>

8. Restriction and Facets

To place constraints on data values using facets:

Example: Restricting String Length

<xs:simpleType name="usernameType">
<xs:restriction base="xs:string">
<xs:minLength value="5"/>
<xs:maxLength value="12"/>
</xs:restriction>
</xs:simpleType>

9. Reusing Schema Components

- **Named Types**: Reuse custom types across the schema
- **Include / Import**: Modularize large schemas

Include Another Schema

<xs:include schemaLocation="commonTypes.xsd"/>

10. Namespaces in XML Schema

Namespaces prevent element name conflicts.

Example

<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.com/student"
xmlns="http://www.example.com/student"
elementFormDefault="qualified">

11. Validating XML with Schema

XML files can be validated against an XSD to ensure structural and data correctness using tools like:

- XML parsers (e.g., Xerces, XMLSpy)
- IDEs (e.g., Eclipse, IntelliJ)
- Java (via JAXP)

12. Example: XML and XSD

Sample XML Document

```
<student>
<name>John</name>
<age>21</age>
</student>
```

Corresponding XSD

```
<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="student">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

13. Tools Supporting XML Schema

- **Editors**: Oxygen XML Editor, XMLSpy
- **Parsers**: Xerces, SAXON
- **Programming APIs**:
  - Java: JAXP**,** JAXB
  - .NET: XmlSchemaSet**,** XDocument

Note:

XML Schema is a robust and feature-rich way to define the structure and constraints of XML documents. It supports a wide range of data types, complex content modeling, namespaces, and extensibility, making it ideal for applications requiring strict data validation and integration across systems.

8.4 Document Object model

1. Introduction to DOM

The **Document Object Model (DOM)** is a W3C standard that defines a platform- and language-neutral interfaceto access and manipulate the content, structure, and style of XML or HTML documents.

- It represents a document as a tree structure.
- Each part of the document (elements, attributes, text) is a node in the tree.
- DOM allows programsand scripts to dynamically access and update the document's content, structure, and style.

2. Key Features of DOM

- **Tree Structure**: Represents documents as a hierarchy of nodes.
- **Language-Independent**: DOM can be used in Java, JavaScript, Python, etc.
- **Dynamic**: Allows dynamic modification of documents.
- **Standardized**: Defined by the W3C DOM Specification.

3. DOM Tree Structure

DOM views an XML/HTML document as a **tree of nodes**.

Types of Nodes

Node Type	Description
Document Node	Root of the document tree
Element Node	Represents XML/HTML elements
Attribute Node	Represents attributes of elements
Text Node	Represents text content
Comment Node	Represents comments
Processing Instruction	Represents special instructions

Example XML

```
<student>
<name>John</name>
<age>21</age>
</student>
```

Corresponding DOM Tree

```
Document
└── Element: student
    ├── Element: name
    │   └── Text: John
    └── Element: age
```

└── Text: 21

4. DOM Levels

DOM has been standardized in **multiple levels**:

- **DOM Level 1**: Core functionalities – tree structure, basic node access.
- **DOM Level 2**: Adds events, style, and support for namespaces.
- **DOM Level 3**: Adds support for loading/saving documents, validation.

5. Accessing DOM with Java (Using JAXP)

Java provides **JAXP (Java API for XML Processing)** to work with DOM.

Steps to Use DOM in Java

1. **Create DocumentBuilderFactory**
2. **Create DocumentBuilder**
3. **Parse the XML file to get Document**
4. **Traverse or modify the DOM tree**

Java Example: Reading XML using DOM

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;

public class DOMReadExample {
   public static void main(String[] args) throws Exception {
      File inputFile = new File("student.xml");
      DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
      DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
      Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();

      System.out.println("Root element: " + doc.getDocumentElement().getNodeName());

      NodeList nodeList = doc.getElementsByTagName("student");
      for (int i = 0; i < nodeList.getLength(); i++) {
         Node node = nodeList.item(i);
         if (node.getNodeType() == Node.ELEMENT_NODE) {
            Element element = (Element) node;
            System.out.println("Name: " +
element.getElementsByTagName("name").item(0).getTextContent());
            System.out.println("Age: " +
element.getElementsByTagName("age").item(0).getTextContent());
         } } }}
```

6. Common DOM Interfaces (Java - org.w3c.dom)

Interface	Description
Node	Base interface for all nodes
Element	Represents an element
Attr	Represents an attribute
Text	Represents text within elements
Document	Represents the entire XML/HTML document
NodeList	A list of nodes
NamedNodeMap	Map for attributes

7. DOM Operations

a. Traversing Nodes

- getFirstChild(), getLastChild()
- getNextSibling(), getPreviousSibling()
- getParentNode(), getChildNodes()

b. Modifying Document

- createElement(), createTextNode()
- appendChild(), removeChild()
- setAttribute(), removeAttribute()

c. Reading Data

- getNodeName(), getNodeValue(), getNodeType()
- getTextContent(), getElementsByTagName()

8. Advantages of DOM

- **Random Access**: Any node can be accessed anytime.
- **Modifiable**: Nodes can be added, updated, or deleted.
- **Standard Interface**: Supported across multiple languages.
- **Rich Functionality**: Allows full document manipulation.

9. Disadvantages of DOM

- **Memory Intensive**: Loads the entire document into memory.
- **Slower for Large Files**: Not suitable for very large XML documents.
- **More Complex**: DOM API can be verbose and complex for beginners.

10. DOM vs SAX

Feature	DOM	SAX
Parsing Mode	Loads entire document in memory	Event-based (reads sequentially)
Access	Random access to any part	Sequential access only
Modification	Supports document modification	Read-only
Performance	Slower for large files	Faster and memory-efficient

11. DOM in Web Browsers (JavaScript)

DOM is also widely used in web browsers via JavaScript to manipulate HTML documents dynamically.

Example (HTML + JavaScript)

```
<p id="demo">Hello</p>
<script>
document.getElementById("demo").innerHTML = "Hello, DOM!";
</script>
```

Note:

The **Document Object Model (DOM)** is a critical concept for working with both XML and HTML documents. It offers a structured, object-oriented view of a document, enabling dynamic access and manipulation. While powerful, it should be used carefully for large documents due to memory and performance considerations.

8.5 Presenting XML

1. Introduction

Presenting XML refers to the methods and technologies used to display or render the data stored in XML (eXtensible Markup Language) documents in a human-readable and visually appealing format. By itself, XML is only a data representation format; it does not define how **the** data should appear on screen or on paper.

To make XML content presentable, we need to use associated technologies that can transform or style XML data into HTML, PDF, plain text, or other output formats.

2. Need for Presenting XML

- XML is self-descriptive but not inherently visual.

- For users to understand or interact with XML content, it must be transformed into a user-friendly format**.**
- Enables data sharing across different platforms and presentation customization**.**

3. Techniques for Presenting XML

There are several technologies and methods for presenting XML data effectively:

A. Using XSLT (Extensible Stylesheet Language Transformations)

- **XSLT** is a W3C standard used to transform XMLdata into other formats such as HTML, text, or another XML structure.
- It works by applying templates and rules to match elements in the XML file and transform them accordingly.

Example

XML File: students.xml

```
<students>
<student>
<name>John</name>
<age>21</age>
</student>
</students>
```

XSLT File: students.xsl

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>Student Information</h2>
<table border="1">
<tr><th>Name</th><th>Age</th></tr>
<xsl:for-each select="students/student">
<tr>
<td><xsl:value-of select="name"/></td>
<td><xsl:value-of select="age"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Linking XML with XSLT
<?xml-stylesheet type="text/xsl" href="students.xsl"?>

B. Using CSS with XML

- **CSS (Cascading Style Sheets)** can be used to apply basic formatting to XML documents, just like in HTML.
- XML must be well-structured and follow a specific format for CSS to work effectively.
- Best suited for simple styling (fonts, colors, borders).

Example

XML
<?xml-stylesheet type="text/css" href="style.css"?>
<note>
<to>Tina</to>
<from>John</from>
<body>Hello, how are you?</body>
</note>

CSS (style.css)
note {
  display: block;
  background-color: lightyellow;
  padding: 10px;
  font-family: Arial;
}

to, from, body {
  display: block;
  margin: 5px 0;
}

C. Converting XML to HTML via Programming

- You can use programming languages like Java, Python, PHP, or JavaScript to read XML data and present it as HTML.
- Commonly used in web applications and dynamic content generation.

Example: Using JavaScript
<script>
fetch('students.xml')
.then(response => response.text())
.then(data => {

```
    const parser = new DOMParser();
    const xmlDoc = parser.parseFromString(data, "text/xml");
    const name = xmlDoc.getElementsByTagName("name")[0].childNodes[0].nodeValue;
document.body.innerHTML = "<h1>Student: " + name + "</h1>";
  });
</script>
```

D. Converting XML to PDF or Other Formats

- XML can be converted to **PDF** using tools like **Apache FOP (Formatting Objects Processor)** and **XSL-FO (XSL Formatting Objects)**.
- Common in business applications where XML-based data must be printed or archived.

Workflow

1. XML + XSL-FO → Apache FOP → PDF

4. Technologies Involved in XML Presentation

Technology	Purpose
XSLT	Transforms XML into HTML, text, or other XML
XSL-FO	Converts XML into print formats like PDF
CSS	Styles XML elements
JavaScript	Dynamically reads and displays XML in web pages
Apache FOP	Converts XSL-FO documents into PDF
Programming APIs	Java (JAXP), Python (lxml), etc., for customized rendering

5. Presentation Best Practices

- Always validate XML before presenting.
- Use XSLT for complex formatting and CSS for simple visual enhancements**.**
- Ensure cross-browser compatibility when using XML on the web.
- For large documents, consider server-side transformation for better performance.
- Use responsive and accessible design when converting to HTML.

6. Applications of Presenting XML

- Web content management systems
- Online reporting systems
- E-commerce catalogs
- Invoice and billing systems
- Educational content presentation
- News and media feeds (RSS/Atom)

Presenting XML is essential to make XML data usable and understandable for human users. Whether through XSLT, CSS, or programmingtechniques, transforming XML into a readable format bridges the gap between raw structured data and practical user applications.

By leveraging the right tools and technologies, XML data can be presented effectively across web, print, and mobile platforms.

1. Introduction

An **XML Processor** is a software component or engine that reads, interprets, and processes XML documents according to defined standards (like XML 1.0 by W3C). It ensures that an XML document is well-formed, and optionally,valid according to a DTD or XML Schema.

The XML processor is also commonly referred to as an XML parser.

2. Types of XML Processors

XML processors are categorized into two main types based on how they access and process the document:

A. Validating Processor

- Checks both well-formedness and validity.
- Validates the document against a DTD or XML Schema.
- Reports errors if the document doesn't conform to the structure.

B. Non-Validating Processor

- Only checks for well-formedness.
- Does not validate against any DTD or Schema.
- Faster and simpler, suitable when validation isn't required.

3. Functions of an XML Processor

Function	Description
Parsing	Reads XML text and constructs a tree or events
Validation	Checks XML against a DTD or Schema (optional)
Reporting Errors	Identifies and reports syntax or structure errors
Providing Interfaces	Supplies access through APIs like DOM or SAX
Data Extraction	Enables retrieval of specific information from XML documents

4. Common XML Processing Models

There are two primary programming models for using XML processors:

A. DOM (Document Object Model)

- Tree-based processing.
- Loads the entire XML document into memory as a tree structure.
- Allows random access, traversal, and modification.

Pros:

- Easy to use and understand.
- Supports both reading and writing.

Cons:

- High memory usage for large documents.

B. SAX (Simple API for XML)

- Event-based processing.
- Parses the document sequentially and generates events (startElement, endElement, etc.).
- No tree is built; ideal for read-only, forward-only access**.**

Pros:

- Fast and memory-efficient.
- Good for large files.

Cons:

- More complex to code.
- No backward access or modification.

5. Popular XML Processors

Processor	Language	Type	Description
Xerces	Java, C++	Validating	Apache XML processor supporting DOM, SAX, Schema
MSXML	C++, COM, VB	Validating	Microsoft XML Parser for Windows platforms
libxml2	C	Validating	Open-source XML parser from the GNOME project
Expat	C	Non-validating	Fast, lightweight stream-oriented parser
JAXP	Java	Both	Java API for XML Processing (uses DOM, SAX, StAX)
lxml	Python	Validating	Powerful Python binding for libxml2 and libxslt

6. Using XML Processors in Java (JAXP)

Java provides JAXP (Java API for XML Processing) which supports DOM, SAX, and StAX models.

Example: Using DOM Parser

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;

public class DOMParserExample {
   public static void main(String[] args) throws Exception {
      File inputFile = new File("students.xml");
      DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
      DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
      Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();

      System.out.println("Root element: " + doc.getDocumentElement().getNodeName());
   }
}
```

Example: Using SAX Parser

```
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;

public class SAXParserExample {
   public static void main(String[] args) throws Exception {
      SAXParserFactory factory = SAXParserFactory.newInstance();
      SAXParser saxParser = factory.newSAXParser();

      DefaultHandler handler = new DefaultHandler() {
         public void startElement(String uri, String localName, String qName, Attributes
attributes) {
            System.out.println("Start Element: " + qName);
         }
      };

      saxParser.parse("students.xml", handler);
   }
}
```

7. Well-Formedness vs. Validity

Criterion	Well-Formed XML	Valid XML
Syntax Rules	Must follow basic syntax	Must follow syntax and structure
DTD or Schema	Not required	Required for validation
Checked by	All processors	Only validating processors

8. Error Handling in XML Processors

- **Well-formedness errors**: Missing tags, improper nesting, etc.
- **Validation errors**: Mismatch with DTD/Schema definitions.
- Most processors provide mechanisms to report**,** log, and sometimes recover from errors.

9. Performance Considerations

Model	Memory Use	Speed	Use Case
DOM	High	Medium	Small to medium documents
SAX	Low	High	Large, read-only documents
StAX	Medium	High	Event-driven, pull-based parsing

10. Applications of XML Processors

- Web services (SOAP, REST)
- Configuration files (Spring, Maven)
- Data exchange between systems
- Digital publishing
- Document storage and retrieval
- Enterprise systems integration

Note: An XMLprocessor is an essential tool for working with XML documents. It validates, parses, and provides programmatic access to the data, enabling applications to read**,** transform**, and** presentXML content.

Choosing the right processor (DOM, SAX, or StAX) depends on the application requirements**,** such as performance, memory constraints, and the need for validation.

8.6 DOM and SAX

DOM and SAX in XML Processing

1. Introduction

When working with XML documents programmatically, two primary APIs are commonly used for parsing and processing:

- DOM (Document Object Model)
- SAX (Simple API for XML)

Both provide ways to access the data and structure of XML documents, but they do so using different approaches suited for different use cases.

2. What is DOM (Document Object Model)?

Definition:

DOM is a tree-based parsing method. It represents the entire XML document as a hierarchical tree of nodes in memory. This allows developers to access, navigate, and manipulate any part of the XML document at any time.

How It Works:

- Loads the entire XML document into memory.
- Constructs a tree where each element, attribute, or text is a node.
- Provides random access to any node.

Key Features:

- Tree structure (parent-child relationships)
- Allowsread and write access
- Supports navigation and modification

DOM Parser Workflow:

1. Parse the XML document.
2. Create a tree (DOM tree) in memory.
3. Access or modify nodes using methods.

DOM Example in Java:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;
public class DOMExample {
   public static void main(String[] args) throws Exception {
      File file = new File("students.xml");
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();
      Document doc = builder.parse(file);
      NodeList list = doc.getElementsByTagName("student");
      for (int i = 0; i <list.getLength(); i++) {
         Element student = (Element) list.item(i);
         System.out.println("Name: " +
student.getElementsByTagName("name").item(0).getTextContent());
      }
}
}
```

3. What is SAX (Simple API for XML)?

Definition:

SAX is an event-based parsing method. Instead of building a tree, it reads the XML document sequentially and fires events (start element, end element, characters) when it encounters different components of the document.

How It Works:

- Reads the XML document line by line**.**
- Generates events such as:
  - startElement()
  - characters()
  - endElement()
- The application must handle these events**.**

Key Features:

- Does not load the entire document into memory.
- Suitable for large XML documents.
- Fast and memory-efficient**.**

SAX Parser Workflow:

1. Set up a handler to listen for XML events.
2. Parse the document.
3. React to events like element starts and ends.

SAX Example in Java:

```
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;
public class SAXExample {
   public static void main(String[] args) throws Exception {
      SAXParserFactory factory = SAXParserFactory.newInstance();
      SAXParser parser = factory.newSAXParser();
      DefaultHandler handler = new DefaultHandler() {
         public void startElement(String uri, String localName, String qName, Attributes
attributes) {
            if (qName.equalsIgnoreCase("name")) {
               System.out.print("Name: ");
            }
         }
         public void characters(char ch[], int start, int length) {
            System.out.println(new String(ch, start, length));
         }
      };

parser.parse("students.xml", handler);
   }
}
```

4. Comparison Between DOM and SAX

Feature	DOM	SAX
Model	Tree-based	Event-based
Memory Usage	High (loads whole XML in memory)	Low (sequential reading)
Speed	Slower for large files	Faster for large files
Access Type	Random access to any part	Sequential access only
Modification	Supports modification	Read-only
Complexity	Easier to implement	More complex due to event handling
Use Case	Small to medium-sized documents	Large documents, stream processing
Navigation	Built-in methods (e.g., getChild)	No navigation—developer handles flow

5. When to Use DOM

- You need to modify or update XML data.
- Random access to nodes is necessary.
- The XML document is smallormedium-sized.
- You want a simpler programming model**.**

6. When to Use SAX

- The XML document is very large.
- You only need to read data (not modify).
- You require faster processing.
- Memory usage must be minimal.

7. Hybrid Approach

Some applications use a combination of DOM and SAX, known as StAX (Streaming API for XML), which gives the developer more control (pull-based parsing) over the event stream.

Note:BothDOM and SAX provide powerful ways to work with XML data, but their usage depends on the requirements:

- Use DOM when you need full access**,** modification, and tree-based structure.
- Use SAX when you want efficient**,** stream-based**,** andlightweight processing of large XML documents.

Understanding the difference helps in choosing the right parser for your application's performance, memory, and complexity needs.

8.7 Summary

XML (eXtensible Markup Language) is a standardized language used to structure, store, and transport data. A Document Type Definition (DTD**)** defines the legal building blocks of an XML document by specifying its structure with a list of allowed elements and attributes. In contrast, XML Schemas are more powerful than DTDs, allowing data type definitions, namespaces, and detailed validation rules using XML syntax itself. The Document Object Model (DOM**)** is a tree-based representation of an XML document in memory, allowing developers to access, modify, and navigate the document structure dynamically. Presenting XML involves transforming XML content into human-readable formats using tools like CSS for styling and XSLT (Extensible Stylesheet Language Transformations) for converting XML into HTML or other formats. XML data is not typically presented directly; it requires formatting or transformation to be user-friendly.To work with XML programmatically, XML Processors (or parsers) are used. These processors validate the document's structure and convert it into usable data formats for applications. There are two main types of processors: DOMand SAX. The DOMparser reads the entire XML into memory and builds a node tree, suitable for editing and random access. On the other hand, the SAX parser is event-driven, reading the document sequentially and firing events during parsing, which is ideal for large, read-only documents. DOM is easier for manipulation but memory-intensive, while SAX is faster and more efficient for large-scale XML. Understanding these technologies is essential for developers working with XML in real-world applications.

8. 8 Key Terms

XML, DTD, XML Schema (XSD), Well-formed XML, Valid XML, DOM, SAX, XML Processor, XSLT, Namespace.

8. 9 Self-Assessment Questions

1. What is the purpose of a Document Type Definition (DTD) in XML?
2. How does an XML Schema differ from a DTD?
3. What is meant by a well-formed XML document?
4. What is the role of the Document Object Model (DOM) in XML processing?
5. How does SAX process XML documents differently from DOM?
6. What is an XML Processor, and what are its main types?
7. How can XML data be presented to users in a readable format?

8.10Further Readings

1. Beginning XML, Fifth Edition by David Hunter, Jeff Rafter, and Joe Fawcett. Wiley Publishing.
2. Learning XML, Second Edition by Erik T. Ray. O'Reilly Media.
3. XML in a Nutshell, Third Edition by Elliotte Rusty Harold and W. Scott Means. O'Reilly Media.
4. Internet and World Wide Web: How to Program, Third Edition by Paul Deitel and Harvey Deitel. Pearson Education.

Dr. Vasantha Rudramalla

LESSON- 9

INTRODUCTION TO JDBC AND DATABASE CONNECTIVITY IN JAVA

Aims and Objectives:

- Gain insight into JDBC and recognize its role as a standardized interface for database interaction in Java.
- Identify and understand the key steps required to connect Java applications with relational databases.
- Investigate the different types of JDBC drivers and how they support communication between Java and databases.
- Examine the inner workings and stages of a JDBC connection from initiation to termination.
- Learn to create, handle, and optimize database connections efficiently within Java-based systems.

STRUCTURE:

9.1 JDBC: INTRODUCTION TO JDBC

JDBC (Java Database Connectivity) is a Java-based API (Application Programming Interface) that enables Java applications to interact with databases in a platform-independent and standardized way. Developed by Sun Microsystems (now part of Oracle Corporation), JDBC is part of the Java Standard Edition and is widely used for building robust, data-driven applications.

9.1.1Purpose of JDBC

The main purpose of JDBC is to allow Java applications to perform the following operations on relational databases:

- Establish a connection to a database
- Send SQL queries and update statements
- Retrieve and process the results of SQL queries

- Handle errors and exceptions
- Perform transaction management

Need for JDBC

Before JDBC, accessing databases in Java required platform-specific and third-party APIs, making applications less portable and harder to maintain. JDBC solves this by providing a uniform interface for accessing different types of relational databases like MySQL, Oracle, PostgreSQL, SQL Server, etc., using the same code structure.

JDBC Architecture

JDBC architecture consists of two layers:

1. **JDBC API:** This provides the application-level interface for Java developers to write database code.
2. **JDBC Driver:** This handles the communication between the Java application and the actual database. The driver translates JDBC calls into database-specific calls.

The JDBC API utilizes a **DriverManager** along with database-specific drivers to enable seamless and transparent connectivity to various types of relational databases, regardless of vendor differences.The **DriverManager** is responsible for selecting and managing the appropriate driver for each database connection request. It supports the operation of multiple drivers simultaneously, allowing connections to several heterogeneous databases within a single Java application.

The following architectural figure: 9.1 illustrates the position of the DriverManager in relation to the JDBC drivers and the Java application:

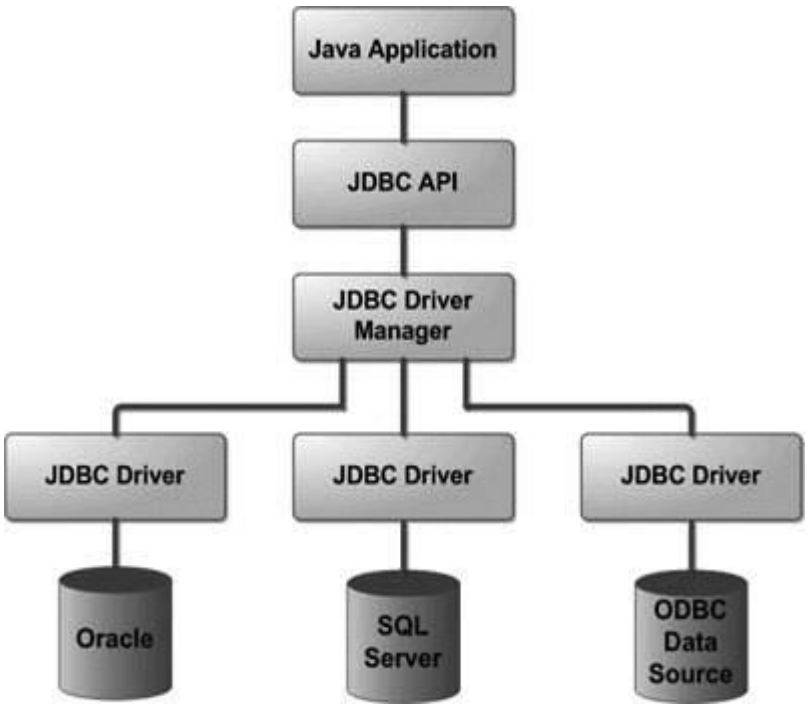


Figure: 9.1 the position of the DriverManager in relation to the JDBC drivers and the Java application:

Core JDBC Components

1. **DriverManager:** Manages the list of database drivers and establishes a connection to the database.
2. **Connection:** Represents a session with a specific database.
3. **Statement:** Used to execute SQL queries.
4. **PreparedStatement:** A subclass of Statement that allows precompiled queries with parameters.
5. **CallableStatement:** Used to execute stored procedures.
6. **ResultSet:** Represents the result of a SQL query and allows traversal through query results.
7. **SQLException:** Handles database-related exceptions and errors.

What is a JDBC Driver?

A **JDBC Driver** is a software component that enables Java applications to interact with a specific database. It acts as a bridge between the Java application and the database**,** translating the standard JDBC API calls into database-specific calls that the database can understand and execute.

Since different databases (like MySQL, Oracle, SQL Server, PostgreSQL, etc.) use different communication protocols, JDBC drivers are tailored for each type of database.

9.1.2 Types of JDBC Drivers

There are four types of JDBC drivers, also known as JDBC driver types:

1. **Type 1:** JDBC-ODBC Bridge Driver

In a **Type 1 JDBC driver**, a JDBC-ODBC bridge is used to connect Java applications to databases through existing ODBC (Open Database Connectivity**)** drivers installed on each client machine. To use this approach, the system must be configured with a Data Source Name (DSN), which identifies the target database.When Java was first introduced, this driver type was practical because most relational databases primarily supported ODBC for connectivity. However, with the evolution of native JDBC drivers, the Type 1 driver is now considered outdated and is recommended only for testing or experimental purposes or in cases where no other driver is available.

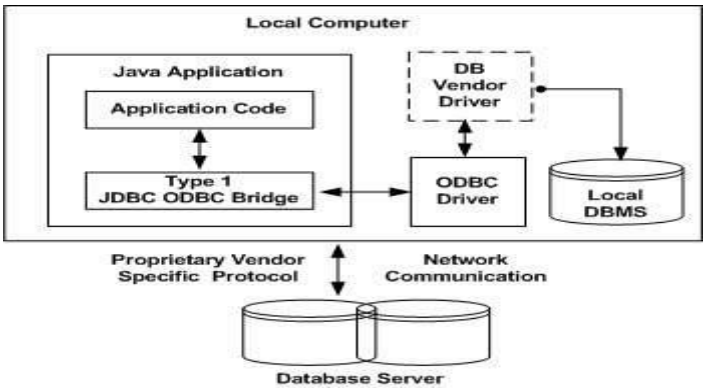


Figure: 9.2 Type 1: JDBC-ODBC Bridge Driver

An example of this type of driver is the JDBC-ODBC Bridge provided with JDK 1.2.

2. **Type 2:** Native-API Driver

A **Type 2 JDBC driver** converts JDBC API calls into native C/C++ API calls that are specific to the database being used. These drivers are usually provided by the database vendors and function similarly to the JDBC-ODBC Bridge, but without relying on ODBC. However, they require the native driver to be installed on each client machine.

Since these drivers are database-specific, switching to a different database would require replacing the underlying native APIs, making them less portable. Although largely outdated today, Type 2 drivers can offer better performance than Type 1 drivers by avoiding the overhead introduced by ODBC.

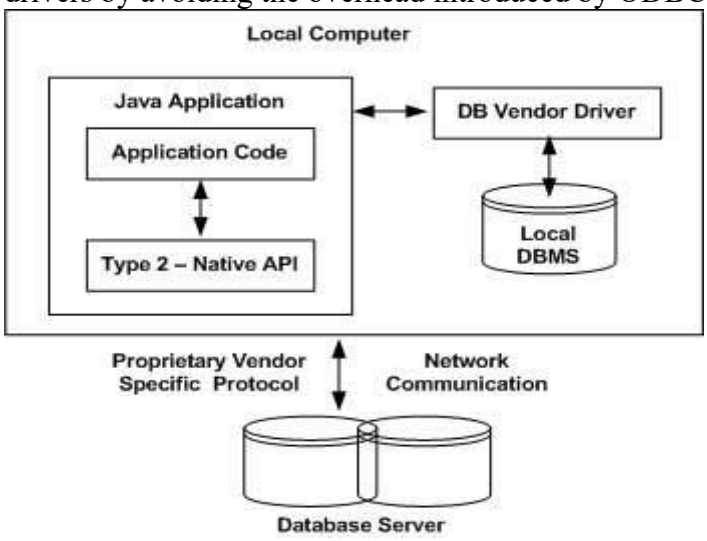


Figure9.3 Type 2: Native-API Driver

A well-known example of a Type 2 driver is the Oracle Call Interface (OCI) driver.

3. **Type 3:** Network Protocol Driver

A **Type 3 JDBC driver** follows a three-tier architecture to access databases. In this setup, the JDBC client communicates with a middleware application server using standard network **sockets**. The middleware server then translates these requests into database-specific calls and forwards them to the appropriate database server.

This driver type is highly flexible and scalable, as it does not require any database-specific code on the client side. A single Type 3 driver can provide access to multiple types of databases through the middleware. Essentially, the application server acts as a JDBC proxy, handling database operations on behalf of the client.

To effectively use a Type 3 driver, you must be familiar with the configuration of the middleware server. Internally, the server may use a Type 1, 2, or 4 driver to communicate with the actual database, so understanding how it is set up can help optimize performance and compatibility.

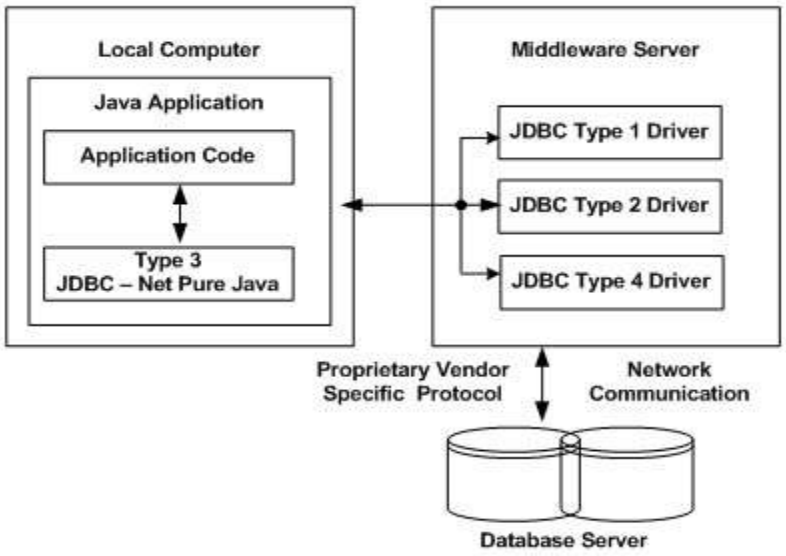


Figure: 9.4 Type 3: Network Protocol Driver

4. **Type 4:** Thin Driver (Pure Java driver)

A **Type 4 JDBC driver** is a pure Java driver that communicates directly with the database **server** using the database vendor's native network protocol over a socket connection. This type of driver offers the highest performance and is typically provided by the database vendor.

Type 4 drivers are highly portable and easy to use, as they require no additional software installation on either the client or server. Additionally, they can often be dynamically downloaded at runtime, further simplifying deployment. An example of a Type 4 driver is MySQL's Connector J. Due to the proprietary nature of database communication protocols, these drivers are usually developed and maintained by the database vendors themselves.

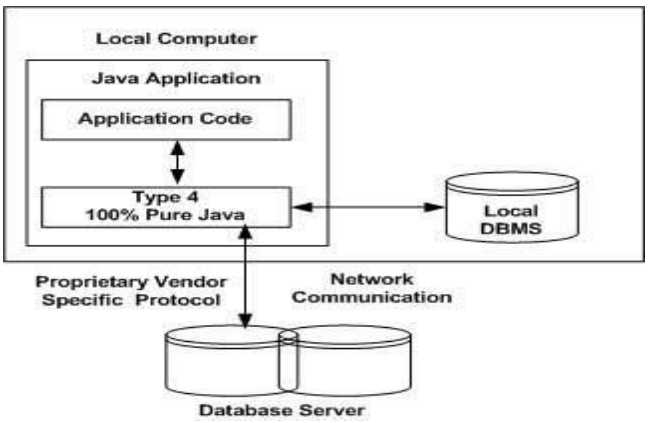


Figure 9.5: Type 4: Thin Driver (Pure Java driver)

Note:

A JDBC driver is essential for establishing communication between a Java application and a relational database. Choosing the right driver type depends on factors like performance,

portability, and the specific use case. Type 4 drivers are typically preferred for their simplicity and cross-platform compatibility.

Basic JDBC Workflow

1. Load the JDBC driver.
2. Establish a connection using DriverManager.getConnection().
3. Create a Statement or PreparedStatement.
4. Execute a query using executeQuery() or update using executeUpdate().
5. Process the ResultSet.
6. Close the connection and other resources.

Example Code Snippet

```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
      try {
         // Load the JDBC driver
         Class.forName("com.mysql.cj.jdbc.Driver");
         // Establish the connection
         Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydatabase", "username", "password");

         // Create a statement
         Statement stmt = conn.createStatement();

         // Execute a query
         ResultSet rs = stmt.executeQuery("SELECT * FROM students");

         // Process the result set
         while (rs.next()) {
            System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
         }

         // Close resources
         rs.close();
         stmt.close();
         conn.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Advantages of JDBC

- Platform-independent database connectivity
- Enables connection to various databases using a uniform API
- Supports basic and advanced SQL operations
- Easy integration with enterprise Java technologies (JSP, Servlets, Spring, etc.)
- Good performance with Type 4 drivers

Note: JDBC is a powerful and essential API for Java developers who need to interact with relational databases. It abstracts the complexity of database communication and provides a clean, simple, and extensible framework for data access, making it a cornerstone of Java database programming.

9.2 Connections

Establishing a JDBC Connection in Java

Establishing a JDBC (Java Database Connectivity) connection in Java is a straightforward process involving a few essential steps. These steps include importing the required packages, registering the JDBC driver, formulating the database URL, creating a connection object, and finally, closing the connection properly.

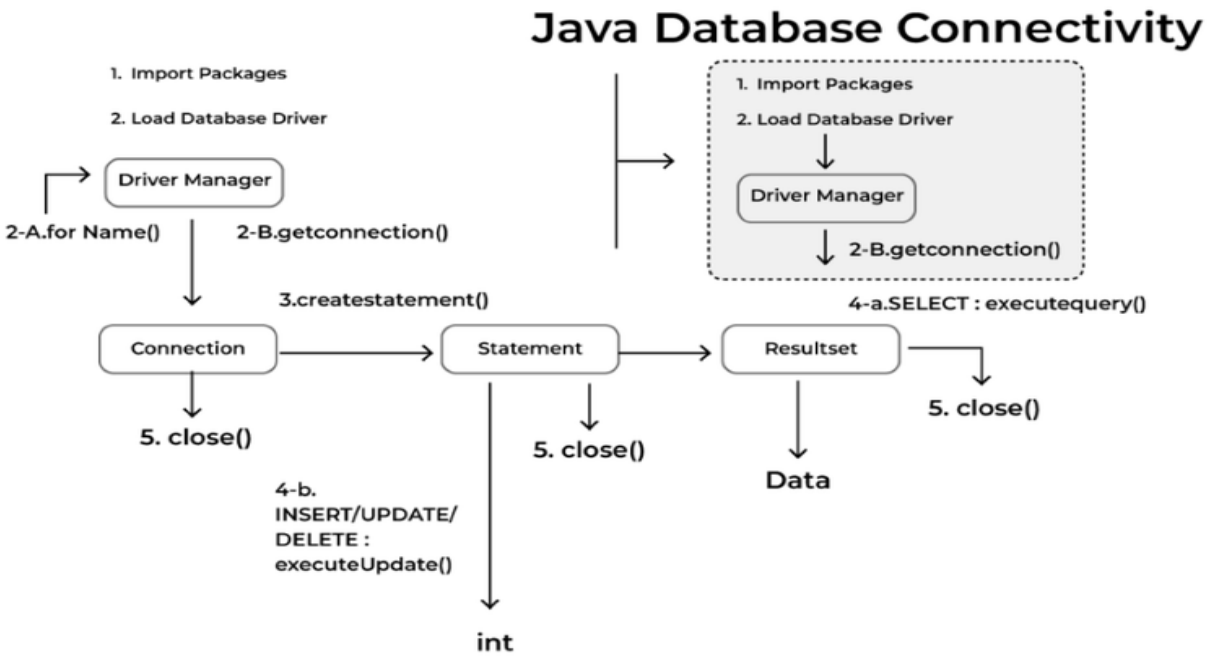


Figure 9.6: Establishing a JDBC Connection in Java

Step 1: Import JDBC Packages

Before working with JDBC, you need to import the necessary classes provided by the Java API. These imports allow your program to interact with databases by enabling operations like querying, inserting, updating, and deleting data.

Add the following import statements at the beginning of your Java source file:

```
import java.sql.*;   // For standard JDBC classes
import java.math.*;  // For BigDecimal and BigInteger support
```

Step 2: Register the JDBC Driver

Registering the JDBC driver is the process of loading the database-specific driver class into memory. This step must be performed only once in your program. There are two approaches to register a driver:

Approach I – Using Class.forName()

This is the most commonly used and portable method.

```
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (ClassNotFoundException ex) {
  System.out.println("Error: unable to load driver class!");
  System.exit(1);
}
```

For non-compliant JVMs, you can use newInstance() with exception handling:

```
try {
  Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
} catch (ClassNotFoundException ex) {
  System.out.println("Error: unable to load driver class!");
  System.exit(1);
} catch (IllegalAccessException ex) {
  System.out.println("Error: access problem while loading!");
  System.exit(2);
} catch (InstantiationException ex) {
  System.out.println("Error: unable to instantiate driver!");
  System.exit(3);
}
```

Approach II – Using DriverManager.registerDriver()

This method is useful for non-JDK compliant JVMs:

```
try {
  Driver myDriver = new oracle.jdbc.driver.OracleDriver();
  DriverManager.registerDriver(myDriver);
} catch (Exception ex) {
  System.out.println("Error: unable to register driver!");
```

```
  System.exit(1);
}
```

Step 3: Formulate the Database URL

A database URL is used to specify the location of the database to which you want to connect. The DriverManager.getConnection() method requires this URL.

Common URL formats for various databases are:

RDBMS	JDBC Driver Class	URL Format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/databaseName
Oracle	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname:port/databaseName

Note: Replace hostname, port, and databaseName with actual values based on your database configuration.

Step 4: Create the Connection Object

To establish a connection, use one of the three overloaded versions of the DriverManager.getConnection() method:

1. Using Database URL, Username, and Password

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password";
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

2. Using a Database URL with Embedded Credentials

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

3. Using a Database URL and Properties Object

```
import java.util.*;
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties();
info.put("user", "username");
info.put("password", "password");
Connection conn = DriverManager.getConnection(URL, info);
```

Step 5: Close the Connection

After completing database operations, it is crucial to close the connection to free up database resources. Never rely on Java's garbage collector for this task, as it is a poor programming practice.

Always use the close() method in a finally block to ensure the connection is properly closed:

```
try {
  // database operations
} finally {
  if (conn != null) conn.close();
}
```

Closing connections properly helps avoid memory leaks and ensures efficient use of DBMS resources.

Note:

In JDBC programming, establishing a connection follows five main steps:

1. **Import** the required JDBC classes.
2. **Register** the JDBC driver.
3. **Formulate** the database URL correctly.
4. **Create** the connection using DriverManager.getConnection().
5. **Close** the connection properly after use.

By following this structure, your JDBC application will maintain reliability, portability, and good resource management practices.9.3 Internal Database Connections

In Java applications, internal database connections refer to the underlying process of how a connection is established between a Java program and a database using the JDBC API. Internally, JDBC acts as a bridge between the Java application and the database, abstracting the complexity of database communication through a set of well-defined interfaces and classes.

Key Components Involved

1. **DriverManager Class**
   - Manages a list of database drivers.
   - Matches the connection request with the appropriate driver using the JDBC URL.
   - Returns a Connection object to the application.
2. **Driver Interface**
   - Every JDBC driver must implement the java.sql.Driver interface.
   - When the driver class is loaded, it automatically registers itself with DriverManager.
3. **Connection Interface**
   - Represents a session with the database.

- Provides methods for creating Statement, PreparedStatement, and CallableStatement objects.

Internal Connection Flow

1. **Load the JDBC Driver**
2. Class.forName("com.mysql.cj.jdbc.Driver");
   - Registers the driver with DriverManager.
   - Enables the driver to handle future connection requests.
3. **Establish the Connection**
4. Connection con = DriverManager.getConnection(
5. "jdbc:mysql://localhost:3306/mydatabase", "username", "password");
   - The DriverManager scans through registered drivers.
   - The driver that understands the URL (jdbc:mysql://...) takes over.
6. **Driver Internally Performs:**
   - **URL Parsing**: Validates if the URL is compatible.
   - **Authentication**: Uses username and password to connect.
   - **Socket Creation**: Establishes a network socket to the database server.
   - **Protocol Handling**: Implements database-specific communication protocol.
7. **Return Connection Object**
   - Once the connection is successfully established, a Connection object is returned to the Java program.
   - The application can now execute SQL queries.

Example Code

```
import java.sql.*;

public class InternalConnectionExample {
   public static void main(String[] args) {
      try {
         // Step 1: Load JDBC driver
         Class.forName("com.mysql.cj.jdbc.Driver");

         // Step 2: Establish connection
         Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydatabase", "root", "password");

         System.out.println("Connected Successfully!");

         // Close the connection
         con.close();
      } catch (Exception e) {
         e.printStackTrace();
```

```
        }
    }
}
```

Best Practices

- Always close the connection to avoid resource leakage.
- Use connection pooling (like HikariCP, Apache DBCP) for performance in enterprise applications.
- Prefer PreparedStatement for security and efficiency over plain Statement.

Note: Internal database connections in JDBC abstract the complex mechanics of establishing a link between a Java application and a relational database. Understanding this process helps developers write efficient and secure database applications.

9.3 INTERNAL DATABASE CONNECTIONS

Introduction

In web technology, internal database connections refer to the behind-the-scenes process by which a web application connects and communicates with a relational database management system (RDBMS**)** like MySQL, Oracle, or PostgreSQL. These connections are crucial for storing, retrieving, and managing dynamic content in modern web applications.

Key Components Involved

1. **Client**: The end-user interface (browser or mobile app).
2. **Web Server**: Handles HTTP requests and responses.
3. **Application Logic**: Server-side scripts or programs (e.g., Servlets, JSP, PHP, ASP.NET).
4. **Database Driver**: A JDBC driver or equivalent that enables communication with the database.
5. **Database Server**: Stores application data (e.g., MySQL, Oracle DB).

Step-by-Step Internal Workflow

1. Client Sends Request

The process begins when a client (usually a web browser) submits an HTTP request, such as a login form or data entry form.

```
<form action="LoginServlet" method="post">
  <input type="text" name="username">
  <input type="password" name="password">
  <input type="submit" value="Login">
</form>
```

2. Web Server Processes Request

The **Web Server** (e.g., Apache Tomcat) receives the request and forwards it to the appropriate component (Servlet, JSP, etc.).

3. Backend Logic Extracts Input

Server-side code (e.g., a Servlet) extracts input parameters.

```
String uname = request.getParameter("username");
String pwd = request.getParameter("password");
```

4. Load JDBC Driver

Before connecting to the database, the application loads the JDBC driver.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Internally:

- The class loader loads the driver.
- It registers the driver with DriverManager using a static block:
- static {
- DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
- }

5. Establish Database Connection

Use DriverManager or a DataSource to establish a connection.

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/webapp", "root", "password");
```

Internally:

- DriverManager checks the URL and hands the request to the matching driver.
- A socket connection is established to the database.
- User authentication is validated.
- A Connection object is returned to the application.

6. Create SQL Statement

To perform queries, a Statement or PreparedStatement object is created.

```
PreparedStatement pst = con.prepareStatement(
  "SELECT * FROM users WHERE username=? AND password=?");
pst.setString(1, uname);
pst.setString(2, pwd);
```

Internally:

- The SQL is precompiled (in case of PreparedStatement).
- Parameters are safely injected.
- Execution plan may be reused for performance.

7. Execute SQL Query

The query is sent to the database server.

```
ResultSet rs = pst.executeQuery();
```

Internally:

- JDBC translates the query into native database protocol.
- The query is executed on the database engine.
- Results are streamed back to the application as a ResultSet.

8. Process Results

Results from the ResultSet are processed.

```
if(rs.next()) {
  out.println("Login Successful");
} else {
  out.println("Invalid credentials");
}
```

Internally:

- ResultSet manages cursor movement and type conversion.
- Data is converted from SQL types to Java types (e.g., VARCHAR to String).

9. Close Resources

Resources must be closed to release memory and database connections.

```
rs.close();
pst.close();
con.close();
```

Internally:

- JDBC notifies the driver to release sockets, buffers, and connections.
- Connections return to the pool if pooling is used.

Connection Lifecycle Summary

Client Request
↓
Web Server

```
    ↓
Servlet/JSP → JDBC Driver → Database
    ↓
ResultSet ← Response
    ↓
Client
```

Connection Pooling (Advanced Approach)

In enterprise web applications, connection pooling is used to avoid the overhead of frequent connection creation and closing.

Example using DataSource (JNDI in a Servlet):

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MyDB");
Connection con = ds.getConnection();
```

Internally:

- Connections are managed in a pool.
- The same physical connection is reused, improving performance.

Best Practices

Practice	Benefit
Use PreparedStatement	Avoids SQL injection
Always close resources	Prevents memory leaks
Use connection pooling	Enhances performance
Validate inputs	Improves security
Use MVC architecture	Clean code organization

Technologies Involved

Layer	Example
Client	HTML, CSS, JS
Server-Side	Java Servlets, JSP, Spring MVC
Database	MySQL, Oracle
JDBC Driver	MySQL JDBC, Oracle JDBC
Web Server	Tomcat, GlassFish

Note: Internal database connections are a core part of web application architecture. They enable dynamic content generation, user authentication, and business logic execution. Understanding the internal workflow helps in writing efficient, secure, and scalable web applications.

9.4 SUMMARY

JDBC (Java Database Connectivity) is a Java API that enables platform-independent access to relational databases. It allows Java applications to connect to databases, execute SQL queries, and process results efficiently. The core components of JDBC include DriverManager, Connection, Statement, PreparedStatement, CallableStatement, and ResultSet, all working together to manage database communication.

JDBC drivers translate Java calls into database-specific protocols, with four types available: Type 1 (ODBC Bridge), Type 2 (Native API), Type 3 (Network Protocol), and Type 4 (Pure Java/Thin Driver). The typical workflow involves loading the driver, establishing a connection, executing queries, processing results, and closing resources.

In web applications, internal database connections occur when server-side logic (Servlets, JSPs) interacts with the database using JDBC. This involves processing client requests, extracting input, loading drivers, and sending queries via JDBC. Internally, the driver handles socket creation, protocol translation, and authentication.

Best practices include using PreparedStatement to prevent SQL injection, employing connection pooling for performance, and ensuring all resources are closed properly. Web technologies like HTML, JSP, JDBC drivers, and web servers such as Tomcat work together to deliver data-driven applications. Connection pooling through JNDI and DataSource improves resource management. Understanding JDBC's architecture and internal workflow is essential for building efficient and scalable Java-based web systems.

9.5 KEY TERMS

JDBC (Java Database Connectivity),DriverManager, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, SQLException, JDBC,Type 1 Driver, Type 2 Driver, Type 3 Driver, Type 4 Driver, Connection Pooling, DataSource .

9.6 SELF-ASSESSMENT QUESTIONS

1. What is JDBC and why is it used in Java applications?
2. List any three core components of the JDBC API.
3. What are the two main layers of JDBC architecture?
4. Why was JDBC introduced, replacing earlier third-party or platform-specific APIs?
5. What is the role of the DriverManager class in JDBC?
6. Differentiate between Statement and PreparedStatement**.**
7. What is a Type 4 JDBC driver and why is it widely preferred?
8. What steps are involved in establishing a JDBC connection?
9. What does the Connection interface represent in JDBC?
10. How does internal database connection flow work in a web application using JDBC?

9.7 Further Readings

1. Java: The Complete Reference, Twelfth Edition by Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approach by Bart Baesens, Aimee Backiel, and Seppe vanden Broucke. Wiley.
3. Java Programming with Oracle *JDBC* by Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development by David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applications by Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBC by Gregory Brill. Sybex.

Mrs. Appikatla Pushpa Latha

LESSON-10

EXECUTING SQL WITH JDBC

Aim and Objectives:

- To understand the concept and purpose of JDBC as an interface between Java applications and databases.
- To learn how to establish, manage, and close connections between a Java program and a database using JDBC.
- To explain the working and components of JDBC, including DriverManager, Connection, Statement, and ResultSet.
- To explore the internal process of how JDBC drivers communicate with the database for query execution.
- To develop the ability to write and execute SQL statements in Java programs using JDBC connections effectively.

STRUCTURE:

10.1 Introduction to JDBC

What is JDBC?
JDBC (Java Database Connectivity) is a Java API that allows Java applications to interact with various databases. It provides a standard interface for connecting to relational databases, executing SQL queries, and retrieving results.

JDBC is crucial in web technologies because many web applications need to communicate with databases to store, retrieve, and manipulate data dynamically.

Why JDBC?

- **Database Independence**: JDBC abstracts the database-specific details, allowing developers to write database-agnostic code.
- **Integration**: Enables Java-based web applications (Servlets, JSP, Spring, etc.) to interact with backend databases.
- **Standardization**: Offers a uniform API to work with different relational databases like MySQL, Oracle, SQL Server, PostgreSQL, etc.

- **Supports CRUD Operations**: Create, Read, Update, Delete operations on the database.

JDBC Architecture

The JDBC API follows a client-server model and has four main components:

1. **JDBC Drivers**
   These are specific implementations provided by database vendors to communicate between Java applications and the database. Types include:
   - Type 1: JDBC-ODBC Bridge Driver
   - Type 2: Native-API Driver
   - Type 3: Network Protocol Driver
   - Type 4: Thin Driver (Pure Java)

2. **DriverManager**
   Manages the set of JDBC drivers and establishes connections to the database.
3. **Connection Interface**
   Represents a connection session with a specific database.
4. **Statement Interface**
   Used to execute SQL queries (Statement, PreparedStatement, CallableStatement).
5. **ResultSet Interface**
   Represents the result set from SQL query execution, allowing navigation and retrieval of data.

How JDBC Works in Web Technologies?
In web applications, typically the architecture looks like this:

Client (Browser) → Web Server (Servlet/JSP/Spring MVC) → JDBC Layer → Database

Step-by-step workflow:

1. **Load the JDBC Driver**
   This registers the driver with the DriverManager.
   Class.forName("com.mysql.cj.jdbc.Driver");

2. **Establish a Connection**
   Use DriverManager.getConnection() with a database URL, username, and password.
   Connection conn = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/mydb", "root", "password");

3. **Create a Statement**
   Create a statement object to send SQL commands.
   Statement stmt = conn.createStatement();

4. **Execute SQL Queries**
   - For SELECT queries, use executeQuery().
   - For INSERT, UPDATE, DELETE, use executeUpdate().

   ResultSet rs = stmt.executeQuery("SELECT * FROM users");

5. **Process the Results**
   Iterate through the ResultSet to read data.
   ```
   while(rs.next()) {
      System.out.println("User: " + rs.getString("username"));
   }
   ```

6. **Close Resources**
   Close ResultSet, Statement, and Connection to free resources.
   ```
   rs.close();
   stmt.close();
   conn.close();
   ```

JDBC in Web Frameworks

- **Servlets and JSPs**: Use JDBC directly to interact with the database.
- **Spring Framework**: Uses JdbcTemplate for simplified database operations built on top of JDBC.
- **Hibernate / JPA**: ORM frameworks that abstract JDBC, but ultimately rely on JDBC to communicate with the database.

Benefits of Using JDBC in Web Technologies

- Enables dynamic content based on database info.
- Supports scalable, enterprise-level database applications.
- Portable across databases with minimal changes.
- Well-integrated into Java EE and Spring frameworks.

Common Challenges

- **Resource Management**: Forgetting to close connections can lead to memory leaks.
- **SQL Injection**: Must use PreparedStatement to safely pass parameters.
- **Error Handling**: Must handle SQLException properly.
- **Connection Pooling**: Needed for performance in high-traffic web applications.

Sample Code Snippet for JDBC in a Web Application

```
public class UserServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      Connection conn = null;
      PreparedStatement ps = null;
      ResultSet rs = null;

      try {
         Class.forName("com.mysql.cj.jdbc.Driver");
         conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydb", "root", "password");

         String sql = "SELECT username, email FROM users WHERE status =?";
```

```
            ps = conn.prepareStatement(sql);
            ps.setString(1, "active");
            rs = ps.executeQuery();

            while (rs.next()) {
                String username = rs.getString("username");
                String email = rs.getString("email");
                // process user data or add to request attributes
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try { if (rs != null) rs.close(); } catch (SQLException e) { e.printStackTrace(); }
            try { if (ps != null) ps.close(); } catch (SQLException e) { e.printStackTrace(); }
            try { if (conn != null) conn.close(); } catch (SQLException e) { e.printStackTrace(); }
        }
    }
}
```

Note:

- JDBC is a Java API that facilitates communication between Java web applications and databases.
- It provides a standard way to execute SQL commands and process database results.
- It's fundamental for dynamic, database-driven web applications.
- Used in Java Servlets, JSP, Spring, and other Java web frameworks.
- Requires proper resource management, security considerations, and may benefit from connection pooling for high performance.

10.2. Connections

A **JDBC Connection** is the link between your Java application and the database, enabling the execution of SQL statements, retrieval of data, and management of transactions.

Purpose of JDBC Connection

The JDBC Connection object is responsible for:

- Establishing a **session** with the database.
- Sending SQL statements to the DB.
- Handling transactions.
- Providing access to metadata.
- Managing resources (connection closing, etc).

Basic JDBC Connection Workflow

Here's the typical process for using JDBC:

1. Load the JDBC Driver
2. Establish the Connection
3. Create SQL Statements
4. Execute SQL Queries
5. Process Results
6. Close the Connection

JDBC Connection in Detail

1. Loading the JDBC Driver

This step loads the JDBC driver class provided by the database vendor.

```
Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL
```

As of JDBC 4.0 (Java 6+), this step is often optional if the driver is available on the classpath.

2. Establishing a Connection

Use the DriverManager class to create a connection to the database.

```
Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
```

Connection URL Format

jdbc:subprotocol:subname

Example (for MySQL):

jdbc:mysql://localhost:3306/mydatabase

Component	Description
jdbc	Protocol
mysql	Subprotocol (DB type)
localhost	Hostname of DB server
3306	Port number
mydatabase	Name of the database

3. Common JDBC Driver Connection URLs

Database	JDBC Driver Class	Example URL
MySQL	com.mysql.cj.jdbc.Driver	jdbc:mysql://localhost:3306/mydb
PostgreSQL	org.postgresql.Driver	jdbc:postgresql://localhost:5432/mydb
Oracle	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@localhost:1521:xe
SQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:sqlserver://localhost:1433;databaseName=mydb

4. Example Code: Establishing JDBC Connection

```
import java.sql.*;

public class JdbcConnectionExample {
   public static void main(String[] args) {
      String url = "jdbc:mysql://localhost:3306/mydb";
      String user = "root";
      String password = "admin";

      try (Connection conn = DriverManager.getConnection(url, user, password)) {
         System.out.println("Connected to database successfully!");
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
}
```

JDBC Connection Interface Methods

The java.sql.Connection interface provides many useful methods:

Method	Purpose
createStatement()	Creates a basic SQL statement
prepareStatement(String sql)	Creates a precompiled SQL statement
setAutoCommit(boolean)	Enables/disables auto-commit mode
commit()	Commits transaction manually
rollback()	Rolls back transaction manually
close()	Closes the connection
isClosed()	Checks if connection is closed
getMetaData()	Returns database metadata

Transaction Management

By default, JDBC is in auto-commit mode.

Auto-Commit Mode (Default)

Each SQL statement is committed immediately after execution.

```
conn.setAutoCommit(true);  // default
```

Manual Commit Mode

You can turn off auto-commit and commit manually:

```
conn.setAutoCommit(false);

try {
   // Execute SQL statements
   conn.commit();  // Commit if all succeed
} catch (SQLException e) {
   conn.rollback();  // Rollback if any fails
}
```

Handling Exceptions and Closing Connection

It's important to close connections to avoid memory leaks.

Try-With-Resources (Recommended)

```
try (Connection conn = DriverManager.getConnection(url, user, password)) {
   // Work with connection
} catch (SQLException e) {
   e.printStackTrace();
}
```

This automatically closes the connection at the end of the block.

JDBC Connection in Web Applications

In Java web apps (Servlets, JSPs, Spring), JDBC is commonly used for database access.

Example in a Servlet:

```
public class UserServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response) {
      try (Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root", "pass")) {
         PreparedStatement ps = conn.prepareStatement("SELECT * FROM users");
         ResultSet rs = ps.executeQuery();
         while (rs.next()) {
```

```
            // process data
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

JDBC Connection Pooling (Advanced)

Creating a new connection for every request is expensive. Connection pooling improves performance by reusing existing connections.

Popular Connection Pooling Libraries:

Library	Features
HikariCP	Fast and lightweight
Apache DBCP	Easy integration
C3P0	Automatic testing and recovery

Spring Boot uses HikariCP by default.

Security Tips for JDBC Connections

- Avoid hardcoding DB credentials.
- Use config files or environment variables.
- Always close your connection objects.
- Use PreparedStatement to prevent SQL Injection.

JDBC Connection Best Practices

Practice	Why It's Important
Use try-with-resources	Ensures connections are closed properly
Use connection pooling	Improves performance for web applications
Close ResultSet and Statement	Frees resources and avoids memory leaks
Avoid hardcoded credentials	Improves security
Use transactions where needed	Maintains data integrity

Table

Feature	Description
Connection	Interface to connect Java app to DB
Created By	DriverManager.getConnection()
Needs Driver?	Yes, specific to DB (e.g., MySQL, Oracle)
Connection URL	Follows format: jdbc:subprotocol:subname
Auto-Commit	Default is true, can be set to manual
Use in Web Apps	JDBC connects backend to DB in Servlets, JSPs, Spring etc.

Connection Pooling Reuses DB connections, improves performance

[

10.3. Internal Database Connections

1. Internal Database Connection

An **Internal Database Connection** refers to the behind-the-scenes process through which a Java application communicates with a database via JDBC.

- It is not visible to the user but is essential for executing SQL commands, retrieving results, and maintaining sessions**.**
- JDBC abstracts these internal processes, so developers focus on Java code rather than database protocols.

2. Components Involved in Internal Connections

When a JDBC program connects to a database, several internal components work together:

Component	Role
DriverManager	Maintains a registry of JDBC drivers and selects the appropriate driver for a database URL.
JDBC Driver	Converts Java method calls into database-specific network protocol commands.
Connection Object	Represents a live session with the database, including metadata like database version, session info, and active transactions.
Statement / PreparedStatement / CallableStatement	Sends SQL queries to the database and processes results.
ResultSet Object	Stores the query results and provides methods to retrieve data row by row.

3. How Internal Database Connections Work

Here’s a step-by-step explanation of the internal workflow when a Java program connects to a database:

Step 1: Driver Registration

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- The JDBC driver class is loaded.
- The driver registers itself with DriverManager, making it available for connection requests.

Step 2: Requesting a Connection

```
Connection con = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/world", "root", "12345");
```

- The DriverManager checks its list of registered drivers.
- It selects a driver that supports the provided database URL**.**

Step 3: Authentication and Session Creation

- The driver sends the username and password to the database server.
- The database validates credentials.
- If valid, a **session** is created, and a Connection object is returned to the Java program.

Step 4: SQL Query Execution

- SQL commands executed via Statement or PreparedStatement are internally converted into database-specific requests.
- The driver handles protocol translation, sending the commands over TCP/IP to the database server.

Step 5: Result Processing

- The database executes the query and returns results.
- The driver converts the database-specific response into a ResultSet object.
- The Java application iterates over the ResultSet to retrieve data.

Step 6: Closing the Connection

```
con.close();
```

- Closing the connection informs the driver and database to release resources.
- Internally, the session is terminated, and sockets are closed.

4. Key Features of Internal Database Connections

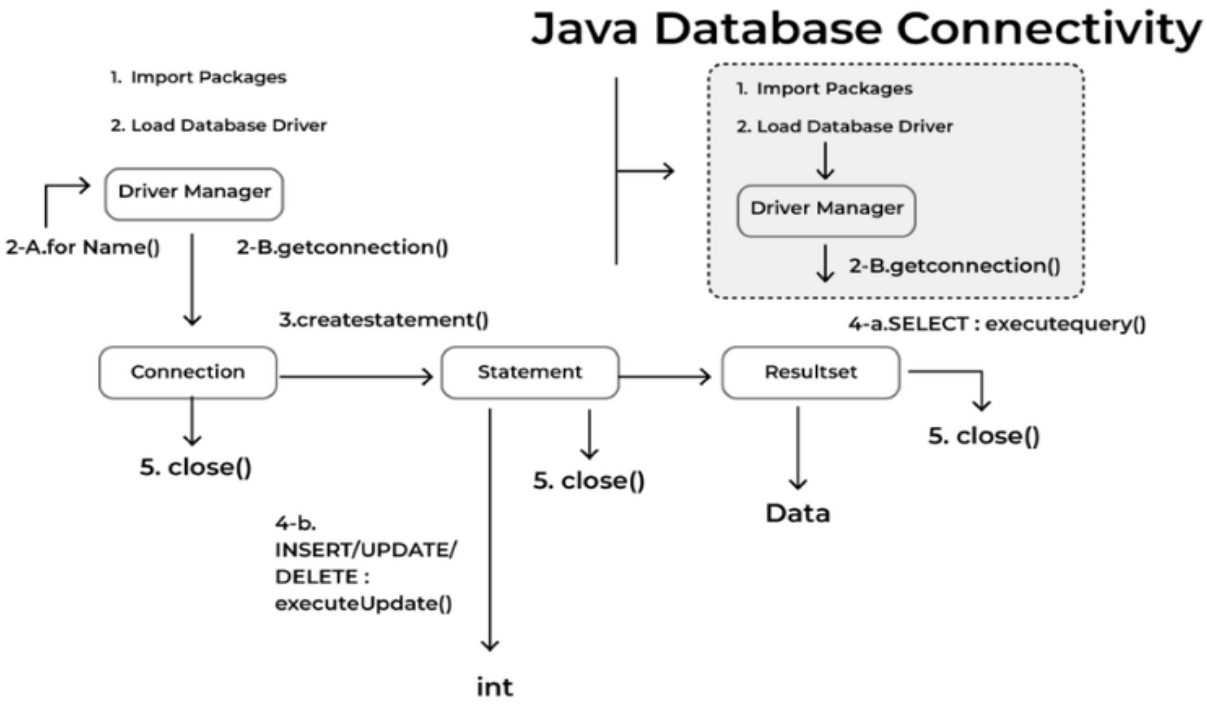
1. **Automatic Protocol Handling:** Converts Java calls into database-specific commands.
2. **Session Management:** Maintains database session metadata such as user, privileges, and active transactions.
3. **Result Handling:** Converts raw database results into ResultSet objects for Java applications.

4. **Security:** Handles authentication and prevents unauthorized access at the session level.
5. **Transparency:** Developers don't need to know the underlying network or protocol details.

5. Advantages of Understanding Internal Connections

- **Optimized Performance:** Helps in implementing connection pooling to reuse connections efficiently.
- **Better Debugging:** Understanding internal flow helps troubleshoot connection failures or slow queries.
- **Security Awareness:** Explains how credentials and queries travel from Java to the database.
- **Resource Management:** Encourages proper closing of connections, statements, and result sets.
- **Scalability:** Enables designing multi-tier applications with minimal connection overhead.

6. Diagram: Internal Database Connection Flow



7. Best Practices for Internal Database Connections

1. Always close connections, statements, and result sets in finally blocks or use try-with-resources**.**
2. Use PreparedStatement to improve performance and prevent SQL injection.
3. Use connection pooling in enterprise applications to reduce connection overhead.
4. Minimize open connections to reduce memory and server load.
5. Handle SQLException properly to catch connection or query failures.

Note:

- Internal database connections are the hidden processes that allow Java applications to communicate with databases via JDBC.
- JDBC DriverManager + Driver + Connection + Statement + ResultSet work together internally.
- Understanding internal connections helps with performance, security, and proper resource management**.**
- While transparent to developers, this internal workflow is critical for building robust, scalable, and secure database applications**.**

10.4. STATEMENTS

In Java, the **Statement** interface of JDBC (Java Database Connectivity) is used to create and execute SQL queries within Java applications.JDBC provides three types of statements to interact with the database:

- **Statement** -> For executing static SQL queries.
- **PreparedStatement** -> For executing parameterized queries.
- **CallableStatement** -> For executing stored procedures.

1. Statement

A Statement object is used for general-purpose access to databases and is useful for executing static SQL statements at runtime.

Syntax:

Statement statement = connection.createStatement();

Execution Methods

- **execute(String sql):** Executes any SQL (SELECT, INSERT, UPDATE, DELETE). Returns true if a ResultSet is returned.
- **executeUpdate(String sql):** Executes DML (INSERT, UPDATE, DELETE). Returns number of rows affected.
- **executeQuery(String sql):** Executes SELECT queries. Returns a ResultSet.

Example:

```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
      try {
         // Step 1: Load the MySQL JDBC Driver
         Class.forName("com.mysql.cj.jdbc.Driver");

         // Step 2: Establish a connection to the database
         Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/world", "root", "12345");

         // Step 3: Create a Statement object to send SQL queries
         Statement stmt = con.createStatement();

         // Step 4: Define and execute an SQL SELECT query
         String query = "SELECT * FROM people";
         ResultSet rs = stmt.executeQuery(query);
```

```
            // Step 5: Process the ResultSet
            while (rs.next()) {
                String name = rs.getString("name");
                int age = rs.getInt("age");
                System.out.println("Name: " + name + ", Age: " + age);
            }

            // Step 6: Close all resources
            rs.close();
            stmt.close();
            con.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Output: Name and age are as shown for random inputs.

10.5 SUMMARY

JDBC (Java Database Connectivity) is a Java API that allows Java applications to interact with relational databases in a standardized way. It enables executing SQL queries, retrieving results, and performing CRUD operations, making it crucial for dynamic, database-driven web applications. JDBC provides a uniform interface for different databases such as MySQL, Oracle, PostgreSQL, and SQL Server, ensuring database independence.

The JDBC architecture follows a client-server model and consists of components like JDBC drivers, DriverManager, Connection, Statement, and ResultSet. JDBC drivers handle communication between Java programs and databases, converting Java calls into database-specific protocol commands. The Connection object establishes a session with the database, manages transactions, and provides metadata access. Internal database connections represent the hidden workflow of driver registration, authentication, session creation, SQL execution,

result processing, and closing resources. Developers interact with Statement objects to execute SQL queries, with three types available: Statement for static queries, PreparedStatement for parameterized queries, and CallableStatement for stored procedures. Using PreparedStatement improves performance and prevents SQL injection.

JDBC in web technologies is widely used in Servlets, JSP, Spring, and other Java frameworks, allowing backend databases to provide dynamic content. Proper resource management, such as closing connections, statements, and result sets, is critical to avoid memory leaks. Connection pooling is recommended for high-performance web applications. Transactions can be managed automatically or manually using commit and rollback methods. Understanding internal connections helps with debugging, security, and optimizing database interactions. Overall, JDBC provides a robust, scalable, and secure way to integrate Java applications with databases.

10.6 KEY TERMS

JDBC (Java Database Connectivity) **,** DriverManager **,**Connection , JDBC Driver , Connection Pooling , SQL Injection,Transaction Management, Internal Database Connection, Statement.

10.7 SELF-ASSESSMENT QUESTIONS

1. Define JDBC and explain its importance in Java web applications.
2. Explain the architecture of JDBC and describe the role of each component.
3. What are the different types of JDBC drivers? Explain with examples.
4. Describe the process of establishing a JDBC connection, including the connection URL format.
5. Explain the difference between Statement?
6. What is an internal database connection in JDBC? Describe the workflow from driver registration to closing the connection.
7. Explain how transaction management works in JDBC and the difference between auto-commit and manual commit modes.
8. What are the best practices for using JDBC in web applications? Explain with reasons.

10.8 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition by Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approach by Bart Baesens, Aimee Backiel, and Seppe vanden Broucke. Wiley.
3. Java Programming with Oracle JDBC by Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development by David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applications by Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBC by Gregory Brill. Sybex.

LESSON- 11

ADVANCED JDBC: PREPARED AND CALLABLE STATEMENTS

Aim and Objectives:

- Understand the purpose of Statement objects in executing SQL queries from Java programs.
- Learn how ResultSet stores and allows navigation through data retrieved from a database.
- Explore PreparedStatement for executing parameterized queries to prevent SQL injection and improve performance.
- Understand CallableStatement for invoking stored procedures in a database from Java applications.
- Develop the ability to perform database operations efficiently using different types of JDBC statements.

Structure:

11.1. STATEMENTS

1. Introduction

In JDBC (Java Database Connectivity), a Statement is an interface used to execute SQL queries against a database. It allows Java programs to interact with a database, retrieve data, and update it. Statements are best suited for static SQL queries that do not change often, unlike PreparedStatement, which is used for dynamic queries with parameters.

Types of Statements

1. **Statement** – Used for simple SQL queries without parameters.
2. **PreparedStatement** – Used for parameterized queries (dynamic queries).
3. **CallableStatement** – Used to execute stored procedures in a database.

2. Key Features of Statement

- Executes SQL queries like SELECT, INSERT, UPDATE, and DELETE.
- Returns ResultSet for queries that retrieve data.
- Simple to use for basic database operations.

- Not secure for dynamic user inputs (prone to SQL injection).

3. Creating a Statement

To create a Statement object, you need:

1. Load the JDBC driver.
2. Establish a connection to the database.
3. Create a Statement object.
4. Execute SQL queries using the statement.

4. Example Program Using Statement

Database Setup

Assume a database SchoolDB with table Students:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

Java Program: Statement Example

```
importjava.sql.*;

public class StatementExample
 {
public static void main(String[] args)
{
Try
 {
        // 1. Load JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish connection
        Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");

        // 3. Create Statement object
        Statement stmt = con.createStatement();

        // 4. Execute SQL query
ResultSetrs = stmt.executeQuery("SELECT * FROM Students");

        // 5. Process ResultSet
System.out.println("ID\tName\tAge\tGrade");
while(rs.next()) {
```

```
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
        }

        // 6. Close connections
rs.close();
stmt.close();
con.close();

      } catch(Exception e) {
e.printStackTrace();
      }
   }
}
```

5. Input / Output

Input

- The program does not take dynamic input; it directly queries the Students table.

Output (Console)

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

6. Executing Update Queries Using Statement

You can also execute INSERT, UPDATE, or DELETE queries using Statement.

```
// Insert a new student
introwsInserted = stmt.executeUpdate(
   "INSERT INTO Students (ID, Name, Age, Grade) VALUES (4, 'David', 13, 7)");
System.out.println(rowsInserted + " row(s) inserted.");
```

Output:

1 row(s) inserted.

7. Note:

- executeQuery() is used for SELECT queries and returns a ResultSet.
- executeUpdate() is used for INSERT, UPDATE, DELETE queries and returns the number of affected rows.
- For queries with dynamic user input, prefer PreparedStatement to avoid SQL injection.

- Statement is simple but less secure and less efficient for repeated queries.

11.2 RESULTS SETS

1. Introduction

In JDBC, a **ResultSet** is an object that holds the data retrieved from a database after executing a SELECT query using a Statement or PreparedStatement. It acts like a table in memory, allowing you to navigate through rows of data and fetch column values.

Key Points

- Represents tabular data from a database query.
- Cursor-based: can move forward, backward (depending on type), or jump to a specific row.
- Read-only or updatable (depending on how you create it).
- Often used with while(rs.next()) to iterate through rows.

2. Types of ResultSet

Type	Description
TYPE_FORWARD_ONLY	Default. Cursor moves only forward.
TYPE_SCROLL_INSENSITIVE	Cursor can move forward/backward. Changes in DB after query execution are not visible.
TYPE_SCROLL_SENSITIVE	Cursor can move forward/backward. Reflects changes in DB after query execution.

3. Creating a ResultSet

1. Create a Statement object.
2. Execute a SELECT query using executeQuery().
3. Store the returned data in a ResultSet.
4. Iterate through the ResultSet to process data.

4. Example Program Using ResultSet

Database Setup

Assume a database SchoolDB with table Students:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

Java Program: ResultSet Example

```
importjava.sql.*;

public class ResultSetExample
{
public static void main(String[] args)
 {
try
{
        // 1. Load JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish connection
        Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");

        // 3. Create Statement object
        Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);

        // 4. Execute SELECT query
ResultSetrs = stmt.executeQuery("SELECT * FROM Students");

        // 5. Process ResultSet
System.out.println("ID\tName\tAge\tGrade");
while(rs.next())
{
// iterate forward
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
        }

        // 6. Moving cursor backwards (if scrollable)
System.out.println("\nLast record:");
if(rs.last())
 {
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
        }

        // 7. Close connections
rs.close();
stmt.close();
con.close();
```

```
      } catch(Exception e)
{
e.printStackTrace();
      }
   }
}
```

5. Input / Output

Input

- The program directly queries the Students table. No dynamic input is required.

Output (Console)

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

Last record:

3	Charlie	14	8

6. Updating Data Using ResultSet (Optional)

Some ResultSet objects can be updatable. Example:

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

ResultSetrs = stmt.executeQuery("SELECT * FROM Students WHERE ID=2");
if(rs.next())
{
rs.updateInt("Age", 16); // update age
rs.updateRow();          // commit changes
}
```

This directly updates the database without executing a separate UPDATE query.

7. Note:

- rs.getInt("columnName") or rs.getString("columnName") fetches column data.
- Always close ResultSet and Statement to free resources.
- For large datasets, consider using pagination because ResultSet loads data into memory.
- Scrollable and updatable ResultSets are more flexible but slightly slower.

11.3 Prepared Statements

1. Introduction

A **PreparedStatement** is a feature of JDBC used to execute parameterized SQL queries. Unlike a regular Statement, which executes raw SQL strings, PreparedStatement allows placeholders (?) for values, which are supplied at runtime.

Advantages

1. **Prevents SQL Injection** – User input is treated as data, not code.
2. **Improves Performance** – The SQL query is precompiled by the database.
3. **Reusability** – The same query can be executed multiple times with different inputs.
4. **Type Safety** – You can explicitly set the data type of parameters.

2. Syntax

PreparedStatementpstmt = con.prepareStatement("SQL query with ?");

- ? is a placeholder for a value.
- Values are set using methods like:
  - setInt(index, value)
  - setString(index, value)
  - setDouble(index, value)

3. Example Program Using PreparedStatement

Database Setup

Assume a database SchoolDB with table Students:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9

Java Program: Insert Using PreparedStatement

```
importjava.sql.*;

public class PreparedStatementExample
 {
public static void main(String[] args)
{
Try
 {
        // 1. Load JDBC Driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish Connection
```

```
        Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");

        // 3. Prepare SQL Query with placeholders
        String sql = "INSERT INTO Students (ID, Name, Age, Grade) VALUES (?, ?, ?, ?)";
PreparedStatementpstmt = con.prepareStatement(sql);

        // 4. Set parameter values
pstmt.setInt(1, 3);          // ID
pstmt.setString(2, "Charlie"); // Name
pstmt.setInt(3, 14);          // Age
pstmt.setInt(4, 8);          // Grade

        // 5. Execute query
int rows = pstmt.executeUpdate();
System.out.println(rows + " row(s) inserted.");

        // 6. Close connection
pstmt.close();
con.close();

      }
catch(Exception e)
 {
e.printStackTrace();
      }
   }
}
```

4. Input / Output

Input

- Directly set in the program using pstmt.setXXX() methods.

Output

1 row(s) inserted.

- After execution, the table Students will have:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

5. Example Program: Select Using PreparedStatement

```
String sql = "SELECT * FROM Students WHERE Age = ?";
PreparedStatementpstmt = con.prepareStatement(sql);
pstmt.setInt(1, 14); // parameter for age
ResultSetrs = pstmt.executeQuery();

while(rs.next()) {
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
}
```

Output

```
ID   Name     Age   Grade
1    Alice    14    8
3    Charlie  14    8
```

6. Note:

- PreparedStatement is preferred over Statement for dynamic queries.
- Supports batchprocessing for multiple inserts efficiently:

```
pstmt.setInt(1, 4);
pstmt.setString(2, "David");
pstmt.setInt(3, 15);
pstmt.setInt(4, 9);
pstmt.addBatch(); // add to batch

int[] result = pstmt.executeBatch(); // execute all at once
```

- Always close PreparedStatement and Connection to free resources.

11.4 CALLABLE STATEMENTS

1. Introduction

A **CallableStatement** is used in JDBC to execute storedprocedures in a database. Stored procedures are precompiled SQL programs stored in the database.

Advantages

1. **Encapsulation** – Database logic is centralized in procedures.
2. **Performance** – Stored procedures are precompiled.
3. **Security** – Reduces SQL injection risk.
4. **Ease of Maintenance** – Changes in logic don’t require Java code changes.

2. Syntax

CallableStatementcstmt = con.prepareCall("{call procedure_name(?, ?)}");

- {callprocedure_name(?, ?)} – ? are placeholders for input/output parameters.
- Parameters can be:
  - **IN** – Input to the procedure
  - **OUT** – Output from the procedure
  - **INOUT** – Input and output

Set parameters with:

- cstmt.setInt(index, value)
- cstmt.setString(index, value)

Register output parameters with:

- cstmt.registerOutParameter(index, type)

3. Example: Stored Procedure in MySQL

Create a stored procedure in MySQL SchoolDB:

```
DELIMITER //

CREATE PROCEDURE GetStudentByID(IN sid INT, OUT sname VARCHAR(50), OUT
sage INT)
BEGIN
   SELECT Name, Age INTO sname, sage FROM Students WHERE ID = sid;
END //

DELIMITER ;
```

4. Java Program Using CallableStatement

```
importjava.sql.*;

public class CallableStatementExample {
public static void main(String[] args) {
try {
        // 1. Load JDBC Driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish Connection
        Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");

        // 3. Prepare CallableStatement
CallableStatementcstmt = con.prepareCall("{call GetStudentByID(?, ?, ?)}");
```

```
        // 4. Set input parameter
cstmt.setInt(1, 1); // ID = 1

        // 5. Register output parameters
cstmt.registerOutParameter(2, Types.VARCHAR); // Name
cstmt.registerOutParameter(3, Types.INTEGER); // Age

        // 6. Execute stored procedure
cstmt.execute();

        // 7. Retrieve output
        String name = cstmt.getString(2);
int age = cstmt.getInt(3);

System.out.println("Student Name: " + name);
System.out.println("Student Age: " + age);

        // 8. Close connection
cstmt.close();
con.close();

      } catch(Exception e) {
e.printStackTrace();
      }
   }
}
```

5. Input / Output

Input

- ID = 1 (passed as input to stored procedure)

Output

Student Name: Alice
Student Age: 14

6. Note:

- CallableStatement can handle IN, OUT, and INOUT parameters.
- It is mainly used when complex database operations are handled inside stored procedures.
- Example of using INOUT parameter:

```
CREATE PROCEDURE IncreaseAge(INOUT sid INT, IN increment INT)
BEGIN
   UPDATE Students SET Age = Age + increment WHERE ID = sid;
   SELECT Age INTO sid FROM Students WHERE ID = sid;
```

END;

- In Java, register sid as INOUT:

cstmt.registerOutParameter(1, Types.INTEGER); // INOUT parameter

Difference between CallableStatement and PreparedStatement :

CallableStatement	PreparedStatement
It is used when the stored procedures are to be executed.	It is used when SQL query is to be executed multiple times.
You can pass 3 types of parameter IN, OUT, INOUT.	You can pass any type of parameters at runtime.
Used to execute functions.	Used for the queries which are to be executed multiple times.
Performance is very high.	Performance is better than Statement.
Used to call the stored procedures.	Used to execute dynamic SQL queries.
It extends PreparedStatement interface.	It extends Statement Interface.
No protocol is used for communication.	Protocol is used for communication.

Overview of the Statement PreparedStatement CallableStatement :

Feature	**Statement**	**PreparedStatement**
Purpose	Executes static SQL queries	Executes parameterized queries
SQL injection protection	No	Yes
Performance	Normal (re-parsed every time)	Faster (precompiled)
Used for	Simple queries	Dynamic queries with parameters

Table for the PreparedStatement CallableStatement ResultSet :

Feature	PreparedStatement	CallableStatement	ResultSet
Main Use	Execute parameterized SQL queries	Execute stored procedures	Hold query results
SQL type	DML (SELECT, INSERT, UPDATE, DELETE)	Stored procedure/function	Output of a query
Parameter Handling	Uses ? placeholders	Supports IN, OUT, INOUT parameters	Access data via column names/index
Compilation	Precompiled (faster)	Precompiled (DB-side procedure)	Not compiled; returned as data
SQL Injection Safe	Yes	Yes	N/A
Return Type	ResultSet or update count	ResultSet or output params	Data from query
Example	SELECT * FROM emp WHERE dept=?	{call getEmployeeByDept(?)}	Iterating over query results

11.5 SUMMARY

In JDBC, several key objects are used to interact with databases, each serving a distinct purpose. The Statement object is used to execute simple SQL queries such as SELECT, INSERT, UPDATE, and DELETE. However, since it directly embeds user inputs into SQL strings, it is vulnerable to SQL injection attacks, making it suitable only for static or simple queries. The ResultSet object stores the data retrieved from queries and provides cursor-based navigation to move through rows of results. Depending on its type, it can be forward-only or scrollable, and even updatable, allowing direct modification of data within the result set.

For more secure and efficient database operations, the PreparedStatement object is preferred. It allows parameterized queries, preventing SQL injection and improving performance by precompiling the SQL statement. PreparedStatements are reusable, making them ideal for executing similar queries multiple times with different parameters. They also support batch processing for executing multiple SQL commands efficiently. When working with stored procedures, the CallableStatement object is used. It allows the execution of precompiled database procedures that can accept IN, OUT, and INOUT parameters, offering better performance, encapsulation, and security for complex operations.

Proper resource management is crucial when using these JDBC objects—closing the ResultSet, Statement, and Connection objects ensures system efficiency and prevents memory leaks. Overall, using the right JDBC object based on the operation type—Statement for simple queries, PreparedStatement for dynamic queries, and CallableStatement for stored procedures—enables developers to build robust, secure, and maintainable database applications.

11.6 KEY TERMS

Statement, PreparedStatement, CallableStatement, Result Set, SQL, Database, Connection, Stored Procedure, Parameter,Cursor,executeQuery,executeUpdate.

11.7 SELF-ASSESSMENT QUESTIONS

1. What is the purpose of a Statement in JDBC?
2. How does PreparedStatement differ from Statement?
3. What is a CallableStatement used for?
4. What does a ResultSet represent in JDBC?
5. Name the three types of ResultSet cursors.
6. Which method is used to execute a SELECT query: execute Query() or execute Update()?
7. How do you set values in a PreparedStatement?
8. What is the advantage of using Prepared Statement over Statement?
9. What types of parameters can a Callable Statement handle?
10. Why is it important to close Result Set, Statement, and Connection objects?

11.8 Further Readings

1. Java: The Complete Reference, Twelfth Edition *by Herbert Schildt. McGraw-Hill Education.*
2. Beginning Java Programming: The Object-Oriented Approach *by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.*
3. Java Programming with Oracle JDBC *by Donald Bales. O'Reilly Media.*
4. Java EE 8 Application Development *by David R. Heffelfinger. Packt Publishing.*
5. Professional Java for Web Applications *by Nicholas S. Williams. Wrox/Wiley Publishing.*
6. Java 2: Developer's Guide to Web Applications with JDBC *by Gregory Brill. Sybex.*

Mrs. Appikatla PushpaLatha

LESSON- 12

NETWORK PROGRAMMING AND REMOTE METHOD INVOCATION (RMI)

Aim and Objectives:

- Understand the importance of networked Java for building distributed applications.
- Learn basic network concepts, including protocols, IP addresses, and ports.
- Gain the ability to look up Internet addresses and differentiate between URLs and URIs**.**
- Explore UDP datagrams and sockets for sending and receiving data over a network.
- Understand Remote Method Invocation (RMI) to enable communication between Java objects across different machines.

STRUCTURE:

12.1 Network Programming

- It is about writing programs that can send and receive data over a network (like the internet or a local network).
- **Why needed:** For client-server applications, chatting apps, online games, or distributed systems.

- **Key concepts:**
  - **IP Address & Ports:** Identify devices and communication endpoints.
  - **Sockets:** Points where programs connect and exchange data.
  - **Protocols:** TCP (reliable) and UDP (fast, less reliable).
  - **URLs and URIs:** Ways to locate and access resources on the web.

12.2 Why networked Java

Networked Java refers to Java programs that communicate over a network, like the Internet or a local network. Java is widely used for network programming because it provides built-in support for networking via packages like java.net and java.rmi.

Reasons to use Networked Java:

1. **Platform Independence:** Java runs on any machine with a JVM, making networked applications portable.
2. **Built-in Networking API:** Java provides classes like Socket, ServerSocket, DatagramSocket, URL, etc., for easy network communication.
3. **Supports Client-Server Architecture:** You can easily build applications where a client requests data from a server.
4. **RMI (Remote Method Invocation):** Java allows calling methods on remote objects, making distributed systems easier to develop.
5. **Secure Networking:** Java provides SSL and security features for safe network communication.

Basic Java Network Programming Example

1. TCP Client-Server Program

Server Program (TCP)

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

public class TCPServer {
public static void main(String[] args) throws IOException {
ServerSocket server = new ServerSocket(5000);
System.out.println("Server is running and waiting for a client...");

      Socket client = server.accept();
BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
PrintWriter out = new PrintWriter(client.getOutputStream(), true);

      String message = in.readLine();
System.out.println("Client says: " + message);
out.println("Hello Client! Message received.");

client.close();
server.close();
   }
}
```

Client Program (TCP)

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

public class TCPClient {
public static void main(String[] args) throws IOException {
      Socket socket = new Socket("localhost", 5000);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
```

```
out.println("Hello Server! This is Client.");
        String response = in.readLine();
System.out.println("Server says: " + response);

socket.close();
    }
}
```

Input / Output Example

Client Input:

Hello Server! This is Client.

Server Output:

Server is running and waiting for a client...
Client says: Hello Server! This is Client.

Client Output:

Server says: Hello Client! Message received.

2. UDP Datagram Example

Server (UDP)

```
import java.net.*;

public class UDPServer {
public static void main(String[] args) throws Exception {
DatagramSocketserverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];
byte[] sendData;

System.out.println("UDP Server is running...");

DatagramPacketreceivePacket = new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);
        String message = new String(receivePacket.getData(), 0, receivePacket.getLength());
System.out.println("Client says: " + message);

InetAddressclientAddress = receivePacket.getAddress();
intclientPort = receivePacket.getPort();
        String reply = "Message received via UDP!";
sendData = reply.getBytes();

DatagramPacketsendPacket = new DatagramPacket(sendData, sendData.length,
clientAddress, clientPort);
serverSocket.send(sendPacket);
```

```
serverSocket.close();
   }
}
```

Client (UDP)

```
import java.net.*;

public class UDPClient
 {
public static void main(String[] args) throws Exception
{
DatagramSocketclientSocket = new DatagramSocket();
InetAddressIPAddress = InetAddress.getByName("localhost");

byte[] sendData = "Hello UDP Server!".getBytes();
byte[] receiveData = new byte[1024];

DatagramPacketsendPacket = new DatagramPacket(sendData, sendData.length, IPAddress,
9876);
clientSocket.send(sendPacket);

DatagramPacketreceivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
      String response = new String(receivePacket.getData(), 0, receivePacket.getLength());
System.out.println("Server says: " + response);

clientSocket.close();
   }
}
```

Input / Output Example

Client Input:

Hello UDP Server!

Server Output:

UDP Server is running...
Client says: Hello UDP Server!

Client Output:

Server says: Message received via UDP!

Note:

- Networked Java allows communication between programs over TCP or UDP.
- RMI allows calling methods on remote objects as if they were local.

- Java's built-in networking libraries make creating client-server or distributed applications easy and platform-independent.

12.3 Basic Network Concepts

Network programming involves communication between computers over a network. Java provides built-in support via the java.net package.

Key Concepts:

1. **IP Address:**
   - Unique address assigned to each device on a network.
   - Example: 192.168.1.1 (IPv4) or 2001:0db8:85a3::8a2e:0370:7334 (IPv6).
2. **Port Number:**
   - Used to identify a specific process/application on a device.
   - Range: 0–65535 (Ports <1024 are reserved).
3. **Socket:**
   - Endpoint for communication between two machines.
   - Socket class for clients, ServerSocket class for servers.
4. **TCP (Transmission Control Protocol):**
   - Connection-oriented protocol (reliable).
5. **UDP (User Datagram Protocol):**
   - Connectionless protocol (faster but unreliable).
6. **URL and URI:**
   - URL (Uniform Resource Locator) – identifies a resource on the internet.
   - URI (Uniform Resource Identifier) – more general, can be a URL or just a name.

Java Example Programs

1. Finding IP Address

```
import java.net.*;

public class IPAddressExample {
public static void main(String[] args) throws Exception {
InetAddress address = InetAddress.getByName("www.google.com");
System.out.println("Host Name: " + address.getHostName());
System.out.println("IP Address: " + address.getHostAddress());
   }
}
```

Output Example:
Host Name: www.google.com
IP Address: 142.250.72.196

2. Simple TCP Client-Server

Server (TCP)

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

```
import java.net.*;

public class SimpleTCPServer {
public static void main(String[] args) throws IOException {
ServerSocket server = new ServerSocket(6000);
System.out.println("Server waiting for connection...");
      Socket client = server.accept();

BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
PrintWriter out = new PrintWriter(client.getOutputStream(), true);

      String message = in.readLine();
System.out.println("Client says: " + message);
out.println("Message received!");

client.close();
server.close();
   }
}
```

Client (TCP)

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

public class SimpleTCPClient {
public static void main(String[] args) throws IOException {
      Socket socket = new Socket("localhost", 6000);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

out.println("Hello Server!");
System.out.println("Server says: " + in.readLine());

socket.close();
   }
}
```

Input / Output Example

Client Input:

Hello Server!

Server Output:

Server waiting for connection...
Client says: Hello Server!

Client Output:

Server says: Message received!

3. URL Example

```
import java.net.*;

public class URLExample {
public static void main(String[] args) throws Exception {
      URL url = new URL("https://www.example.com");
System.out.println("Protocol: " + url.getProtocol());
System.out.println("Host: " + url.getHost());
System.out.println("File: " + url.getFile());
System.out.println("Port: " + url.getPort());
   }
}
```

Output Example:

Protocol: https
Host: www.example.com
File: /
Port: -1

(Port -1 means default port 443 for HTTPS)

Note:

- **IP & Port** identify a machine and application.
- **Sockets** enable communication (TCP for reliability, UDP for speed).
- **URLs/URIs** locate resources on the internet.
- Java's java.net makes network programming straightforward.

12.4 Looking up Internet Addresses

Looking up internet addresses in Java involves retrieving IP addressesandhost names for computers or domain names. This is done using the InetAddress class from the java.net package.

Key Concepts:

1. **InetAddress Class:**
   - Represents an IP address (IPv4 or IPv6).
   - Can be used to get host name and host address.
2. **Methods of InetAddress:**
   - getByName(String host) – Returns the IP address of a given host.
   - getHostName() – Returns the host name.
   - getHostAddress() – Returns the IP address as a string.
   - getAllByName(String host) – Returns all IP addresses associated with a host.
3. **DNS Lookup:**
   - Java can resolve host names to IP addresses (forward lookup).
   - Can also perform reverse lookup (IP → Host name).

Java Examples

1. Lookup IP Address of a Domain

```
import java.net.*;

public class InetAddressExample {
public static void main(String[] args) throws Exception {
InetAddress address = InetAddress.getByName("www.google.com");
System.out.println("Host Name: " + address.getHostName());
System.out.println("IP Address: " + address.getHostAddress());
   }
}
```

Output Example:

Host Name: www.google.com
IP Address: 142.250.72.196

(IP may vary depending on DNS and location.)

2. Get Local Host Information

```
import java.net.*;

public class LocalHostExample {
public static void main(String[] args) throws Exception {
InetAddress local = InetAddress.getLocalHost();
System.out.println("Local Host Name: " + local.getHostName());
System.out.println("Local IP Address: " + local.getHostAddress());
   }
}
```

Output Example:

Local Host Name: MyPC
Local IP Address: 192.168.1.5

3. Get All IP Addresses of a Domain

```
import java.net.*;

public class AllIPAddresses
{
public static void main(String[] args) throws Exception
 {
InetAddress[] addresses = InetAddress.getAllByName("www.google.com");
System.out.println("All IP addresses for www.google.com:");
for (InetAddressaddr : addresses)
 {
System.out.println(addr.getHostAddress());
      }
```

```
    }
}
```

Output Example:

All IP addresses for www.google.com:
142.250.72.196
142.250.72.228
142.250.72.164

(Multiple IPs are returned because large websites use multiple servers for load balancing.)

4. Reverse DNS Lookup (IP → Hostname)

```
import java.net.*;

public class ReverseLookup {
public static void main(String[] args) throws Exception {
InetAddress address = InetAddress.getByName("8.8.8.8");
System.out.println("Host Name: " + address.getHostName());
System.out.println("IP Address: " + address.getHostAddress());
    }
}
```

Output Example:

Host Name: dns.google
IP Address: 8.8.8.8

Note:

- InetAddress class is used to lookup IP addresses and host names.
- Methods like getByName(), getHostName(), and getAllByName() help in DNS lookups.
- You can perform forward (name → IP) and reverse (IP → name) lookups.
- Useful for networkprogramming, pinging hosts, or validating connectivity.

12.5 URLs and URIs

1. Introduction

In network programming, URLsand URIs are essential because they specify how to locate resources over a network. Java provides classes in the java.net package to handle them.

- **URI (Uniform Resource Identifier):** Identifies a resource. Can be a URL or URN.
- **URL (Uniform Resource Locator):** Specifies where a resource is and how to access it.

Example:

URL: https://www.example.com:443/index.html?user=123#section1
URI: https://www.example.com/index.html

2. Components of URL

1. **Protocol / Scheme**: http, https, ftp
2. **Host / Domain**: www.example.com
3. **Port**: Default 80 for HTTP, 443 for HTTPS
4. **Path**: /index.html
5. **Query**: ?user=123
6. **Fragment**: #section1

3. Java Classes

- **java.net.URI** – Represents a URI, allows parsing and constructing URIs.
- **java.net.URL** – Represents a URL, allows connecting to a resource.
- **java.net.URLConnection** – To read/write data from a URL.

4. Example Programs

Example 1: Parsing a URL

```
import java.net.URL;

public class URLExample {
public static void main(String[] args) {
try {
        URL url = new
URL("https://www.example.com:443/index.html?user=123#section1");

System.out.println("Protocol: " + url.getProtocol());
System.out.println("Host: " + url.getHost());
System.out.println("Port: " + url.getPort());
System.out.println("Path: " + url.getPath());
System.out.println("Query: " + url.getQuery());
System.out.println("Reference: " + url.getRef());
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Input: URL is hardcoded.

Output:

```
Protocol: https
Host: www.example.com
Port: 443
Path: /index.html
Query: user=123
```

Reference: section1

Example 2: Parsing a URI

```
importjava.net.URI;

public class URIExample {
public static void main(String[] args) {
try {
        URI uri = new URI("https://www.example.com/index.html?user=123#section1");

System.out.println("Scheme: " + uri.getScheme());
System.out.println("Host: " + uri.getHost());
System.out.println("Port: " + uri.getPort());
System.out.println("Path: " + uri.getPath());
System.out.println("Query: " + uri.getQuery());
System.out.println("Fragment: " + uri.getFragment());
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Output:

```
Scheme: https
Host: www.example.com
Port: -1
Path: /index.html
Query: user=123
Fragment: section1
```

Note: Port -1 indicates default port is used (443 for HTTPS).

Example 3: Reading Data from a URL

```
import java.net.URL;
importjava.io.BufferedReader;
importjava.io.InputStreamReader;

public class URLReadExample {
public static void main(String[] args) {
try {
        URL url = new URL("https://www.example.com");
BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream()));

        String line;
while ((line = reader.readLine()) != null) {
System.out.println(line);
        }
```

```
reader.close();
        } catch (Exception e) {
e.printStackTrace();
        }
    }
}
```

Input: URL of a website.

Output: HTML content of the page (depends on the website).
Example snippet:

```
<!doctype html>
<html>
<head>
<title>Example Domain</title>
</head>
<body>
<div>
<h1>Example Domain</h1>
<p>This domain is for use in illustrative examples in documents.</p>
</div>
</body>
</html>
```

5. Key Points

1. URLs are used to connect to resources over the network in Java.
2. URIs are used mainly for identification and parsing.
3. Java provides URL and URI classes for network programming.
4. You can also read content from URLs using URLConnection or openStream().
5. URL/URI handling is a fundamental part of network programming and web applications.

Difference Between URI and URL

Feature	URL (Uniform Resource Locator)	URI (Uniform Resource Identifier)
Definition	A URL is a type of URI that specifies the location of a resource and how to access it.	A URI is a generic identifier of a resource, which may or may not include its location.
Purpose	Used to locate and access resources over the internet.	Used to identify a resource uniquely, without necessarily specifying how to access it.
Components	Typically includes protocol, host, port, path, query, fragment.	Can include scheme, path, query, fragment, but location/access info is

Feature	URL (Uniform Resource Locator)	URI (Uniform Resource Identifier)
		optional.
Example	https://www.example.com:443/index.html?user=123#section1	https://www.example.com/index.html#section1 or urn:isbn:0451450523
Class in Java	java.net.URL	java.net.URI
Main Use in Networking	Connect to a web resource, read/write data (e.g., HTTP requests).	Parse, manipulate, or compare resource identifiers.

Key Point:

- All URLs are URIs, but not all URIs are URLs.
- URL = URI + access method/location.
- URI = identifier, may not point to an actual resource.

12.6UDP Datagrams and Sockets

1. What is UDP?

- UDP (User Datagram Protocol) is a connectionless protocol used for sending short messages called datagrams over the network.
- It is faster than TCP because it doesn't establish a connection and has no guarantee of delivery, ordering, or error checking.
- Commonly used in:
  - Video streaming
  - Online gaming
  - DNS queries

2. UDP Concepts

Term	Description
Datagram	A packet of data sent over UDP
DatagramSocket	Socket for sending and receiving datagrams
DatagramPacket	Represents a packet to send or receive data
Port	Identifies the application on a host
Connectionless	No persistent connection between client and server

3. Java Classes for UDP

- java.net.DatagramSocket – Creates a socket to send/receive UDP packets.
- java.net.DatagramPacket – Represents data packets.

4. UDP Server Example

```
import java.net.*;
```

```
public class UDPServer {
public static void main(String[] args) {
try {
DatagramSocketserverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];

System.out.println("Server is running... Waiting for client messages.");

while (true) {
DatagramPacketreceivePacket = new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);

            String message = new String(receivePacket.getData(), 0,
receivePacket.getLength());
System.out.println("Received from client: " + message);

if (message.equalsIgnoreCase("exit")) {
System.out.println("Server exiting...");
break;
            }
          }

serverSocket.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

5. UDP Client Example

```
import java.net.*;
importjava.util.Scanner;

public class UDPClient {
public static void main(String[] args) {
try {
DatagramSocketclientSocket = new DatagramSocket();
InetAddressIPAddress = InetAddress.getByName("localhost");
          Scanner sc = new Scanner(System.in);

System.out.println("Enter messages to send to the server (type 'exit' to quit):");
while (true) {
            String message = sc.nextLine();
byte[] sendData = message.getBytes();

DatagramPacketsendPacket = new DatagramPacket(sendData, sendData.length, IPAddress,
9876);
clientSocket.send(sendPacket);
```

```
if (message.equalsIgnoreCase("exit")) {
System.out.println("Client exiting...");
break;
            }
        }

clientSocket.close();
sc.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

6. Sample Input/Output

Client Input:

Hello Server
How are you?
exit

Server Output:

Server is running... Waiting for client messages.
Received from client: Hello Server
Received from client: How are you?
Received from client: exit
Server exiting...

Client Output:

Enter messages to send to the server (type 'exit' to quit):
Hello Server
How are you?
exit
Client exiting...

7. Key Points

- UDP is faster but unreliable compared to TCP.
- No connection is established; packets may arrive outoforder or getlost.
- Useful for real-timeapplications like streaming, VoIP, and gaming.

12.7 Remote Method Invocation

1. What is RMI?

- RMI (Remote Method Invocation) is a Java API that allows an object running in one Java virtual machine (JVM) to invoke methods on an object in another JVM.

- RMI abstracts the underlying network communication, allowing remote method calls as if they were local.
- It is part of java.rmi package.

Use Cases:

- Distributed applications
- Client-server architecture
- Remote services

2. Key Components of RMI

Component	Description
Remote Interface	Declares the methods that can be called remotely
Remote Object	Implements the remote interface and contains the actual business logic
Stub	Client-side proxy for the remote object (generated automatically)
Skeleton	Server-side entity that dispatches client calls to the remote object (Java 2+ not required explicitly)
RMI Registry	Service that maps names to remote objects, allowing clients to look them up

3. Steps to Create RMI Application

1. Define the Remote Interface **(**extends java.rmi.Remote)
2. Implement the Remote Interface
3. Create and start the RMI Server
4. Bind the remote object in the RMI Registry
5. Create the RMI Client to lookup and invoke methods

4. Example: Simple RMI – Adding Two Numbers

Step 1: Remote Interface

```
importjava.rmi.Remote;
importjava.rmi.RemoteException;
public interface AddInterface extends Remote {
int add(int a, int b) throws RemoteException;
}
```

Step 2: Remote Object Implementation

```
importjava.rmi.server.UnicastRemoteObject;
importjava.rmi.RemoteException;

public class AddImplementation extends UnicastRemoteObject implements AddInterface {
publicAddImplementation() throws RemoteException {
super();
```

```
    }

publicint add(int a, int b) throws RemoteException {
return a + b;
    }
}
```

Step 3: RMI Server

```
importjava.rmi.registry.LocateRegistry;
importjava.rmi.registry.Registry;

public class RMIServer {
public static void main(String[] args) {
try {
AddImplementationaddObj = new AddImplementation();
        Registry registry = LocateRegistry.createRegistry(1099); // default port
registry.rebind("AddService", addObj);
System.out.println("Server is running...");
      } catch (Exception e) {
e.printStackTrace();
      }
    }
}
```

Step 4: RMI Client

```
importjava.rmi.registry.LocateRegistry;
importjava.rmi.registry.Registry;

public class RMIClient {
public static void main(String[] args) {
try {
        Registry registry = LocateRegistry.getRegistry("localhost");
AddInterface stub = (AddInterface) registry.lookup("AddService");

int result = stub.add(10, 20);
System.out.println("Result of addition: " + result);
      } catch (Exception e) {
e.printStackTrace();
      }
    }
}
```

5. How to Run RMI Application

1. Compile all Java files:

   javac *.java
2. Start the RMI registry:

   rmiregistry

(Keep it running in the background)

3. Start the server:

   javaRMIServer
4. Run the client:

   javaRMIClient

6. Sample Output

Server Output:

Server is running...

Client Output:

Result of addition: 30

7. Key Points

- RMI allows transparent communication between JVMs.
- Remote interfaces must extend java.rmi.Remote.
- Remote methods must declare throws RemoteException.
- RMI uses stubs and skeletons (proxy objects) to handle network communication.
- Useful for distributed computing and enterprise applications.

12.8 Summary

Network programming in Java allows applications to communicate over networks by using classes from the **java.net** package. It supports both TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), catering to different communication needs—TCP for reliable, connection-oriented data transfer, and UDP for faster, connectionless communication. Fundamental concepts such as IP addresses, ports, and sockets are essential in establishing communication between devices. Java provides classes like Socket and ServerSocket for TCP-based communication and DatagramSocket and DatagramPacket for UDP-based data exchange.

Additionally, Java simplifies working with web resources through the URL and URI classes, enabling developers to connect to web servers, parse links, and retrieve data directly from websites. The InetAddress class assists in resolving hostnames and IP addresses, making DNS lookups straightforward. For applications requiring high-speed, lightweight

communication—such as gaming, live streaming, or chat systems—UDP programming is ideal due to its low overhead and faster transmission.

For building distributed applications, Java provides RMI (Remote Method Invocation), which allows objects running in different JVMs (Java Virtual Machines) to communicate seamlessly as if they were local. RMI involves defining remote interfaces, implementing them on the server, and using an RMI registry for client-server interaction. Overall, Java's networking features make it a powerful and platform-independent choice for developing secure, scalable, and cross-platform networked applications.

12.9 Key Terms

Socket **,**ServerSocket **,**DatagramSocket **,**IP Address **,**Port Number **,**TCP (Transmission Control Protocol) **,**UDP (User Datagram Protocol) **,**URL (Uniform Resource Locator) **,**InetAddress**,**RMI (Remote Method Invocation).

12.10 Self-Assessment Questions

1. What is network programming and why is it used?
2. What is the purpose of an IP address in network communication?
3. What is the difference between TCP and UDP?
4. Which Java class is used to create a server that listens for client connections?
5. What is the function of the Socket class in Java?
6. What does the InetAddress class do?
7. What is the role of a port number in networking?
8. What does URL stand for, and what is it used for?
9. What is RMI and why is it important in Java network programming?
10. Name two real-world applications that use network programming.

12.11 Further Readings

1. Java: The Complete Reference, Twelfth Edition *by Herbert Schildt. McGraw-Hill Education.*
2. Beginning Java Programming: The Object-Oriented Approach *by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.*
3. Java Programming with Oracle JDBC *by Donald Bales. O'Reilly Media.*
4. Java EE 8 Application Development *by David R. Heffelfinger. Packt Publishing.*
5. Professional Java for Web Applications *by Nicholas S. Williams. Wrox/Wiley Publishing.*
6. Java 2: Developer's Guide to Web Applications with JDBC *by Gregory Brill. Sybex.*

Mrs.AppikatlaPushpaLatha

LESSON- 13

INTRODUCTION TO WEB SERVERS AND THE TOMCAT ENVIRONMENT

Aim and Objectives:

- Understand the role of web servers like Apache Tomcat in hosting and managing web applications.
- Learn the concept and purpose of Servlets as Java programs that handle web requests and responses.
- Describe thelifecycle of a Servlet, including methods like init(), service(), and destroy().
- Explore the Java Servlet Development Kit (JSDK) and how it supports servlet creation and deployment.
- Develop and deploy simple Java-based web applications using servlets on the Tomcat server.

STRUCTURE:

13.1 WEB SERVERS AND SERVLETS

Web Servers and Servlets – Brief Information

A **Web Server** is a software application that handles requests from clients (usually web browsers) and responds with web pages, data, or other resources. It uses the HTTP protocol for communication. Examples include Apache Tomcat, GlassFish, and Jetty. A web server hosts web applications, manages connections, and delivers dynamic or static content to users.
A **Servlet** is a Java program that runs on a web server and is used to create dynamic web content. Servlets handle requests (like form submissions) and generate responses (like HTML pages) dynamically. They are part of Java EE (Jakarta EE**)** and run inside a servlet container such as Tomcat.

The **Servlet Lifecycle** consists of:

1. **init()** – Initializes the servlet (runs once).
2. **service()** – Handles client requests (runs repeatedly).
3. **destroy()** – Cleans up resources before servlet is destroyed.

The **Java Servlet Development Kit (JSDK)** provides tools, libraries, and APIs to develop, test, and deploy servlets easily.

In summary, Web Servers host applications, while Servlets are Java components that process web requests and generate dynamic responses—forming the backbone of many Java-based web applications.

13.2 Tomcat web server

1. Introduction

Apache Tomcat is an open-source web server and servlet container developed by the Apache Software Foundation. It is used to run Java-based web applications that use technologies such as Servlets, JSP (JavaServer Pages), and WebSocket.

Tomcat implements the Jakarta Servlet and Jakarta Server Pages (JSP) specifications and serves as the runtime environment for executing servlets and rendering dynamic web content.

2. Key Features

- Supports Servlet and JSP specifications.
- Lightweight, easy to install, and open-source.
- Handles HTTPrequestsand responses efficiently.
- Provides an administrative GUI and management console.
- Integrates easily with IDEs like Eclipse, IntelliJ, or NetBeans.

3. How Tomcat Works

1. A client (web browser) sends an HTTP request (e.g., form submission).
2. Tomcat receives the request and passes it to the servlet container.
3. The servlet processes the request (e.g., accesses a database, processes data).
4. The servlet generates an HTTP response (usually HTML).
5. Tomcat sends the response back to the client's browser.

4. Installing Tomcat

1. Download Tomcat from:https://tomcat.apache.org
2. Extract it to a folder (e.g., C:\apache-tomcat-10.1).
3. Set environment variables:
    - JAVA_HOME → JDK installation path
    - CATALINA_HOME → Tomcat folder
4. Run startup.bat (Windows) or startup.sh (Linux/Mac) in the bin folder.
5. Open a browser and go to:
   👉http://localhost:8080
   You'll see the Tomcat welcome page if it's running properly.

5. Example Servlet Program Using Tomcat

Step 1 – Directory Structure

```
Tomcat
└── webapps
    └── MyApp
        ├── WEB-INF
        │   ├── web.xml
        │   └── classes
        │       └── HelloServlet.class
        └── index.html
```

Step 2 – Servlet Source Code

HelloServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h2>Hello from Tomcat Servlet!</h2>");
out.println("<p>This response is generated by a Java servlet running on Tomcat.</p>");
out.println("</body></html>");
   }
}
```

Step 3 – Deployment Descriptor

web.xml (inside WEB-INF)

```
<web-app>
<servlet>
<servlet-name>HelloServlet</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>HelloServlet</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

Step 4 – HTML File (Optional)

index.html

```
<html>
<head><title>Welcome Page</title></head>
<body>
<h1>Welcome to My Java Web Application</h1>
<p><a href="hello">Click here to invoke the servlet</a></p>
</body>
</html>
```

Step 5 – Compile and Deploy

1. Compile the servlet:
2. javac -classpath "C:\apache-tomcat-10.1\lib\servlet-api.jar" HelloServlet.java
3. Place HelloServlet.class inside:
   C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF\classes
4. Restart Tomcat.
5. Open your browser and go to:
   👉http://localhost:8080/MyApp/hello

6. Example Input and Output

Input (User Action)

User opens browser and visits:

http://localhost:8080/MyApp/hello

Output (Browser Display)

```
<html>
<body>
<h2>Hello from Tomcat Servlet!</h2>
<p>This response is generated by a Java servlet running on Tomcat.</p>
</body>
</html>
```

7. Servlet Lifecycle in Tomcat

1. **Loading** – Tomcat loads the servlet class.
2. **Initialization** – init() is called once when the servlet is created.
3. **Request Handling** – service() calls doGet() or doPost() for each request.
4. **Destruction** – destroy() is called when the server shuts down.

8. Advantages of Tomcat

- Easy to set up and lightweight.
- Excellent support for Java Servlets and JSP.
- Open-source and actively maintained.

- Can be embedded in Java applications.
- Integrates well with development tools and build systems like Maven.

Note:

- Apache Tomcat is a popular web server and servlet container for running Java web applications**.**
- It processes HTTP requests and executes servlets to generate dynamic responses.
- Using Tomcat, developers can create and deploy powerful Java-based web systems easily.

13.3 Introduction to Servlets

A Servlet is a Java program that runs on a web server and is used to create dynamic web content. It acts as a middle layer between a client request (usually from a web browser) and a server-side resource (like a database or another application). Servlets are part of the Jakarta EE (formerly Java EE) platform and are handled by a Servlet container such as Apache Tomcat.

When a user sends a request through a browser (e.g., submitting a form), the servlet receives this request, processes it (e.g., performs calculations or database queries), and generates a response — usually in the form of HTML that is sent back to the browser.

Key Features

- Written in Java, making them platform-independent and secure.
- Provide faster performance compared to traditional CGI (Common Gateway Interface).
- Can handle multiple requests concurrently using multithreading.
- Easily integrated with JSP, databases, and web frameworks.

Basic Servlet Lifecycle

1. **init()** – Called once when the servlet is first loaded.
2. **service()** – Called each time a client request is received.
3. **destroy()** – Called before the servlet is unloaded from memory.

Example: Simple Servlet

HelloServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
```

```
out.println("<h2>Hello, Welcome to Servlets!</h2>");
   }
}
```

web.xml (Deployment Descriptor)

```
<web-app>
<servlet>
<servlet-name>HelloServlet</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>HelloServlet</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

Input / Output Example

Input:
User opens → http://localhost:8080/MyApp/hello

Output:

Hello, Welcome to Servlets!

Note

- Servlets are server-side Java programs used to handle requests and generate dynamic responses.
- They are managed by a servlet container like Tomcat.
- They follow a defined lifecycle (init(), service(), destroy()).
- Servlets are the foundation of Java web applications, often used with JSP and frameworks like Spring MVC.

13.4 Lifecycle of a Servlet

1. Introduction

A **Servlet** is a Java class that extends the capabilities of servers hosting applications accessed through a request–response model (like web applications).The ServletLifecycle defines how a servlet is loaded, initialized, handles requests, anddestroyed by the web container (e.g., Apache Tomcat).

Servlets are managed automatically by the Servlet Container, which is responsible for creating instances, managing threads, and calling lifecycle methods.

2. Servlet Lifecycle Phases

There are five main phases in the lifecycle of a servlet:

1. Loading and Instantiation

- When the web application starts or when the servlet is first requested, the container loads the servlet class into memory and creates an instance of it.

2. Initialization (init() method)

- Called once in the servlet's lifetime.
- Used to **initialize resources** such as database connections or configuration data.

```
public void init() throws ServletException {
   // Initialization code here
}
```

3. Request Handling (service() method)

- Called for each client request**.**
- Determines whether the request is GET**,** POST, etc., and calls corresponding methods (doGet() or doPost()).

```
public void service(ServletRequestreq, ServletResponse res)
throwsServletException, IOException {
   // Handle client request here
}
```

4. Destruction (destroy() method)

- Called once when the servlet is taken out of service or the server shuts down.
- Used to release resources like database connections or file handles.

```
public void destroy() {
   // Cleanup code here
}
```

5. Garbage Collection

- After destruction, the servlet object becomes eligible for garbage collection by the JVM.

3. Servlet Lifecycle Methods Summary

Method	Description	Called By	Times Called
init()	Initializes the servlet	Container	Once
service()	Handles client requests	Container	Multiple times
destroy()	Cleans up before unloading	Container	Once

4. Example Program – Servlet Lifecycle Demonstration

LifecycleServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class LifecycleServlet extends HttpServlet {

public void init() throws ServletException {
System.out.println("Servlet is initializing...");
   }

public void service(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
System.out.println("Servlet is servicing a request...");
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h2>Servlet Lifecycle Example</h2>");
out.println("<p>Request processed successfully!</p>");
out.println("</body></html>");
   }

public void destroy() {
System.out.println("Servlet is being destroyed...");
   }
}
```

web.xml (Deployment Descriptor)

```
<web-app>
<servlet>
<servlet-name>LifecycleServlet</servlet-name>
<servlet-class>LifecycleServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>LifecycleServlet</servlet-name>
<url-pattern>/lifecycle</url-pattern>
</servlet-mapping>
</web-app>
```

5. How to Run the Example

1. Save the servlet file as LifecycleServlet.java.
2. Compile it with the Servlet API jar:
3. javac -classpath "C:\apache-tomcat-10.1\lib\servlet-api.jar" LifecycleServlet.java
4. Place the compiled class in:
5. C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF\classes

6. Add the web.xml file under:
7. C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF\
8. Start Tomcat and open your browser:
   http://localhost:8080/MyApp/lifecycle

6. Example Input and Output

Input (User Action)

User visits:

http://localhost:8080/MyApp/lifecycle

Output on Browser

```
<html>
<body>
<h2>Servlet Lifecycle Example</h2>
<p>Request processed successfully!</p>
</body>
</html>
```

Output on Tomcat Console

```
Servlet is initializing...
Servlet is servicing a request...
```

If the user refreshes the page:

```
Servlet is servicing a request...
```

When Tomcat is stopped or servlet is unloaded:

```
Servlet is being destroyed...
```

7. Explanation of Output

- When the servlet is first requested → init() is called once.
- Every time a client sends a request → service() executes.
- When the server stops → destroy() runs to release resources.

8. Key Points

- A servletcontainer manages the entire lifecycle.
- init() and destroy() are called once per servlet, while service() is called for every request**.**
- The servlet lifecycle ensures efficient resource management and consistent request handling**.**
- Developers can override these methods to add custom initialization**,** logging**,** orcleanup logic.

Note:

The Servlet Lifecycle represents the journey of a servlet from creation to destruction:

Loading → Initialization → Request Handling → Destruction → Garbage Collection

Using lifecycle methods (init(), service(), destroy()), developers can control how servlets behave during different stages of execution. This lifecycle is managed by the web container (like Apache Tomcat), which ensures reliability, scalability, and performance in Java web applications.

13.5 JSDK

1. Introduction

JSDK (Java Servlet Development Kit) is a toolkit provided by Sun Microsystems (now Oracle) that contains the libraries, classes, and tools required to develop, test, and run Java Servlets.

It was introduced to help developers build dynamic web applications before the release of full enterprise editions like J2EE and later Jakarta EE.

Today, the JSDK's functionality is integrated into modern Java EE / Jakarta EE servers (like Tomcat, GlassFish, and WildFly), but understanding it remains essential for the fundamentals of servlet development.

2. Purpose of JSDK

JSDK provides:

- APIs to develop Servlets and JSPs.
- Tools for compiling, running, and testing servlets locally.
- The Servlet API classes like javax.servlet and javax.servlet.http.
- A small web server (in older versions) to test servlets.

Modern servlet containers (like Tomcat) already include these libraries, but the concept of JSDK is foundational for understanding servlet development.

3. Important Packages in JSDK

Package Name	Description
javax.servlet	Contains core classes and interfaces for building servlets.
javax.servlet.http	Contains classes for HTTP-specific functionalities (GET, POST requests, sessions, cookies).

Common Classes/Interfaces:

- Servlet – Basic interface for all servlets.

- GenericServlet – Provides a framework for non-HTTP servlets.
- HttpServlet – Provides methods for handling HTTP requests (doGet(), doPost()).
- ServletRequest, ServletResponse – Represent client request and server response objects.

4. JSDK Architecture

A servlet works with the help of:

- **Client (Browser):** Sends HTTP requests.
- **Web Server / Servlet Container:** Runs servlets and manages the servlet lifecycle.
- **Servlet:** Processes requests and sends responses.

The JSDK provides the API layer that enables this communication between client and server.

5. Example Program Using JSDK

HelloServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class HelloServlet extends HttpServlet {
public void init() throws ServletException {
System.out.println("Servlet Initialized");
   }

public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h2>Welcome to JSDK Servlet Example</h2>");
out.println("<p>This servlet is running using the Servlet API from JSDK.</p>");
out.println("</body></html>");
   }

public void destroy() {
System.out.println("Servlet Destroyed");
   }
}
```

6. Deployment Descriptor (web.xml)

```
<web-app>
<servlet>
<servlet-name>HelloServlet</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>
```

```
<servlet-mapping>
<servlet-name>HelloServlet</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

7. Steps to Run the Example Using Tomcat (Modern Equivalent of JSDK)

1. Install Apache Tomcat.
2. Save HelloServlet.java in:
   C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF\classes
3. Save web.xml in:
   C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF
4. Compile the servlet:
5. javac -classpath "C:\apache-tomcat-10.1\lib\servlet-api.jar" HelloServlet.java
6. Start Tomcat and open browser:
   ☞http://localhost:8080/MyApp/hello

8. Example Input and Output

Input (User Action)

User opens:

http://localhost:8080/MyApp/hello

Output (Browser Display)

```
<html>
<body>
<h2>Welcome to JSDK Servlet Example</h2>
<p>This servlet is running using the Servlet API from JSDK.</p>
</body>
</html>
```

Tomcat Console Output

Servlet Initialized

When stopping the server:

Servlet Destroyed

9. Explanation

- The JSDK API provides the servlet classes used here (HttpServlet, ServletRequest, ServletResponse).
- When the servlet is first requested, the container loads and initializes it by calling init().
- Each browser request calls the **d**oGet**()** method.
- When the server shuts down or redeploys the application, destroy**()** is called.

10. Modern Equivalent

Today, instead of using the old standalone JSDK, developers use:

- **Apache Tomcat**
- **Jakarta EE (Servlet 5.0 and above)**
- **Maven/Gradle** for dependency management, using:
- <dependency>
- <groupId>jakarta.servlet</groupId>
- <artifactId>jakarta.servlet-api</artifactId>
- <version>5.0.0</version>
- <scope>provided</scope>
- </dependency>

11. Advantages of JSDK

- Simplifies servlet and JSP development.
- Provides standard API for all servlet containers.
- Promotes platform independence and code reusability.
- Lays the foundation for modern Java web frameworks.
- Allows testing servlets locally before deployment.

Note:

Aspect	Description
Full Form	Java Servlet Development Kit
Purpose	Provides API and tools for developing servlets
Main Packages	javax.servlet, javax.servlet.http
Lifecycle Methods	init(), service(), destroy()
Modern Equivalent	Apache Tomcat / Jakarta Servlet API
Output	Dynamic HTML content via Java code

13.6 SUMMARY

The Java Servlet Development Kit (JSDK), developed by Sun Microsystems (now Oracle), provides the essential tools, libraries, and APIs required for building, testing, and deploying Java servlets—programs that run on web servers to handle client requests and generate dynamic responses. Servlets form the foundation of Java-based web applications and are managed by servlet containers like Apache Tomcat, which implement the Servlet and JSP specifications. The servlet lifecycle includes three key phases: initialization using init(), request handling using service() (or doGet()/doPost() for HTTP requests), and cleanup through destroy(). This lifecycle ensures efficient resource management and reliable request processing across multiple clients.

Using the JSDK, developers can compile servlets, deploy them under the WEB-INF/classes directory, and configure them in the web.xml deployment descriptor. Servlets are significantly faster and more scalable than traditional CGI programs, as they support

multithreading—allowing multiple client requests to be processed simultaneously within the same server process. The JSDK includes core packages such as javax.servlet and javax.servlet.http, which define essential classes and interfaces like HttpServlet, ServletRequest, and ServletResponse. These enable interaction between web clients and servers, session management, and dynamic HTML generation.

Modern Java web development builds upon the foundation laid by JSDK, often using Apache Tomcat, Jakarta EE, and build tools like Maven or Gradle for streamlined deployment and dependency management. Despite advancements in frameworks, understanding servlets and the JSDK remains essential for grasping the underlying principles of Java web technologies. Together, they provide a robust, scalable, and maintainable framework for creating interactive, platform-independent, and secure web applications that respond dynamically to user input.

13.7 KEY TERMS

Servlet, Tomcat,Web Server,JSDK,HTTP, JSP (JavaServer Pages),init(),service(),destroy(),Servlet Container

13.8 SELF-ASSESSMENT QUESTIONS

1. What is a web server, and what is its primary function?
2. Define a servlet in Java.
3. Name two popular web servers used with Java applications.
4. What are the three main methods in the servlet lifecycle?
5. What is the role of the init() method in a servlet?
6. How does the service() method handle client requests?
7. What is Tomcat, and why is it commonly used?
8. What is the purpose of the JSDK?
9. Explain the difference between static and dynamic content in web applications.
10. Where do you place a compiled servlet class in a Tomcat web application?

13.9 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition *by Herbert Schildt. McGraw-Hill Education.*
2. Beginning Java Programming: The Object-Oriented Approach *by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.*
3. Java Programming with Oracle JDBC *by Donald Bales. O'Reilly Media.*
4. Java EE 8 Application Development *by David R. Heffelfinger. Packt Publishing.*
5. Professional Java for Web Applications *by Nicholas S. Williams. Wrox/Wiley Publishing.*
6. Java 2: Developer's Guide to Web Applications with JDBC *by Gregory Brill. Sybex.*

Dr. U. Surya Kameswari

CHAPTER 14
INTRODUCTION TO SERVLETS AND HTTP PROGRAMMING

Aim and Objectives:

- Understand the purpose and functionality of the Servlet API in Java web development.
- Learn the core interfaces and classes provided by the javax.servletpackage.
- Explore how to read client request parameters using servlet request methods.
- Understand how to access and use initialization parameters for servlet configuration.
- Study the javax.servlet.httppackage for handling HTTP-specific requests and responses.

STRUCTURE:

14.1 The Servlet API

1. Introduction to Servlet API

A **Servlet** is a Java program that runs on a web server and handles client requests (usually from a web browser) and responses.Servlets are part of the Java EE (Jakarta EE**)** platform and are used to create dynamic web content.

The **Servlet API** provides the classes and interfaces necessary to build and manage servlets.

2. Servlet API Packages

Servlet functionality is mainly provided through two packages:

Package	Description
javax.servlet	Contains general classes and interfaces for servlets.
javax.servlet.http	Provides classes and interfaces specific to HTTP-based servlets.

3. Key Interfaces in Servlet API

Interface	Description
Servlet	Defines methods that all servlets must implement.
ServletRequest	Represents the client's request.
ServletResponse	Represents the server's response.
ServletConfig	Provides configuration information for a servlet.
ServletContext	Provides application-wide information and methods.
HttpServletRequest	Extends ServletRequest; used for HTTP-specific requests.
HttpServletResponse	Extends ServletResponse; used for HTTP-specific responses.
HttpSession	Used for managing user sessions.

4. Servlet Life Cycle

The **lifecycle** of a servlet is managed by the servlet container (e.g., Tomcat).

Stage	Method	Description
1. Loading & Instantiation	Constructor	Servlet is loaded into memory.
2. Initialization	init()	Called once when the servlet is first loaded.
3. Request Handling	service() or doGet(), doPost()	Called for each request.
4. Destruction	destroy()	Called once before the servlet is unloaded.

5. Basic Servlet Example

File: HelloServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class HelloServlet extends HttpServlet {

public void init() throws ServletException {
System.out.println("Servlet Initialized");
   }

public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();

out.println("<html><body>");
out.println("<h2>Welcome to the Servlet API Example!</h2>");
out.println("</body></html>");
   }

public void destroy() {
System.out.println("Servlet Destroyed");
   }
}
```

HTML Form (hello.html)

```
<!DOCTYPE html>
<html>
<body>
<form action="HelloServlet" method="get">
<input type="submit" value="Click Me">
</form>
</body>
</html>
```

Input:

User clicks the **"Click Me"** button.

Output (Browser):

Welcome to the Servlet API Example!

6. Example: Using ServletRequest and ServletResponse

File: RequestInfoServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class RequestInfoServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
```

```
PrintWriter out = response.getWriter();

        String method = request.getMethod();
        String uri = request.getRequestURI();
        String clientIP = request.getRemoteAddr();

out.println("<html><body>");
out.println("<h3>Request Information</h3>");
out.println("<p>Method: " + method + "</p>");
out.println("<p>URI: " + uri + "</p>");
out.println("<p>Client IP: " + clientIP + "</p>");
out.println("</body></html>");
    }
}
```

Input:

User visits http://localhost:8080/RequestInfoServlet

Output:

Request Information
Method: GET
URI: /RequestInfoServlet
Client IP: 127.0.0.1

7. Example: Reading Form Data

File: FormServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class FormServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

        String name = request.getParameter("name");
        String email = request.getParameter("email");

out.println("<html><body>");
```

```
out.println("<h3>Form Data Received:</h3>");
out.println("<p>Name: " + name + "</p>");
out.println("<p>Email: " + email + "</p>");
out.println("</body></html>");
    }
}
```

HTML Form (form.html)

```
<!DOCTYPE html>
<html>
<body>
<form action="FormServlet" method="post">
      Name: <input type="text" name="name"><br>
      Email: <input type="text" name="email"><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Input:

Name: Abc
Email: abc@example.com

Output:

Form Data Received:
Name: Abc
Email: abc@example.com

8. Example: Using ServletConfig and ServletContext

web.xml Configuration

```
<web-app>
<servlet>
<servlet-name>ConfigServlet</servlet-name>
<servlet-class>ConfigServlet</servlet-class>
<init-param>
<param-name>adminEmail</param-name>
<param-value>admin@site.com</param-value>
</init-param>
</servlet>

<servlet-mapping>
```

```
<servlet-name>ConfigServlet</servlet-name>
<url-pattern>/ConfigServlet</url-pattern>
</servlet-mapping>

<context-param>
<param-name>company</param-name>
<param-value> Technologies</param-value>
</context-param>
</web-app>
```

Servlet File: ConfigServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class ConfigServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

ServletConfigconfig = getServletConfig();
      String email = config.getInitParameter("adminEmail");

ServletContext context = getServletContext();
      String company = context.getInitParameter("company");

out.println("<html><body>");
out.println("<h3>Configuration Details</h3>");
out.println("<p>Admin Email: " + email + "</p>");
out.println("<p>Company: " + company + "</p>");
out.println("</body></html>");
   }
}
```

Input:

User visits http://localhost:8080/ConfigServlet

Output:

Configuration Details
Admin Email: admin@site.com
Company: Technologies

9. Example: Using HttpSession

File: SessionExampleServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class SessionExampleServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String username = request.getParameter("username");
HttpSession session = request.getSession();
session.setAttribute("user", username);

out.println("<html><body>");
out.println("<h3>Welcome, " + username + "!</h3>");
out.println("<a href='SessionDisplayServlet'>Go to Next Page</a>");
out.println("</body></html>");
   }
}
```

File: SessionDisplayServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class SessionDisplayServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

HttpSession session = request.getSession(false);
      String name = (String) session.getAttribute("user");

out.println("<html><body>");
out.println("<h3>Hello again, " + name + "!</h3>");
out.println("<p>Your session is active.</p>");
out.println("</body></html>");
```

```
    }
}
```

Input:

Username: Abc

Output:

Page 1:

Welcome, Abc!
Go to Next Page

Page 2 (After clicking link):

Hello again, Abc!
Your session is active.

10. Table

Concept	Description
Servlet API	Provides classes and interfaces for building web apps.
ServletRequest / Response	Handle client requests and server responses.
HttpServlet	Base class for HTTP-specific servlets.
ServletConfig / Context	Provide configuration and application info.
HttpSession	Tracks user data across multiple requests.
Lifecycle Methods	init(), service(), destroy().
Input Handling	request.getParameter() reads form data.
Output Generation	response.getWriter() sends HTML to browser.

Note:
The **Servlet API** is the backbone of Java web development. It provides powerful classes and interfaces to process HTTP requests, generate dynamic responses, manage sessions, and configure web applications securely and efficiently.

14.2 The javax.servlet Package

1. What is javax.servlet?
javax.servlet is a Java package that provides classes and interfaces for building server-side web application**s** — primarily Servlets and Filters — that run on a web server or application server (like Apache Tomcat, Jetty, GlassFish, etc.).

It is part of Java EE (Jakarta EE**)** and used to handle HTTP requests and responses.

2. Key Components of javax.servlet Package

Component	Description
Servlet Interface	Defines methods all servlets must implement (init(), service(), destroy(), etc.).
GenericServlet Class	Implements Servlet interface; can be extended for non-HTTP protocols.
HttpServlet Class	Extends GenericServlet and adds HTTP-specific methods like doGet() and doPost().
ServletRequest	Represents client request; gives access to parameters, headers, etc.
ServletResponse	Represents response sent back to the client.
ServletConfig	Provides servlet configuration data (init parameters).
ServletContext	Provides information shared among servlets (application-wide context).
RequestDispatcher	Forwards requests to another servlet or resource.

3. Servlet Lifecycle

1. **Loading and Instantiation** → Servlet class is loaded.
2. **Initialization (init())** → Called once.
3. **Request Handling (service(), doGet(), doPost())** → Called for each request.
4. **Destruction (destroy())** → Called once before servlet is destroyed.

4. Example 1 — Basic "Hello World" Servlet

Directory Structure

```
MyServletApp/
├── WEB-INF/
│   ├── web.xml
│   └── classes/
│       └── HelloServlet.class
└── index.html
```

(a) HelloServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class HelloServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();

out.println("<html><body>");
out.println("<h2>Hello, Welcome to the Java Servlet Example!</h2>");
out.println("</body></html>");
   }
}
```

(b) web.xml (Deployment Descriptor)

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<servlet>
<servlet-name>hello</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>hello</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>

</web-app>
```

(c) index.html

```
<!DOCTYPE html>
<html>
<head><title>Servlet Test</title></head>
<body>
<h1>Click the link to test the servlet</h1>
<a href="hello">Say Hello</a>
</body>
</html>
```

Input

User opens:

http://localhost:8080/MyServletApp/hello

Output (Browser)

```
<html><body>
<h2>Hello, Welcome to the Java Servlet Example!</h2>
```

```
</body></html>
```

5. Example 2 — Servlet with Request Parameters (GET/POST)

(a) FormServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class FormServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String name = request.getParameter("username");

out.println("<html><body>");
out.println("<h3>Hello, " + name + "! Welcome to Servlets.</h3>");
out.println("</body></html>");
   }
}
```

(b) form.html

```
<!DOCTYPE html>
<html>
<head><title>Form Example</title></head>
<body>
<form action="form" method="post">
      Enter your name: <input type="text" name="username" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

(c) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
<servlet>
<servlet-name>formServlet</servlet-name>
<servlet-class>FormServlet</servlet-class>
</servlet>
<servlet-mapping>
```

```
<servlet-name>formServlet</servlet-name>
<url-pattern>/form</url-pattern>
</servlet-mapping>
</web-app>
```

Input

User fills out form:

Enter your name: Alice

and clicks “Submit”.

Output (Browser)

```
<html><body>
<h3>Hello, Alice! Welcome to Servlets.</h3>
</body></html>
```

6. Common Interfaces in javax.servlet

Interface	Description
Servlet	Base interface for all servlets.
ServletRequest	Request object (data from client).
ServletResponse	Response object (data to client).
Filter	Used to preprocess or postprocess requests.
RequestDispatcher	Forward/include requests to another resource.
ServletContext	Application-level information.
ServletConfig	Servlet-specific configuration data.

7. Deployment & Execution

1. Compile .java → .class
2. Place .class files in WEB-INF/classes/
3. Add mappings in web.xml
4. Deploy project to server (like Tomcat’s webapps folder)
5. Start Tomcat → Access URL in browser.

8. Example Output

Servlet	Input	Output (Browser)
HelloServlet	URL /hello	"Hello, Welcome to the Java Servlet Example!"
FormServlet	POST form with "Alice"	"Hello, Alice! Welcome to Servlets."

9. Transition to Jakarta Servlet

Since **Jakarta EE 9**, the package name changed from
javax.servlet.* → jakarta.servlet.*
But functionality remains the same.

14.3 Reading Servlet parameters

1. What Are Servlet Parameters?

Servlet parameters are data sent by a client (browser**)** to the server (servlet**)** through an HTTPrequest.

There are two types of parameters you can read in servlets:

1. **Request Parameters** – Sent by the client through HTML forms, query strings, or links.
2. **Initialization Parameters** – Configured in web.xml for a servlet or application.

2. Methods to Read Request Parameters

From the HttpServletRequest object:

Method	Description
getParameter(String name)	Returns a single parameter value (as String).
getParameterValues(String name)	Returns multiple values (like checkboxes).
getParameterNames()	Returns all parameter names (as Enumeration).
getParameterMap()	Returns all parameters and their values (as a Map).

3. Example 1 — Reading Parameters from an HTML Form (POST Method)

(a) form.html

```
<!DOCTYPE html>
<html>
<head><title>User Form</title></head>
<body>
<h2>User Registration</h2>
```

```
<form action="register" method="post">
        Name: <input type="text" name="username" /><br><br>
        Email: <input type="text" name="email" /><br><br>
        Gender:
<input type="radio" name="gender" value="Male"> Male
<input type="radio" name="gender" value="Female"> Female<br><br>
        Hobbies:
<input type="checkbox" name="hobby" value="Reading"> Reading
<input type="checkbox" name="hobby" value="Music"> Music
<input type="checkbox" name="hobby" value="Sports"> Sports<br><br>
<input type="submit" value="Register" />
</form>
</body>
</html>
```

(b) RegisterServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class RegisterServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

        // Set response type
response.setContentType("text/html");
PrintWriter out = response.getWriter();

        // Reading parameters from the form
        String name = request.getParameter("username");
        String email = request.getParameter("email");
        String gender = request.getParameter("gender");
String[] hobbies = request.getParameterValues("hobby");

        // HTML Output
out.println("<html><body>");
out.println("<h2>User Registration Details</h2>");
out.println("<p><b>Name:</b> " + name + "</p>");
out.println("<p><b>Email:</b> " + email + "</p>");
out.println("<p><b>Gender:</b> " + gender + "</p>");

if (hobbies != null) {
out.println("<p><b>Hobbies:</b></p><ul>");
```

```
for (String h :hobbies) {
out.println("<li>" + h + "</li>");
        }
out.println("</ul>");
      } else {
out.println("<p><b>Hobbies:</b> None selected</p>");
      }

out.println("</body></html>");
   }
}
```

(c) web.xml (Deployment Descriptor)

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<servlet>
<servlet-name>register</servlet-name>
<servlet-class>RegisterServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>register</servlet-name>
<url-pattern>/register</url-pattern>
</servlet-mapping>

</web-app>
```

Input (From Browser Form)

Name: Alice Johnson
Email: alice@example.com
Gender: Female
Hobbies: Reading, Music

Output (Browser)

```
<html><body>
<h2>User Registration Details</h2>
<p><b>Name:</b> Alice Johnson</p>
<p><b>Email:</b> alice@example.com</p>
<p><b>Gender:</b> Female</p>
<p><b>Hobbies:</b></p>
<ul>
<li>Reading</li>
<li>Music</li>
```

```
</ul>
</body></html>
```

4. Example 2 — Reading Multiple Parameters Using getParameterNames()

(a) ParameterListServlet.java

```
import java.io.*;
importjava.util.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class ParameterListServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      Enumeration<String>paramNames = request.getParameterNames();

out.println("<html><body>");
out.println("<h3>All Parameters Received:</h3>");
out.println("<ul>");

while (paramNames.hasMoreElements()) {
         String paramName = paramNames.nextElement();
String[] values = request.getParameterValues(paramName);

out.print("<li><b>" + paramName + ":</b> ");
for (inti = 0; i<values.length; i++) {
out.print(values[i]);
if (i<values.length - 1)
out.print(", ");
          }
out.println("</li>");
       }

out.println("</ul></body></html>");
   }
}
```

Input URL

http://localhost:8080/MyApp/params?user=Bob&city=Delhi&hobby=Music&hobby=Cricket

Output (Browser)

<html><body>
<h3>All Parameters Received:</h3>

user: Bob
city: Delhi
hobby: Music, Cricket

</body></html>

5. Table

Method	Usage	Example
getParameter("name")	Get single value	String n = req.getParameter("name");
getParameterValues("hobby")	Get multiple values	String[] h = req.getParameterValues("hobby");
getParameterNames()	Get all parameter names	Enumeration e = req.getParameterNames();
getParameterMap()	Get all parameters as map	Map m = req.getParameterMap();

6. Key Points

- Request parameters are **always strings** (even if numbers are entered).
- They can be sent through:
  - HTML forms (GET or POST)
  - Query strings (?param=value)
  - AJAX requests
- You can convert numeric strings to integers with Integer.parseInt() if needed.
- For file uploads, use getPart() from javax.servlet.http.Part (Servlet 3.0+).

14.4. Reading Initialization parameters

1. What Are Initialization Parameters?

Initialization parameters are configuration values defined in the web.xml file (or via annotations in newer Java EE/Jakarta EE versions).

They are not sent by the client — instead, they are set by the developer to configure servlet behavior, like:

- Database connection info
- File paths
- Application settings

There Are Two Types:

Type	Description	Accessed By
Servlet Initialization Parameters	Specific to a single servlet.	ServletConfig object
Context Initialization Parameters	Shared across the entire web app.	ServletContext object

2. Methods to Read Initialization Parameters

From ServletConfig (for one servlet)

Method	Description
getInitParameter(String name)	Returns the value of a specific parameter.
getInitParameterNames()	Returns all parameter names.

From ServletContext (shared for all servlets)

Method	Description
getInitParameter(String name)	Returns the value of a context-wide parameter.
getInitParameterNames()	Returns all context-wide parameter names.

3. Example 1 — Servlet-Specific Initialization Parameters

(a) DatabaseServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DatabaseServlet extends HttpServlet {

   private String dbUrl;
   private String dbUser;
   private String dbPassword;

   public void init() throws ServletException {
      // Get initialization parameters from web.xml
      ServletConfig config = getServletConfig();
      dbUrl = config.getInitParameter("dbURL");
      dbUser = config.getInitParameter("username");
      dbPassword = config.getInitParameter("password");
   }

   public void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
        throws ServletException, IOException {

response.setContentType("text/html");
      PrintWriter out = response.getWriter();

out.println("<html><body>");
out.println("<h2>Database Configuration Details</h2>");
out.println("<p><b>Database URL:</b> " + dbUrl + "</p>");
out.println("<p><b>Username:</b> " + dbUser + "</p>");
out.println("<p><b>Password:</b> " + dbPassword + "</p>");
out.println("</body></html>");
   }
}
```

(b) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<servlet>
<servlet-name>dbServlet</servlet-name>
<servlet-class>DatabaseServlet</servlet-class>

<!-- Servlet-specific initialization parameters -->
<init-param>
<param-name>dbURL</param-name>
<param-value>jdbc:mysql://localhost:3306/mydb</param-value>
</init-param>
<init-param>
<param-name>username</param-name>
<param-value>admin</param-value>
</init-param>
<init-param>
<param-name>password</param-name>
<param-value>secret123</param-value>
</init-param>
</servlet>

<servlet-mapping>
<servlet-name>dbServlet</servlet-name>
<url-pattern>/dbinfo</url-pattern>
</servlet-mapping>

</web-app>
```

Input (Browser Request)

http://localhost:8080/MyApp/dbinfo

Output (Browser)

```
<html><body>
<h2>Database Configuration Details</h2>
<p><b>Database URL:</b>jdbc:mysql://localhost:3306/mydb</p>
<p><b>Username:</b> admin</p>
<p><b>Password:</b> secret123</p>
</body></html>
```

4. Example 2 — Application-Wide Initialization Parameters (ServletContext)

These are shared across allservlets in your application.

(a) ConfigServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ConfigServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      ServletContext context = getServletContext();
      String company = context.getInitParameter("company");
      String supportEmail = context.getInitParameter("supportEmail");

out.println("<html><body>");
out.println("<h2>Application Configuration</h2>");
out.println("<p><b>Company Name:</b> " + company + "</p>");
out.println("<p><b>Support Email:</b> " + supportEmail + "</p>");
out.println("</body></html>");
   }
}
```

(b) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
```

```
<!-- Context-wide (application) initialization parameters -->
<context-param>
<param-name>company</param-name>
<param-value>Tech Innovators Pvt. Ltd.</param-value>
</context-param>

<context-param>
<param-name>supportEmail</param-name>
<param-value>support@techinnovators.com</param-value>
</context-param>

<servlet>
<servlet-name>configServlet</servlet-name>
<servlet-class>ConfigServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>configServlet</servlet-name>
<url-pattern>/config</url-pattern>
</servlet-mapping>

</web-app>
```

Input (Browser Request)

http://localhost:8080/MyApp/config

Output (Browser)

```
<html><body>
<h2>Application Configuration</h2>
<p><b>Company Name:</b> Tech Innovators Pvt. Ltd.</p>
<p><b>Support Email:</b> support@techinnovators.com</p>
</body></html>
```

5. Comparison Table

Type	Defined In	Accessed Using	Scope	Example Usage
Servlet Init Parameter	Inside <servlet> tag in web.xml	ServletConfig	Specific to one servlet	DB username/password
Context Init Parameter	At top level <context-param> in web.xml	ServletContext	Shared among all servlets	Company name, global email, version info

6. Key Points

- init() method is called only once, when the servlet is first loaded.
- Initialization parameters are read-only.
- They are useful for configuration values that might change per deployment (like DB URLs).
- Using ServletContext, all servlets in the web app can share the same parameters.

7. Example Output Summary

Servlet	Input (URL)	Output (Browser)
DatabaseServlet	/dbinfo	Shows servlet-specific DB parameters
ConfigServlet	/config	Shows app-wide company & support info

14.5 The javax.servlet HTTP package

1. What Is javax.servlet.http?

The javax.servlet.http package extends the javax.servlet package to support HTTP-specific functionality.

It provides classes and interfaces for handling:

- HTTP requests and responses
- Cookies
- Sessions
- State management
- HTTP methods like GET, POST, PUT, DELETE, etc.

2. Major Classes & Interfaces in javax.servlet.http

Class / Interface	Description
HttpServlet	Base class for creating HTTP servlets. You extend this class to create your own servlet.
HttpServletRequest	Represents the client's HTTP request (headers, parameters, cookies, etc.).
HttpServletResponse	Represents the HTTP response sent to the client.
HttpSession	Used for session tracking (stores data between multiple requests).
Cookie	Represents HTTP cookies for client-side state management.
HttpSessionBindingListener	Notified when objects are bound/unbound to a session.

Class / Interface	Description
HttpServletRequestWrapper / HttpServletResponseWrapper	Used to modify requests or responses.

3. The HttpServlet Class

Important Methods

Method	Description
doGet(HttpServletRequest req, HttpServletResponse res)	Handles HTTP GET requests.
doPost(HttpServletRequest req, HttpServletResponse res)	Handles HTTP POST requests.
doPut(), doDelete()	Handle other HTTP methods.
getServletInfo()	Returns info about the servlet.

4. Example 1 — Basic HTTP Servlet Using doGet()

(a) HelloHttpServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloHttpServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      // Set the content type
response.setContentType("text/html");

      // Get output writer
      PrintWriter out = response.getWriter();

      // Write response
out.println("<html><body>");
out.println("<h2>Welcome to javax.servlet.http Example!</h2>");
out.println("<p>This response is generated by doGet() method.</p>");
out.println("</body></html>");
   }
}
```

(b) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
```

```
<servlet>
<servlet-name>helloHttp</servlet-name>
<servlet-class>HelloHttpServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>helloHttp</servlet-name>
<url-pattern>/helloHttp</url-pattern>
</servlet-mapping>

</web-app>
```

Input (Browser URL)

http://localhost:8080/MyApp/helloHttp

Output (Browser)

```
<html><body>
<h2>Welcome to javax.servlet.http Example!</h2>
<p>This response is generated by doGet() method.</p>
</body></html>
```

5. Example 2 — Reading Request Data Using HttpServletRequest

(a) UserInfoServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class UserInfoServlet extends HttpServlet {

   protected void doPost(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      // Set content type
response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      // Read parameters from form
      String name = request.getParameter("username");
      String city = request.getParameter("city");

out.println("<html><body>");
out.println("<h2>User Information</h2>");
```

```
out.println("<p><b>Name:</b> " + name + "</p>");
out.println("<p><b>City:</b> " + city + "</p>");
out.println("</body></html>");
   }
}
```

(b) form.html

```
<!DOCTYPE html>
<html>
<head><title>User Info</title></head>
<body>
<form action="userinfo" method="post">
      Name: <input type="text" name="username"><br><br>
      City: <input type="text" name="city"><br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

(c) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
<servlet>
<servlet-name>userinfo</servlet-name>
<servlet-class>UserInfoServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>userinfo</servlet-name>
<url-pattern>/userinfo</url-pattern>
</servlet-mapping>
</web-app>
```

Input

User submits the form:

Name: Alice
City: New York

Output (Browser)

```
<html><body>
<h2>User Information</h2>
<p><b>Name:</b> Alice</p>
<p><b>City:</b> New York</p>
```

```
</body></html>
```

6. Example 3 — Using Cookies (javax.servlet.http.Cookie)

(a) CookieServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CookieServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      // Create a new cookie
      Cookie userCookie = new Cookie("username", "JohnDoe");
      userCookie.setMaxAge(60 * 60); // 1 hour
response.addCookie(userCookie);

out.println("<html><body>");
out.println("<h3>Cookie has been set!</h3>");
out.println("</body></html>");
   }
}
```

Input

```
http://localhost:8080/MyApp/cookie
```

Output (Browser)

```
<html><body>
<h3>Cookie has been set!</h3>
</body></html>
```

(Browser stores cookie username=JohnDoe)

7. Example 4 — Managing Sessions (HttpSession)

(a) SessionServlet.java

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

```
import javax.servlet.*;
import javax.servlet.http.*;

publicclass SessionServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      HttpSession session = request.getSession();
      Integer count = (Integer) session.getAttribute("count");

      if (count == null) {
         count = 1;
      } else {
         count++;
      }

session.setAttribute("count", count);

out.println("<html><body>");
out.println("<h2>Session Example</h2>");
out.println("<p>Session ID: " + session.getId() + "</p>");
out.println("<p>You have visited this page " + count + " times.</p>");
out.println("</body></html>");
   }
}
```

Input (Repeated Browser Visits)

http://localhost:8080/MyApp/session

Output (Browser)

First visit:

```
<h2>Session Example</h2>
<p>Session ID: A12F5C9D2345E...</p>
<p>You have visited this page 1 times.</p>
```

Second visit:

```
<h2>Session Example</h2>
<p>Session ID: A12F5C9D2345E...</p>
```

You have visited this page 2 times.

8. Summary of javax.servlet.http Components

Class / Interface	Purpose	Example Usage
HttpServlet	Base class for HTTP servlets	Extend to create custom servlets
HttpServletRequest	Access client request data	req.getParameter("name")
HttpServletResponse	Send data to client	res.getWriter().println()
HttpSession	Store user session info	session.setAttribute()
Cookie	Manage client-side state	response.addCookie()

9. Key Points

- All HTTP servlets must extend HttpServlet.
- doGet() and doPost() handle client requests.
- HttpServletRequest gives you:
  - Parameters
  - Headers
  - Cookies
  - Session
- HttpServletResponse allows sending HTML, JSON, or file data back.
- Cookies and sessions are used for state management across multiple requests.

10. Output Summary

Servlet	Input (URL or Form)	Output (Browser)
HelloHttpServlet	/helloHttp	Static HTML greeting
UserInfoServlet	POST form	Displays submitted name & city
CookieServlet	/cookie	Sets a cookie
SessionServlet	/session	Shows visit count per session

14.6 SUMMARY

The Servlet API is a core part of Java EE (Jakarta EE) used to create dynamic web applications that process client requests and generate responses. It provides two main packages — **b** for generic servlet functionality and javax.servlet.http for HTTP-specific features. The Servlet interface defines the basic structure and lifecycle of a servlet, including methods like init(), service(), and destroy(). Servlets operate within a servlet container such as Apache Tomcat, which manages their lifecycle and communication with clients. Using **Se**rvletRequest and **S**ervletResponse, developers can read client data and send HTML or other types of output to the browser.

The javax.servletpackage includes key components like ServletConfig for servlet-specific configuration and ServletContext for application-wide settings. It also supports forwarding requests using RequestDispatcher and filtering through the Filter interface. Servlets can read request parameters from forms using methods like getParameter(), getParameterValues(), and getParameterNames(), allowing them to process user input efficiently. Additionally, initialization parameters defined in web.xml provide configurable values for servlets or the entire application through ServletConfig and ServletContext.

The javax.servlet.httppackage extends this functionality to handle HTTP requests and responses. It introduces classes such as HttpServlet, HttpServletRequest, and HttpServletResponse, which manage web-based interactions. It also provides HttpSession for maintaining user sessions and Cookie for client-side state tracking. Methods like doGet() and doPost() are used to handle different HTTP request types. Together, these APIs enable Java developers to build secure, scalable, and interactive web applications capable of handling sessions, cookies, and dynamic content effectively.

14.7 KEY TERMS

Servlet,Servlet API,ServletRequest,ServletResponse,ServletConfig,ServletContext,HttpServlet,HttpSession,Cookie,Initialization Parameters.

14.8 SELF-ASSESSMENT QUESTIONS

1. What is a Servlet?
2. Which interface do all servlets implement?
3. What is the purpose of the ServletRequest object?
4. What does the ServletResponse object do?
5. How is ServletConfig different from ServletContext?
6. What class do most HTTP servlets extend?
7. How can you store data for a user session?
8. What is a Cookie used for?
9. How do you pass initialization parameters to a servlet?
10. Which method is called first when a servlet is loaded?

14.9 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Editionby Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approachby Bart Baesens, Aimee Backiel, and Seppe vanden Broucke. Wiley.
3. Java Programming with Oracle JDBCby Donald Bales. O'Reilly Media.
4. Java EE 8 Application Developmentby David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applicationsby Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBCby Gregory Brill. Sybex.

Dr. U. Surya Kameswari

LESSON- 15

HANDLING REQUESTS, SESSIONS, AND SECURITY IN SERVLETS

Aim and Objectives:

- Reading request and initialization parameters
- Generating HTTP responses
- Using cookies and session tracking
- Introduction to security concerns in web applications

Structure:

15.1 Handling Http Request & Responses

1. Introduction

In web applications, HTTP (HyperText Transfer Protocol) is the foundation of communication between a client (like a web browser) and a server. When a client sends an HTTP request to a server, the server processes it and sends back an HTTP response.

Java provides several APIs and frameworks to handle HTTP requests and responses, most commonly through Servlets and JSP (JavaServer Pages).

2. HTTP Request

An HTTP request is sent by a client to request data or perform an action on the server.
It consists of:

- **Request Line** (method, URL, version)
- **Headers** (metadata like Content-Type, User-Agent, etc.)
- **Body** (optional; contains data for POST/PUT methods)

Common HTTP Methods:

Method	Description
GET	Requests data from the server
POST	Sends data to the server for processing
PUT	Updates an existing resource
DELETE	Deletes a resource
HEAD	Retrieves headers only
OPTIONS	Describes communication options

3. HTTP Response

The HTTP response is sent by the server to the client after processing the request.
It includes:

- **Status Line** (protocol, status code, message)
- **Headers** (metadata such as content type, length, etc.)
- **Body** (data sent to the client)

Common HTTP Status Codes:

Code	Message	Description
200	OK	Request successful
404	Not Found	Resource not found
500	Internal Server Error	Server encountered an error
302	Found	Redirection
403	Forbidden	Access denied

4. Handling HTTP Requests and Responses using Servlets

Servlets are Java programs that run on a server and handle requests/responses dynamically.

Example 1: Handling GET Request

File: GetExampleServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class GetExampleServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String name = request.getParameter("username");

out.println("<html><body>");
out.println("<h2>Welcome, " + name + "!</h2>");
out.println("<p>This is a GET request example.</p>");
out.println("</body></html>");
   }
}
```

HTML Form:

```
<!DOCTYPE html>
<html>
<body>
<form action="GetExampleServlet" method="get">
   Enter your name: <input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Input:

username = ABC

Output (Browser):

Welcome, ABC!
This is a GET request example.

Example 2: Handling POST Request

File: PostExampleServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class PostExampleServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
```

```
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

        String user = request.getParameter("user");
        String email = request.getParameter("email");

out.println("<html><body>");
out.println("<h3>User Registration Details</h3>");
out.println("<p>Username: " + user + "</p>");
out.println("<p>Email: " + email + "</p>");
out.println("</body></html>");
    }
}
```

HTML Form:

```
<!DOCTYPE html>
<html>
<body>
<form action="PostExampleServlet" method="post">
    Username: <input type="text" name="user"><br>
    Email: <input type="text" name="email"><br>
<input type="submit" value="Register">
</form>
</body>
</html>
```

Input:

user = Rani
email = rani@example.com

Output (Browser):

User Registration Details
Username: Rani
Email: rani@example.com

5. Accessing Request Data

You can access information from the request using the following methods:

Method	Description
getParameter(String name)	Returns the value of a form parameter
getHeader(String name)	Returns the value of a request header
getMethod()	Returns the HTTP method (GET/POST)
getRequestURI()	Returns the requested URI
getRemoteAddr()	Returns the IP address of the client

6. Setting Response Data

You can control what the server sends back to the client using:

Method	Description
setContentType(String type)	Sets response MIME type
setStatus(intsc)	Sets the HTTP status code
addHeader(String name, String value)	Adds a header to the response
getWriter()	Returns a writer to output response text

7. Example: Using Both GET and POST

File: RequestResponseServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class RequestResponseServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
processRequest(request, response, "GET");
   }

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
processRequest(request, response, "POST");
   }

private void processRequest(HttpServletRequest request, HttpServletResponse response,
String method)
throwsIOException {
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
        String name = request.getParameter("name");

out.println("<html><body>");
out.println("<h3>Request Method: " + method + "</h3>");
out.println("<p>Hello, " + name + "!</p>");
out.println("</body></html>");
    }
}
```

HTML Form:

```
<!DOCTYPE html>
<html>
<body>
<form action="RequestResponseServlet" method="post">
   Name: <input type="text" name="name"><br>
<input type="submit" value="Send Request">
</form>
</body>
</html>
```

Input:

name = ABC123

Output:

Request Method: POST
Hello, ABC123!

8. Table

Concept	Description
HTTP Request	Data sent from client to server
HTTP Response	Data sent from server to client
GET method	Used to retrieve data
POST method	Used to send data securely
Servlet	Java class to handle request/response dynamically
Response writer	Used to generate HTML or data back to client

15.2 USING COOKIES-SESSION TRACKING

1. Introduction

In web technologies, HTTP is a stateless protocol, meaning the server does not remember any informationabout users between requests.To maintain information (like login details, user preferences, or shopping cart items) across multiple requests, SessionTracking is used.

2. What is Session Tracking?

Session Tracking is the process of maintaining user state and data across multiple requests between a client (browser) and a server.

When a user visits a website, a session begins. During this session, the server can store user-specific information and recall it in subsequent interactions.

3. Need for Session Tracking

HTTP is stateless, so:

- Each request from the browser is treated as independent.
- The server cannot identify whether two requests came from the same user.

To overcome this, web applications use session tracking mechanisms to identify returningusers and retain information during their visit.

4. Session Tracking Techniques

Technique	Description
Cookies	Information stored on the client side (browser)
Hidden Form Fields	Hidden input fields within HTML forms
URL Rewriting	Adding session data in the URL
HttpSession	Server-side session object managed by servlet container

Part A: Using Cookies

5. What is a Cookie?

A Cookie is a small piece of text data stored by the browser on the client'scomputer.
It helps the server recognize users during subsequent requests.

Key Properties of Cookies:

- Stored as key–value pairs.
- Sent automatically with each request to the same domain.
- Can have an expiration time (persistent or session cookies).

6. Types of Cookies

Type	Description
Session Cookie	Temporary; deleted when browser closes
Persistent Cookie	Stored on disk until expiry date or manually deleted

7. Example 1 – Creating and Reading Cookies

HTML Form: cookie_form.html

```
<!DOCTYPE html>
<html>
<head>
<title>Cookie Example</title>
</head>
<body>
<h3>Enter Your Name</h3>
<form action="SetCookieServlet" method="post">
      Name: <input type="text" name="username"><br><br>
<input type="submit" value="Set Cookie">
</form>
</body>
</html>
```

Servlet 1: SetCookieServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class SetCookieServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String name = request.getParameter("username");
      Cookie cookie = new Cookie("user", name);
response.addCookie(cookie);

out.println("<html><body>");
out.println("<h3>Cookie has been set successfully!</h3>");
out.println("<a href='GetCookieServlet'>Click here to read the cookie</a>");
```

```
out.println("</body></html>");
   }
}
```

Servlet 2: GetCookieServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class GetCookieServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
Cookie[] cookies = request.getCookies();
      String userName = "Guest";
if (cookies != null) {
for (Cookie c :cookies) {
if (c.getName().equals("user")) {
userName = c.getValue();
            }
         }
      }

out.println("<html><body>");
out.println("<h2>Welcome Back, " + userName + "!</h2>");
out.println("</body></html>");
   }
}
```

Sample Input:

Name = ABC

Output:

1.After submitting form:

Cookie has been set successfully!
Click here to read the cookie

2.After clicking the link:

Welcome Back, ABC!

8. Cookie Methods

Method	Description
Cookie(String name, String value)	Creates a cookie
getName()	Returns cookie name
getValue()	Returns cookie value
setMaxAge(int expiry)	Sets cookie lifetime (in seconds)
response.addCookie(Cookie c)	Adds cookie to response
request.getCookies()	Returns cookies sent by client

Part B: Using HttpSession

9. What is an HttpSession?

HttpSession is a server-side object that stores user-specific information. It automatically assigns a unique Session ID to each client.

Unlike cookies, session data is stored on the server, making it more secure.

10. Session Lifecycle

1. Session Created — when a user first accesses the servlet.
2. Data Stored — attributes are added using setAttribute().
3. Data Accessed — using getAttribute().
4. Session Expired — automatically after inactivity (default 30 mins).

11. Example 2 – Using HttpSession

HTML Form: session_form.html

```
<!DOCTYPE html>
<html>
<head>
<title>Session Tracking Example</title>
</head>
<body>
<h3>User Login</h3>
<form action="SessionServlet1" method="post">
      Name: <input type="text" name="username"><br><br>
<input type="submit" value="Login">
</form>
</body>
```

```
</html>
```

Servlet 1: SessionServlet1.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class SessionServlet1 extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
      String name = request.getParameter("username");
HttpSession session = request.getSession();
session.setAttribute("user", name);
out.println("<html><body>");
out.println("<h3>Welcome, " + name + "!</h3>");
out.println("<a href='SessionServlet2'>Go to next page</a>");
out.println("</body></html>");
   }
}
```

Servlet 2: SessionServlet2.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class SessionServlet2 extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
HttpSession session = request.getSession(false); // don't create a new one
      String userName = (String) session.getAttribute("user");
out.println("<html><body>");
out.println("<h2>Hello again, " + userName + "!</h2>");
out.println("<p>Your session is still active.</p>");
out.println("</body></html>");
   }
}
```

Sample Input:

username = Rani

Output:

Page 1:

Welcome, Rani!
Go to next page

Page 2 (after clicking link):

Hello again, Rani!
Your session is still active.

12. HttpSession Methods

Method	Description
request.getSession()	Creates a new session if not exists
getId()	Returns session ID
getCreationTime()	Returns time of creation
setAttribute(String name, Object value)	Stores value in session
getAttribute(String name)	Retrieves stored value
invalidate()	Terminates session

13. Difference Between Cookies and Sessions

Feature	Cookies	HttpSession
Storage	Client-side	Server-side
Security	Less secure	More secure
Capacity	Limited (~4KB)	Larger (server memory)
Lifetime	Depends on expiry	Until session timeout
Usage	Lightweight info	Sensitive user data

14. Example 3 – Using Cookies + Session Together

In real-world applications, cookies often store the Session ID, while the server uses that ID to retrieve session data.

For example:

- Browser stores a cookie named JSESSIONID.
- Server uses this ID to map user data stored in the session.

15. Table

Concept	Description
Cookie	Small client-side storage mechanism
Session	Server-side user tracking mechanism
Session ID	Unique identifier for each user
Session Tracking	Maintaining user data across requests
Servlet API	Provides HttpSession and Cookie classes

16. Best Practices

- Use HttpSession for sensitive data.
- Set appropriate cookie expiration and security flags.
- Invalidate sessions on logout.
- Avoid storing large data in sessions.
- Encrypt cookie values if used for authentication.

17. Output Snapshot

Page 1:

Welcome, ABC!
Cookie has been set successfully!

Page 2:

Welcome Back, ABC!
Your session is still active.

18. Note:

Session tracking through Cookies and HttpSession is vital in building dynamic web applications that remember users and personalize content.
They form the core of state management in web technologies, enabling features like:

- User login systems
- Shopping carts
- Personalized dashboards

15.3 SECURITY ISSUES

1. Introduction

In Web Technologies**,** security refers to the protection of data and resources from unauthorized access, modification, or destruction.Since web applications are accessed over the internet, they are vulnerable to a wide range of security threats.
A secure web application ensures:

- **Confidentiality** – data is accessible only to authorized users
- **Integrity** – data cannot be altered by unauthorized users
- **Availability** – services are always available to legitimate users
- **Authentication** – verifies user identity
- **Authorization** – grants appropriate access based on privileges

2. Why Web Security Is Important

Web applications often handle sensitive data such as:

- Login credentials
- Banking and financial data
- Personal information (email, address, contact numbers)

If not properly protected, attackers can exploit vulnerabilities to:

- Steal data
- Alter website content
- Impersonate users
- Gain administrative control of systems

3. Common Security Threats in Web Applications

Threat	Description	Example
Cross-Site Scripting (XSS)	Inserting malicious scripts into webpages	Attacker injects JavaScript into form fields
SQL Injection	Injecting SQL commands to manipulate databases	OR '1'='1' in login fields
Cross-Site Request Forgery (CSRF)	Forcing a logged-in user to perform unwanted actions	Attacker uses hidden forms to make a user send a request
Session Hijacking	Stealing valid session ID to impersonate users	Capturing cookies to access user account
Broken Authentication	Weak login or password handling	No password encryption or reuse
Data Exposure	Sending sensitive data without encryption	Transmitting credentials over HTTP instead of HTTPS
Denial of Service (DoS)	Overloading a server to crash it	Flooding the server with fake requests

4. Major Security Issues Explained

4.1. SQL Injection

Description:

Occurs when unvalidated input is directly used in SQL queries, allowing attackers to manipulate database operations.

Vulnerable Example:

```
String user = request.getParameter("username");
String pass = request.getParameter("password");

Statement stmt = conn.createStatement();
String query = "SELECT * FROM users WHERE username='" + user + "' AND password='"
+ pass + "'";
ResultSetrs = stmt.executeQuery(query);
```

If input:

username = admin
password = ' OR '1'='1

Then query becomes:

SELECT * FROM users WHERE username='admin' AND password='' OR '1'='1'

This condition always returns **true**, allowing unauthorized access.

Secure Example (Using PreparedStatement):

```
String user = request.getParameter("username");
String pass = request.getParameter("password");
PreparedStatementps = conn.prepareStatement(
   "SELECT * FROM users WHERE username=? AND password=?");
ps.setString(1, user);
ps.setString(2, pass);
ResultSetrs = ps.executeQuery();
```

Prevents SQL Injection by treating user input as data, not code.

Sample Input:

username = admin
password = ' OR '1'='1

Output:

Login failed. Invalid credentials.

4.2. Cross-Site Scripting (XSS)

Description:

XSS allows attackers to inject malicious JavaScript into web pages viewed by other users.

Vulnerable Example:

```
String name = request.getParameter("username");
out.println("<h3>Welcome " + name + "</h3>");
```

Input:

```
<script>alert('Hacked!');</script>
```

Output on browser:

- A popup appears: Hacked!

Secure Example:

Use HTML encoding to sanitize user input.

```
String name = request.getParameter("username");
name = name.replaceAll("<", "&lt;").replaceAll(">", "&gt;");
out.println("<h3>Welcome " + name + "</h3>");
```

Input:

```
<script>alert('Hacked!');</script>
```

Output:

Welcome <script>alert('Hacked!');</script>

(Script is displayed as text, not executed)

4.3. Session Hijacking

Description:

Session Hijacking occurs when an attacker steals a valid session ID (stored in cookies) to impersonate a legitimate user.

Prevention Techniques:

- Always use HTTPS
- Regenerate SessionID on login
- Set cookies as HttpOnly and Secure
- Invalidate sessions on logout

Example:

```
HttpSession session = request.getSession();
session.setAttribute("user", username);
// Set secure cookie
Cookie c = new Cookie("JSESSIONID", session.getId());
c.setHttpOnly(true);
c.setSecure(true);
response.addCookie(c);
```

4.4. Cross-Site Request Forgery (CSRF)

Description:

An attacker tricks a logged-in user into performing actions they didn't intend (e.g., submitting a form).

Prevention:

- Use a CSRF token with each form submission.

Example:

In form:

```
<form action="TransferServlet" method="post">
<input type="hidden" name="csrfToken" value="${token}">
   Amount: <input type="text" name="amount">
<input type="submit" value="Transfer">
</form>
```

In servlet:

```
String token = (String) session.getAttribute("csrfToken");
String formToken = request.getParameter("csrfToken");
if (token != null &&token.equals(formToken)) {
out.println("Transaction Successful");
} else {
out.println("Security Error: Invalid CSRF Token");
}
```

4.5. Data Encryption

Description:

Sensitive data (like passwords) should never be stored or transmitted as plain text**.**

Example:

```
importjava.security.MessageDigest;
```

```
String password = request.getParameter("password");
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] hash = md.digest(password.getBytes());
String encrypted = new String(hash);
```

Output (hashed password):

E99A18C428CB38D5F260853678922E03ABD8334A

5. Example Program – Secure Login Using PreparedStatement and Session

HTML Form (login.html)

```
<!DOCTYPE html>
<html>
<head><title>Secure Login</title></head>
<body>
<h3>User Login</h3>
<form action="LoginServlet" method="post">
      Username: <input type="text" name="username"><br><br>
      Password: <input type="password" name="password"><br><br>
<input type="submit" value="Login">
</form>
</body>
</html>
```

Servlet: LoginServlet.java

```
import java.io.*;
import java.sql.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class LoginServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String uname = request.getParameter("username");
      String pass = request.getParameter("password");

try {
Class.forName("com.mysql.jdbc.Driver");
         Connection con = DriverManager.getConnection(
           "jdbc:mysql://localhost:3306/webdb", "root", "admin");
```

```
PreparedStatementps = con.prepareStatement(
            "SELECT * FROM users WHERE username=? AND password=?");
ps.setString(1, uname);
ps.setString(2, pass);

ResultSetrs = ps.executeQuery();

if (rs.next()) {
HttpSession session = request.getSession();
session.setAttribute("user", uname);
out.println("<h3>Welcome, " + uname + "!</h3>");
        } else {
out.println("<h3>Invalid username or password</h3>");
        }
con.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Sample Input:

username = Rani
password = rani123

Output:

Welcome, Rani!

If invalid credentials:

Invalid username or password

6. Best Security Practices

- Always use PreparedStatement to prevent SQL Injection.
- Validate and sanitize all userinputs.
- Use HTTPS for secure data transmission.
- Implement strongauthentication (use hashed passwords).
- Regenerate session IDs after login and invalidateon logout**.**
- Use HttpOnly and Secure cookies.
- Apply CSRFtokens in forms.
- Never expose sensitive data in URLs.
- Keep all frameworks and servers up to date**.**
- Use exception handling to avoid leaking system details.

7. Table

Security Issue	Description	Prevention Technique
SQL Injection	Malicious SQL statements	Use PreparedStatement
XSS	JavaScript injection	Sanitize HTML input
CSRF	Unauthorized actions	Use CSRF tokens
Session Hijacking	Stealing session ID	Secure cookies & HTTPS
Data Exposure	Plain text data	Encrypt sensitive data

8. Note:

Security in web applications is not optional — it's essential.
By understanding and implementing these measures, developers can protect users from:

- Data theft
- Unauthorized access
- Financial loss
- System compromise

Secure coding, input validation, and encryption are the pillarsof safe web development**.**

15.4 SUMMARY

Web applications communicate through the HTTP protocol, where clients send requests and servers respond with appropriate data. In Java, this interaction is primarily managed using Servlets and JSP (JavaServer Pages). Common HTTP methods include GET for retrieving information and POST for securely sending data to the server. Each server response contains an HTTP status code, such as 200 (OK) for successful operations or 404 (Not Found) for missing resources. Because HTTP is a stateless protocol, it does not retain user information between requests. To overcome this, web applications use session tracking mechanisms like Cookies and HttpSession. Cookies store small amounts of data on the client side, while HttpSession securely maintains user details on the server, enabling features like user authentication, personalized dashboards, and shopping carts.

Security is a critical concern in web applications, as they are vulnerable to threats such as SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and Session Hijacking. These attacks can compromise data integrity, steal credentials, or execute malicious scripts. To mitigate such risks, developers use PreparedStatements to prevent SQL injection, perform input validation to filter harmful data, and implement CSRF tokens and secure cookies to protect session data. Using HTTPS for encrypted communication, regenerating session IDs after login, and properly invalidating sessions after logout further strengthen security.

By following best practices—such as sanitizing user inputs, employing secure transport protocols, and enforcing proper authentication and authorization—developers can ensure confidentiality, integrity, and availability of web applications. A secure and state-aware web

design not only enhances user trust but also protects sensitive information from unauthorized access and data breaches, forming the foundation of reliable and professional web systems.

15.5 KEY TERMS

HTTP (HyperText Transfer Protocol), Servlet, HTTP Request, HTTP Response, GET Method, POST Method, Session Tracking, Cookie, HttpSession, PreparedStatement, SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Session Hijacking, Encryption

15.6 SELF-ASSESSMENT QUESTIONS

1. What does HTTP stand for?
2. What is the role of a Servlet in a web application?
3. Which HTTP method is used to retrievedata from the server?
4. Which HTTP method is used to senddata to the server?
5. What is a Cookie used for?
6. What is an HttpSession?
7. Why do we need sessiontracking in web applications?
8. What is SQLInjection?
9. What is the purpose of encryption in web security?
10. Name one way to prevent Cross-Site Scripting (XSS) attacks.

15.7 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition by Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approach by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.
3. Java Programming with Oracle JDBC by Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development by David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applications by Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBC by Gregory Brill. Sybex.

Dr. U. Surya Kameswari

LESSON- 16

JSP FUNDAMENTALS AND MVC ARCHITECTURE

Aim and Objectives:

- Identify the problems and limitations of Servlets that led to the introduction of JSP.
- Explain the structure and components of a JSP page and their purposes.
- Describe the JSP processing mechanismand how a JSP request is handled by the server.
- Understand how to design JSP applications using the MVC architecture for better separation of concerns.
- Learn to set up the JSP environmentby installing the Java SDK, configuring the Tomcat server, and testing the setup.

STRUCTURE:

16.1 INTRODUCTION TO JSP

What is JSP?

JSP (JavaServer Pages) is a server-side technology used to create dynamic web content. It allows developers to embed Java code directly into HTML pages.

JSP is built on top of Servlet technology, but it provides a simpler way to develop web applications by separating the presentation layer (HTML) from the business logic (Java code).

Why JSP? (The Problem with Servlets)

Servlets require writing a lot of Java code to generate HTML responses.
For example, to display a simple message using a servlet, you must write:

```
out.println("<html>");
```

```
out.println("<body>");
out.println("<h1>Welcome to Java Servlets</h1>");
out.println("</body>");
out.println("</html>");
```

This approach mixes Java code and HTML, making maintenance difficult.
JSP solves this problem by allowing you to write HTML normally and embed Java logic only where needed.

Anatomy of a JSP Page

A JSP page typically has the .jsp extension and can include:

Element Type	Syntax	Description
Directive	<%@ directive %>	Defines page settings (e.g., importing classes).
Declaration	<%! code %>	Declares variables or methods.
Scriptlet	<% code %>	Contains Java code executed at request time.
Expression	<%= expression %>	Outputs the result of a Java expression.
Comments	<%-- comment --%>	JSP comment (not sent to client).

How JSP Works (Processing)

When a JSP page is requested:

1. The web server (e.g., Tomcat) converts the JSP file into a Servlet.
2. The servlet is compiled into bytecode.
3. The compiled servlet runs and produces HTML output.
4. The HTML is sent to the client's browser.

Setting Up JSP Environment

1. **Install Java Development Kit (JDK)**
   - Download and install from Oracle.
   - Set environment variable: JAVA_HOME.
2. **Install Apache Tomcat Server**
   - Download from https://tomcat.apache.org.
   - Unzip and configure the path.
3. **Deploy JSP File**
   - Place your .jsp file inside apache-tomcat/webapps/ROOT/.
   - Start the Tomcat server using startup.bat (Windows) or startup.sh (Linux).
   - Access it via:

     👉http://localhost:8080/filename.jsp

Example 1: Simple JSP Program

File: hello.jsp

```
<html>
<head><title>Welcome Page</title></head>
<body>
<h2>Welcome to JSP!</h2>
<%
      String name = "Student";
out.println("<p>Hello, " + name + "! This is your first JSP program.</p>");
   %>
</body>
</html>
```

Output in Browser:

Welcome to JSP!
Hello, Student! This is your first JSP program.

Example 2: JSP with User Input

File: input.jsp

```
<html>
<head><title>User Input Example</title></head>
<body>
<form action="display.jsp" method="post">
      Enter your name: <input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

File: display.jsp

```
<html>
<head><title>Display Page</title></head>
<body>
<%
      String user = request.getParameter("username");
      if (user == null || user.trim().equals("")) {
out.println("<h3>Please enter a valid name.</h3>");
      } else {
out.println("<h3>Hello, " + user + "! Welcome to JSP programming.</h3>");
```

```
    }
  %>
</body>
</html>
```

Input:

User enters: John

Output:

Hello, John! Welcome to JSP programming.

16.2 THE PROBLEM WITH SERVLET

1. Introduction:-Servlets are Java programs that run on a web server and dynamically generate web pages. They handle client requests and responses using Java code.

However, when building large web applications, Servlets become difficult to maintain, hard to read, and time-consuming for web designers — mainly because they mix HTML and Java logic together.

2. Typical Servlet Workflow

1. The client (browser) sends a request to the server.
2. The server calls the Servlet, which contains Java code.
3. The Servlet generates HTML output using Java code.
4. The generated HTML is sent back to the client.

3. The Core Problem

While Servlets are powerful, they have some serious drawbacks when used to create complex web pages.

Main Problems with Servlets:

Problem	Description
1. Mixed Code (HTML + Java)	Writing HTML inside Java out.println() statements makes the code long and messy.
2. Poor Readability	Hard for web designers to edit HTML within Java source code.
3. Maintenance Difficulty	Any small change in the webpage design requires Java recompilation.
4. No Clear Separation	Servlets combine both business logic and presentation logic.
5. Slower Development	Requires both Java and HTML knowledge for every modification.

4. Example: Servlet Generating HTML Output

File: HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
response.setContentType("text/html");
      PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head><title>Servlet Example</title></head>");
out.println("<body>");
out.println("<h2>Welcome to Java Servlets!</h2>");
out.println("<p>This page is generated using Java code.</p>");
out.println("</body>");
out.println("</html>");
out.close();
   }
}
```

Web Deployment Descriptor (web.xml):

```
<web-app>
<servlet>
<servlet-name>hello</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>hello</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

Browser Request (Input):

http://localhost:8080/YourApp/hello

Output (in Browser):

Welcome to Java Servlets!
This page is generated using Java code.

It works fine — but the HTML is buried inside Java code!

5. Why Is This a Problem?

Imagine the designer wants to change the HTML layout, color, or font.

They would need to open the Java file, find the out.println() lines, and modify the HTML — then recompile and redeploy the servlet.

That's inefficient and error-prone.

6. Overview-The below is a conceptual flowdiagram showing how Servlets mix business and presentation logic:

```
+---------------------------------------+
|         Java Servlet (HelloServlet)   |
|---------------------------------------|
| Business Logic (Java Code)            |
| out.println("<html>...");             |
| out.println("<h1>Hello User</h1>");    |
| out.println("</html>");               |
+---------------------------------------+
          ↓
     Compiled & Executed
          ↓
    HTML Response Sent to Browser
```

Problem: The same file handles both application logic and HTML layout, which is not modular.

7. Comparison: Servlet vs JSP

Feature	Servlet	JSP
Code Type	Java code with embedded HTML	HTML with embedded Java
Ease of Design	Difficult for web designers	Easier for web designers
Maintenance	Requires recompilation	JSP auto-compiles
Use Case	Good for backend logic	Good for presentation layer

8. Example of JSP Solution

Let's see how JSP fixes the same problem:

File: hello.jsp

```
<html>
<head><title>JSP Example</title></head>
<body>
<h2>Welcome to JSP!</h2>
<p>This page is generated using JSP — HTML is clean and easy to read.</p>
</body>
</html>
```

Output:

Welcome to JSP!
This page is generated using JSP — HTML is clean and easy to read.

No messy out.println() statements — only simple HTML!

9. Servlet Problems

Issue	Explanation
Mixing of HTML & Java	Reduces readability and maintainability.
Difficult Design Changes	Designers can't edit HTML without Java knowledge.
No MVC Separation	Business logic and view are tightly coupled.
Compilation Overhead	Every design change requires recompiling the servlet.

16.3 THE ANATOMY OF A JSP PAGE

A **JSP (JavaServer Page)** is essentially an HTML page enhanced with Java code. It enables developers to create dynamic web content by embedding Java inside HTML tags.

Each JSP file is eventually translated into a Servlet by the web server (e.g., Apache Tomcat). This allows the Java code to be executed on the server before sending the HTML result to the client.

2. Basic Structure of a JSP Page

Here's what a simple JSP file looks like:

```
<%@ page language="java" contentType="text/html" pageEncoding="UTF-8"%>
<html>
<head><title>My First JSP Page</title></head>
<body>
<h2>Welcome to JSP!</h2>
<%
    // Scriptlet: Java code inside JSP
```

```
        String name = "Student";
out.println("<p>Hello, " + name + "!</p>");
   %>
</body>
</html>
```

Output in Browser:

Welcome to JSP!

Hello, Student!

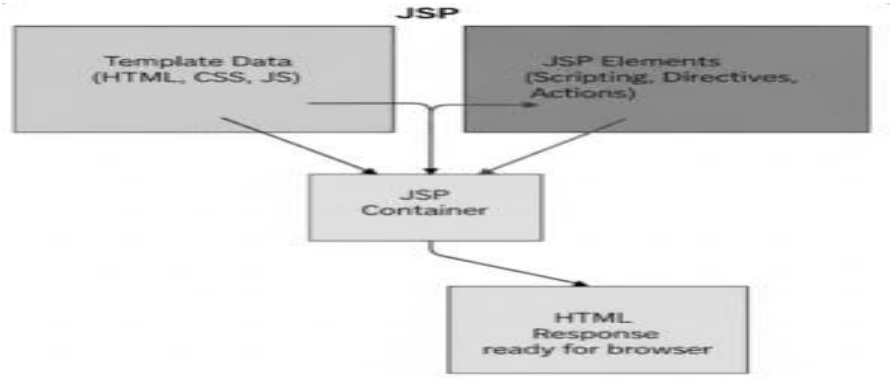


Figure: Anatomy of a JSP Page

3. Major Components (Anatomy) of a JSP Page

A JSP page can contain the following elements:

Element Type	Syntax	Description
1. Directive	<%@ ... %>	Gives instructions to the JSP engine (e.g., importing packages, setting content type).
2. Declaration	<%! ... %>	Declares variables and methods that become part of the servlet class.
3. Scriptlet	<% ... %>	Contains Java code executed at request time.
4. Expression	<%= ... %>	Inserts the result of a Java expression directly into the HTML output.
5. Comment	<%-- ... --%>	JSP comment (not visible in the HTML sent to client).
6. Action Tags	<jsp:... />	Invokes built-in JSP actions (like forwarding requests, using beans, etc.).

4. JSP Directives

Syntax:

<%@ directive attribute="value" %>

Common Directives:

- **page** – Defines page-level settings (language, import, content type).
- **include** – Includes another file at translation time.
- **taglib** – Declares a tag library.

Example:

```
<%@ page import="java.util.Date" %>
<html>
<body>
<h3>Current Date and Time:</h3>
<%
   Date today = new Date();
out.println(today);
%>
</body>
</html>
```

Output:

Current Date and Time:
Wed Oct 29 12:30:45 IST 2025

5. JSP Declarations

Syntax:

<%! code %>

Used to declare variables or methods that will become instance members of the servlet class.

Example:

```
<%!
   int counter = 0;
   int incrementCounter() {
      return ++counter;
   }
%>
<html>
<body>
<p>Counter value: <%= incrementCounter() %></p>
</body>
</html>
```

Output (each refresh increases count):

Counter value: 1
Counter value: 2
Counter value: 3
...

6. JSP Scriptlets

Syntax:

<% code %>

Contains Java statements that execute every time the page is requested.

Example:

```
<html>
<body>
<%
   for (int i = 1; i <= 3; i++) {
out.println("<p>Number: " + i + "</p>");
   }
%>
</body>
</html>
```

Output:

Number: 1
Number: 2
Number: 3

7. JSP Expressions

Syntax:

<%= expression %>

Displays the result of a Java expression directly in the output stream (like out.println() shortcut).

Example:

```
<html>
```

```
<body>
<h3>The sum of 10 and 20 is: <%= 10 + 20 %></h3>
</body>
</html>
```

Output:

The sum of 10 and 20 is: 30

8. JSP Comments

Syntax:

```
<%-- comment text --%>
```

Unlike HTML comments, JSP comments are not sent to the client — they are completely hidden.

Example:

```
<html>
<body>
<%-- This is a JSP comment, it will not appear in browser source code --%>
<p>This is visible text.</p>
</body>
</html>
```

Output:

This is visible text.

(The JSP comment will not appear in “View Source.”)

9. JSP Action Tags

Syntax:

```
<jsp:action attribute="value" />
```

Used to control the behavior of the JSP engine — like forwarding requests or using JavaBeans.

Example 1: jsp:forward

```
<jsp:forward page="nextpage.jsp" />
```

Redirects the user to another JSP page.

Example 2: jsp:include

```
<jsp:include page="header.jsp" />
```

Includes the contents of header.jsp in the current page.

10. Complete Example: Combining All Elements

File: anatomyExample.jsp

```
<%@ page import="java.util.*" %>
<%! int counter = 0; %>
<html>
<head><title>JSP Anatomy Example</title></head>
<body>
<%-- This is a JSP comment, not visible in browser --%>
<h2>Understanding JSP Components</h2>
<%
   counter++;
   String user = "Alice";
   Date now = new Date();
%>
<p>Hello, <%= user %>!</p>
<p>You are visitor number <%= counter %>.</p>
<p>Current date and time: <%= now %></p>
<jsp:include page="footer.jsp" />
</body>
</html>
```

File: footer.jsp

```
<hr>
<p>Thank you for visiting our JSP demo page.</p>
```

Output in Browser:

Understanding JSP Components
Hello, Alice!
You are visitor number 1.
Current date and time: Wed Oct 29 13:05:10 IST 2025

Thank you for visiting our JSP demo page.

11. Table

Component	Syntax	Purpose	Executes When
Directive	<%@ ... %>	Page instructions	At translation time
Declaration	<%! ... %>	Define fields/methods	Once per class
Scriptlet	<% ... %>	Execute Java code	On every request
Expression	<%= ... %>	Output value	On every request
Comment	<%-- ... --%>	Hidden note	Never executed
Action Tag	<jsp:... />	JSP behavior control	At request time

16.4JSP PROCESSING

1. JSP (JavaServer Pages) is a server-side technology that allows you to create dynamic web content by embedding Java code within HTML.
However, unlike plain HTML pages, JSPs are compiled and executed on the server before the output (HTML) is sent to the client browser.The server handles the processing of a JSP file in several stages — converting it into a Servlet, compiling it, and executing it to produce HTML output.

2. JSP Processing Overview

When a client (like a web browser) requests a JSP page, theweb container (e.g., Apache Tomcat) performs the following steps:

1. **Translation Phase:**
   The JSP file is translated into a Java Servlet source file.
2. **Compilation Phase:**
   The generated Servlet source is compiled into a .class file (bytecode).
3. **Loading & Initialization Phase:**
   The Servlet class is loaded into memory and initialized.
4. **Request Processing Phase:**
   The Servlet's service() method executes, processing client requests and generating responses.
5. **Response Phase:**
   The output (HTML) is sent back to the client browser.
6. **Destruction Phase:**
   When the application is stopped, the JSP Servlet is destroyed.

4. JSP Processing Flow Diagram

The complete JSP processing sequence creates a smooth transition from human-readable .jsp code to machine-executable bytecode, ultimately producing a dynamic web response.

This powerful lifecycle serves as the foundation for all dynamic web applications developed using JSP. As shown in below figure.

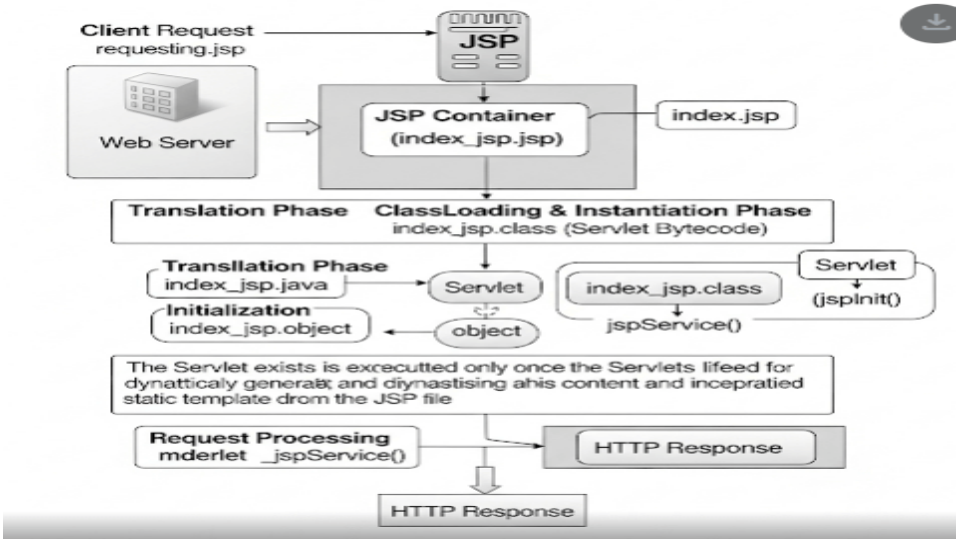


Figure: JSP Processing Flow Diagram

4. JSP Lifecycle Methods

Internally, a JSP page is converted into a Servlet, and the Servlet follows a defined lifecyclemanaged by the JSP container**.**

JSP Lifecycle Method	Description
jspInit()	Called once when the JSP is initialized.
_jspService()	Called for each client request (contains the main logic).
jspDestroy()	Called before the JSP is destroyed.

5. Step-by-Step JSP Processing Example

Let’s demonstrate JSP processing using a real example.

Example Files:

File 1:welcome.jsp
File 2:web.xml (deployment descriptor)

File: welcome.jsp

<%@ page language="java" contentType="text/html" pageEncoding="UTF-8"%>

```
<html>
<head><title>JSP Processing Example</title></head>
<body>
<%-- JSP comment: This will not appear in browser --%>
<%!
   int visitCount = 0;
   // This method runs once when JSP is initialized
   public void jspInit() {
      System.out.println("JSP Initialized...");
   }
   // This method runs when the JSP is destroyed
   public void jspDestroy() {
      System.out.println("JSP Destroyed...");
   }
%>
<%
   // This part executes for each request
   visitCount++;
   String user = request.getParameter("name");
   if (user == null || user.equals("")) {
      user = "Guest";
   }
%>
<h2>Welcome to JSP Processing!</h2>
<p>Hello, <%= user %>!</p>
<p>You are visitor number: <%= visitCount %></p>
<form action="welcome.jsp" method="post">
   Enter your name: <input type="text" name="name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Step-by-Step JSP Processing Flow:

Step 1 – Client Request

The browser requests:

http://localhost:8080/JSPApp/welcome.jsp

Step 2 – Translation Phase

The server (Tomcat) translates welcome.jsp → welcome_jsp.java
(This is an automatically generated servlet.)

Step 3 – Compilation

welcome_jsp.java → compiled into welcome_jsp.class

Step 4 – Initialization

The container calls:

jspInit();

This runs only once (prints "JSP Initialized..." on the server console).

Step 5 – Request Handling

The container calls:

_jspService(request, response);

This method executes the main JSP code — reading parameters, generating output, etc.

Step 6 – Response Generation

The generated HTML is sent back to the client browser.

Step 7 – Destruction

When the JSP is reloaded or the server stops, the container calls:

jspDestroy();

6. Example Input and Output

Input 1 (First Visit):

User opens the page without entering a name.

URL:

http://localhost:8080/JSPApp/welcome.jsp

Output 1:

Welcome to JSP Processing!
Hello, Guest!
You are visitor number: 1
[Text box to enter name]

Input 2 (Second Visit with Name):

User enters “Alice” in the text box and submits.

Output 2:

Welcome to JSP Processing!
Hello, Alice!
You are visitor number: 2
[Text box again for new input]

Server Console Output:

JSP Initialized...
JSP Destroyed... (when server stops or reloads)

7. Generated Servlet Code (Simplified View)

When Tomcat processes your JSP, it internally generates something like this:

```
public final class welcome_jsp extends HttpJspBase {
   int visitCount = 0;
   public void jspInit() {
      System.out.println("JSP Initialized...");
   }
   public void _jspService(HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {
      visitCount++;
      String user = request.getParameter("name");
      if (user == null) user = "Guest";
      JspWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h2>Welcome to JSP Processing!</h2>");
out.println("<p>Hello, " + user + "!</p>");
out.println("<p>You are visitor number: " + visitCount + "</p>");
out.println("</body></html>");
   }
   public void jspDestroy() {
      System.out.println("JSP Destroyed...");
   }
}
```

This shows how a JSP page internally becomes a servlet that executes Java code to produce HTML output.

7. Visual Representation of JSP Lifecycle

The life cycle of a JavaServer Page (JSP) involves several stages, beginning with its creation, followed by its translation into a servlet, and ultimately governed by the servlet

lifecycle. This entire process is automatically managed by the JSP engine.Asshown in below figure.

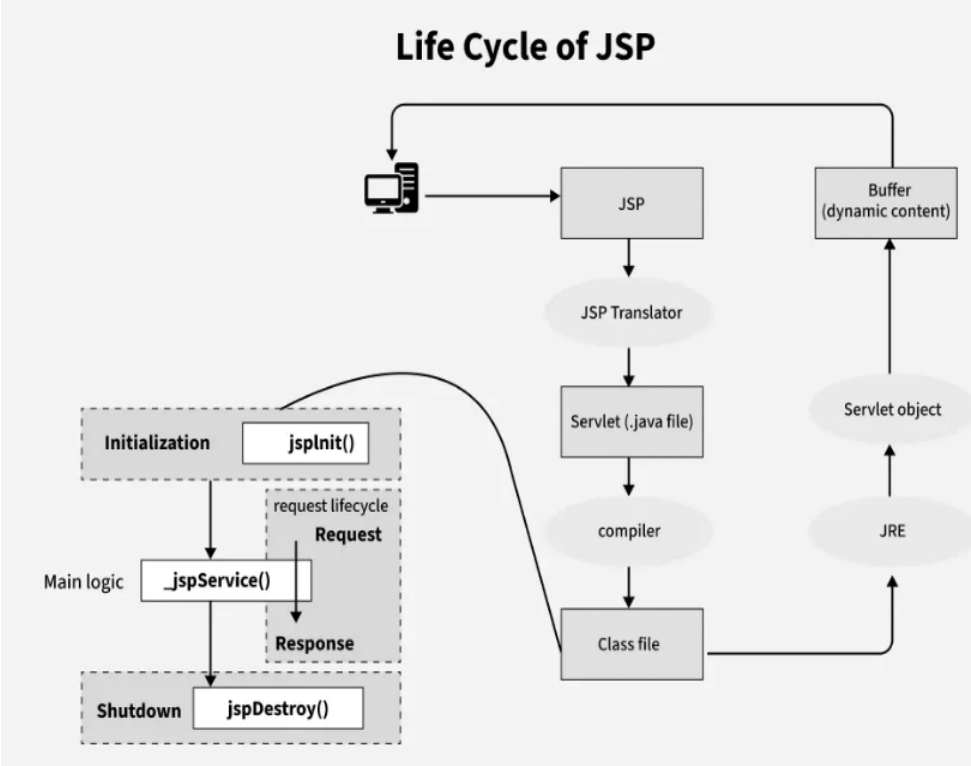


Figure: Life Cycle of JSP

Steps in the JSP Life Cycle:

1. **Translation:** The JSP page is translated into a corresponding servlet.
2. **Compilation:** The translated JSP file (e.g., test.java) is compiled into a class file (test.class).
3. **Class Loading:** The generated servlet class is loaded into the container.
4. **Instantiation:** An instance of the generated servlet class is created.
5. **Initialization:** The container invokes the jspInit() method to initialize the servlet.
6. **Request Processing:** The container calls the _jspService() method to handle client requests and generate responses.
7. **Cleanup:** The jspDestroy() method is invoked by the container to release resources before the JSP is unloaded.

8. Advantages of JSP Processing

JSP pages are compiled only once, not every time they are requested.
Subsequent requests are faster since the compiled servlet is reused.
Clear separation of presentation (HTML) and business logic (Java)**.**
Automatic servlet generation— developers focus on web design, not boilerplate Java code.

9. Table

Phase	Action	Description
Translation	JSP → Servlet	JSP converted into Java source code
Compilation	Servlet → Class	Compiled into bytecode
Initialization	jspInit()	Runs once when JSP loads
Request Handling	_jspService()	Runs for every request
Destruction	jspDestroy()	Runs when JSP unloads

16.5 JSP APPLICATION DESIGN WITH MVC SETTING UP AND JSP ENVIRONMENT

What is MVC?

MVC (Model–View–Controller) is a design pattern used in web development to separate an application into three logical layers:

Component	Description	Technology Used
Model	Contains business logic and data handling.	JavaBeans / POJO classes
View	Handles presentation and user interface.	JSP pages
Controller	Controls the flow of data between Model and View.	Servlets

This separation improves code organization, reusability, and maintenance.

How MVC Works in JSP Applications

1. Client (Browser) sends a request (e.g., submits a form).
2. The Controller (Servlet) receives the request and processes it.
3. The Controller interacts with the Model (Java class/Bean**)** for data or business logic.
4. The result from the Model is sent back to the Controller.
5. The Controller forwards the data to a View (JSP page) for displaying output.
6. The JSP renders the final HTML response to the browser.

MVC Flow Diagram

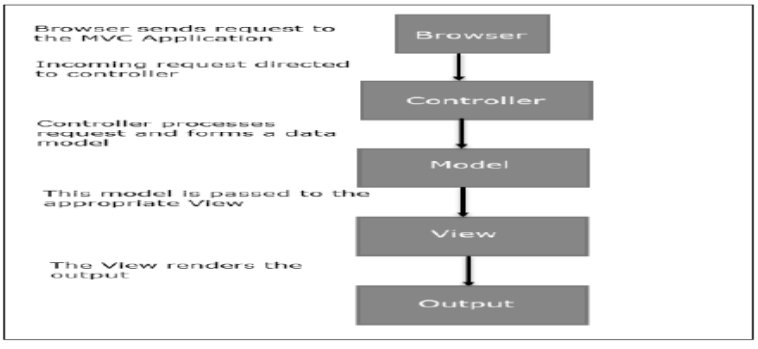


Figure: MVC Flow Diagram.

2. Example Program: JSP Application using MVC

Let's build a small MVC example where the user enters their name, and the application displays a welcome message.

Step 1 – Model (JavaBean)

File: User.java

```
package mvcapp;
public class User {
   private String name;
   public User() {}
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String greetUser() {
      return "Welcome, " + name + "! You are learning JSP with MVC.";
   }
}
```

Step 2 – Controller (Servlet)

File: UserServlet.java

```
package mvcapp;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class UserServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      // Get data from the form
      String username = request.getParameter("username");
      // Create model object
      User user = new User();
user.setName(username);
      // Store model data in request scope
request.setAttribute("userData", user);
```

```
        // Forward data to the view (JSP)
        RequestDispatcher rd = request.getRequestDispatcher("welcome.jsp");
        rd.forward(request, response);
    }
}
```

Step 3 – View (JSP Page)

File: welcome.jsp

```
<%@ page import="mvcapp.User" %>
<html>
<head><title>Welcome Page</title></head>
<body>
<%
        User user = (User) request.getAttribute("userData");
    %>
<h2><%= user.greetUser() %></h2>
</body>
</html>
```

Step 4 – HTML Form (Input Page)

File: index.html

```
<html>
<head><title>JSP MVC Example</title></head>
<body>
<h2>Enter Your Name</h2>
<form action="UserServlet" method="post">
<input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Step 5 – Web Deployment Descriptor

File: web.xml

```
<web-app>
<servlet>
<servlet-name>userServlet</servlet-name>
<servlet-class>mvcapp.UserServlet</servlet-class>
</servlet>
```

<servlet-mapping>
<servlet-name>userServlet</servlet-name>
<url-pattern>/UserServlet</url-pattern>
</servlet-mapping>
</web-app>

Input / Output

Input (from index.html):

User enters:

John

Output (in Browser – welcome.jsp):

Welcome, John! You are learning JSP with MVC.

Advantages of MVC with JSP

- Clear separation of presentation (JSP), logic (Servlet), and data (Model).
- Easier maintenance and updates.
- Designers and developers can work independently.
- Promotes reusability and scalability.

3. Setting Up JSP Environment

To execute JSP programs, you need a proper Java web environment with the JDK and Tomcat server.

Step 1: Install Java Development Kit (JDK)

- Download from: https://www.oracle.com/java/technologies/javase-downloads.html
- Install and set environment variables:
- JAVA_HOME = C:\Program Files\Java\jdk-<version>
- PATH = %JAVA_HOME%\bin
- Verify installation:
- java -version

Step 2: Install Apache Tomcat Server

- Download from: https://tomcat.apache.org
- Extract it to: C:\Tomcat
- Set environment variable:
- CATALINA_HOME = C:\Tomcat
- Start Tomcat:

- C:\Tomcat\bin\startup.bat
- Test in browser:
- http://localhost:8080/

You should see the Tomcat welcome page.

Step 3: Deploy JSP Application

1. Inside C:\Tomcat\webapps, create a folder named mvcapp.
2. Inside it, create WEB-INF folder and place:
    - web.xml
    - classes folder (for .class files)
3. Place index.html and welcome.jsp in the mvcapp root folder.
4. Compile Java files (User.java, UserServlet.java) and place .class files inside:
5. C:\Tomcat\webapps\mvcapp\WEB-INF\classes\mvcapp\

Step 4: Test the Application

Open your browser and go to:

http://localhost:8080/mvcapp/index.html

Input:

Name: Alice

Output:

Welcome, Alice! You are learning JSP with MVC.

4. Table

Step	Component	File	Purpose
1	Model	User.java	Holds data & logic
2	Controller	UserServlet.java	Processes requests
3	View	welcome.jsp	Displays output
4	Configuration	web.xml	Maps servlet
5	Deployment	Tomcat Server	Runs JSP & Servlets

16.6 INSTALLING THE JAVA SOFTWARE DEVELOPMENT KIT

1. What is JDK?

JDK (Java Development Kit) is a software package that provides all the tools required to develop and run Java applications, including:

- JVM (Java Virtual Machine) – executes Java bytecode
- JRE (Java Runtime Environment) – runtime environment for running Java programs
- Compiler (javac) – converts Java source code into bytecode
- Development tools – java, javac, jar, javadoc, etc.

2. Steps to Install JDK

Step 1: Download JDK

- Visit the official Oracle website: Java SE Downloads
- Select the latest Java SE Development Kit and download for your operating system (Windows, Mac, or Linux).

Step 2: Install JDK

- Run the downloaded installer.
- Follow instructions and choose an installation directory (e.g., C:\Program Files\Java\jdk-21).
- Click **Next** and finish the installation.

Step 3: Set Environment Variables (Windows)

1. Open System Properties → Advanced → Environment Variables
2. Add a new system variable:
3. Variable name: JAVA_HOME
4. Variable value: C:\Program Files\Java\jdk-21
5. Update the PATH variable:
6. %JAVA_HOME%\bin
7. Open Command Prompt and check installation:
8. java -version
9. javac -version

Example Output:

```
java version "21"
Java(TM) SE Runtime Environment (build 21+35)
Java HotSpot(TM) 64-Bit Server VM (build 21+35, mixed mode)
javac 21
```

3. Writing and Running a Java Program

Once JDK is installed, you can write and run Java programs using Command Prompt or IDE (Eclipse, IntelliJ, NetBeans).

Step 3.1 – Create a Java Program

File: HelloWorld.java

```
// A simple Java program to display a message
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, Java is installed successfully!");  }}
```

Step 3.2 – Compile Java Program

1. Open Command Prompt
2. Navigate to the folder containing HelloWorld.java
3. Compile using javac:

```
javac HelloWorld.java
```

- This generates a HelloWorld.class file (Java bytecode) in the same folder.

Step 3.3 – Run Java Program

```
java HelloWorld
```

Output:

```
Hello, Java is installed successfully!
```

4. Another Example: Add Two Numbers

File: AddNumbers.java

```
import java.util.Scanner;
public class AddNumbers {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter first number: ");
      int a = sc.nextInt();
      System.out.print("Enter second number: ");
      int b = sc.nextInt();
      int sum = a + b;
      System.out.println("Sum = " + sum);   }}
```

Input (from user):

```
Enter first number: 10
```

Enter second number: 20

Output:

Sum = 30

16.7 TOMCAT SERVER & TESTING TOMCAT

1. What is Tomcat?

Apache Tomcat is a web server and servlet container developed by Apache Software Foundation. It is used to:

- Run Java Servlets and JavaServer Pages (JSP).
- Serve as a platform for Java-based web applications.
- Provide HTTP web server functionality without needing a separate web server like Apache HTTPD.

Key Components:

Component	Description
Catalina	Servlet container
Coyote	HTTP connector
Jasper	JSP engine (converts JSP to Servlet)
Cluster	Handles session replication for high availability

2. Installing Apache Tomcat

Step 1: Download Tomcat

- Visit: https://tomcat.apache.org
- Choose the latest Tomcat version (e.g., Tomcat 10.x or 9.x).
- Download Core zip or installer for your OS.

Step 2: Install Tomcat

- **Windows**: Extract the zip file to C:\Tomcat
- **Linux/Mac**: Extract to /usr/local/tomcat or preferred location.

Step 3: Set Environment Variables

- CATALINA_HOME = Tomcat installation folder (e.g., C:\Tomcat)
- Add %CATALINA_HOME%\bin to PATH.

Step 4: Start Tomcat

- Windows: Run startup.bat in C:\Tomcat\bin
- Linux/Mac: Run startup.sh in terminal

Test Tomcat:

Open browser and go to:

http://localhost:8080/

You should see the Tomcat Welcome Page.

3. Deploying JSP Programs in Tomcat

Step 1: Tomcat Directory Structure

```
C:\Tomcat
│
├── bin/        # Startup and shutdown scripts
├── webapps/    # Place your JSP/Servlet projects here
│   └── ROOT/   # Default web application folder
├── conf/       # Configuration files
├── logs/       # Log files
└── lib/        # Library files
```

- JSP files should go inside webapps/ROOT/ or a subfolder like webapps/MyApp/.

Step 2: Create a Sample JSP Page

File: hello.jsp (inside C:\Tomcat\webapps\ROOT\)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<html>
<head>
<title>Hello JSP</title>
</head>
<body>
<h2>Hello, Tomcat is working perfectly!</h2>
</body>
</html>
```

Step 3: Access JSP in Browser

Open:

http://localhost:8080/hello.jsp

Output:

Hello, Tomcat is working perfectly!

4. Example: JSP with Form Input

File: greet.jsp (inside ROOT/)

```
<%@ page import="java.util.*" %>
<html>
<head><title>Greeting Page</title></head>
<body>
<h2>Enter Your Name</h2>
<form action="greet.jsp" method="post">
<input type="text" name="username">
<input type="submit" value="Submit">
</form>
<%
      String name = request.getParameter("username");
if(name != null && !name.isEmpty()) {
out.println("<h3>Welcome, " + name + "! Your JSP program is working.</h3>");}
   %>
</body>
</html>
```

Input (from form): Name: Alice

Output (in browser): Welcome, Alice! Your JSP program is working.

5. Stopping Tomcat

- Windows: shutdown.bat in C:\Tomcat\bin
- Linux/Mac: shutdown.sh in terminal

6. Folder Structure Example for JSP Project

```
webapps/
└─ MyApp/
   ├─ index.jsp
   ├─ hello.jsp
   ├─ greet.jsp
   └─ WEB-INF/
      ├─ web.xml
```

└─ classes/

- JSP pages → MyApp/
- Servlets and Java classes → WEB-INF/classes/
- Deployment descriptor → WEB-INF/web.xml

16.8 SUMMARY

JavaServer Pages (JSP) is a technology used to create dynamic web content in Java, addressing some limitations of traditional Servlets. Servlets, while powerful, mix business logic with presentation, making web applications harder to maintain and scale. JSP separates presentation from logic by allowing HTML and Java code to coexist, forming a structured page with directives, scripting elements, and standard actions, collectively known as the anatomy of a JSP page. When a JSP page is requested, the server processes it by translating it into a Servlet, compiling it, and executing the generated bytecode, a process known as JSP processing. For organized development, JSP applications often follow the Model-View-Controller (MVC) design pattern, separating data handling, business logic, and user interface.

Setting up a JSP environment requires installing the Java Software Development Kit (JDK), which provides tools for compiling and running Java applications. Alongside JDK, Apache Tomcat is commonly used as the web server and servlet container to host JSP applications. Installing and configuring Tomcat, setting environment variables, and starting the server allows developers to deploy and test JSP pages. Simple JSP programs, such as displaying messages or handling form input, verify that the environment is correctly configured. With these components, developers can create dynamic, interactive web applications with clear separation of concerns.

The combination of JDK, Tomcat, and MVC design ensures maintainable, scalable, and efficient Java web applications. Proper understanding of JSP anatomy and processing helps in debugging and optimizing web applications. The setup process also includes verifying installations through example programs. Ultimately, JSP simplifies web development by integrating Java's capabilities with web presentation, providing a robust framework for enterprise-level applications.

16.9 KEY TERMS

Directive, Scriptlet, Translation, Compilation, MVC (Model-View-Controller), Servlet Container, Implicit Objects, Environment Variable, Deployment, RequestDispatcher.

16.10 SELF-ASSESSMENT QUESTIONS

1. What does the acronym JSP stand for?
2. Name one directive you can use in a JSP page.
3. What is the first step in the JSP processing lifecycle?
4. In the MVC architecture, which component handles user requests in a JSP application?
5. What software must you install before you can develop JSP pages?
6. What is the default port when you start Apache Tomcat locally?
7. What file extension is typically used for JSP pages?

8. In the JSP/Servlet environment, what folder is commonly used for placing JSP files in Tomcat's webapps?
9. Which phase of JSP processing involves converting the JSP page into a Java Servlet source file?

16.11 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Editionby Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approachby Bart Baesens, Aimee Backiel, and Seppe vanden Broucke. Wiley.
3. Java Programming with Oracle JDBCby Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development *by* David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applicationsby Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBCby Gregory Brill. Sybex.
7. Beginning JSP, JSF and Tomcat Web Development: From Novice to Professionalby Giulio Zambon & Michael Sekler. Apress.

Dr. U. Surya Kameswari