Session 1

Introduction to Java

Java is a high-level, object-oriented programming language developed by Sun Microsystems in 1995. It is widely used for building desktop applications, web applications, Android apps, and enterprise systems.

History of Java

Origin (1991)

Java was started in 1991 by James Gosling and his team at Sun Microsystems. Initially, it was designed for embedded systems and consumer electronics. The project was originally named **Oak**, after the oak tree outside Gosling's office.

Renamed to Java (1995)

- Oak was renamed to **Java** because the name Oak was already trademarked.
- Java was officially released in 1995.
- Motto: **Write Once, Run Anywhere (WORA)**.

Key Milestones

- **1996** – First public release (Java 1.0)
- **1998** – Java 2 introduced (J2SE, J2EE, J2ME editions)
- **2006** – Java became open-source (OpenJDK)
- **2010** – Sun Microsystems acquired by Oracle Corporation
- **2014+** – Java 8 introduced Lambdas and Streams (major update)
- Java continues evolving with new versions such as Java 11, 17, and 21 (LTS)

Why Java Was Created

Java was designed to:

- Be simple and easier than C++
- Remove complex features like pointers
- Provide secure and portable programs
- Support networked and distributed systems

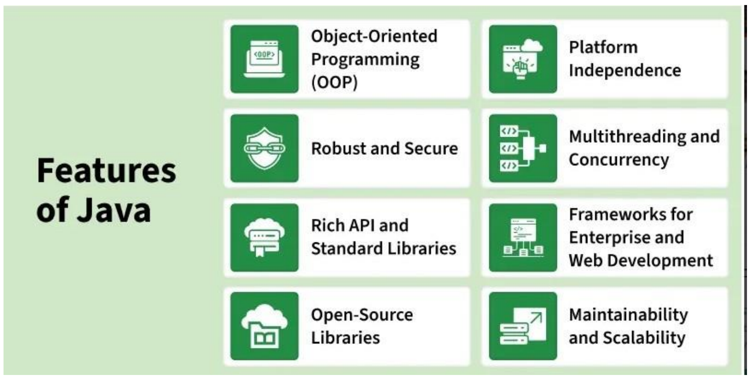

Features of Java
Object-Oriented Programming (OOP)
Platform Independence
Robust and Secure
Multithreading and Concurrency
Rich API and Standard Libraries
Frameworks for Enterprise and Web Development
Open-Source Libraries
Maintainability and Scalability

Figure 1: Features of Java

Key Features

- Object-Oriented Programming (OOP)
- Platform Independence (JVM-based execution)
- Robust and Secure
- Multithreading and Concurrency
- Rich API and Standard Libraries
- Enterprise and Web Development Framework Support
- Open-Source Libraries
- Maintainability and Scalability

Hello World Program in Java

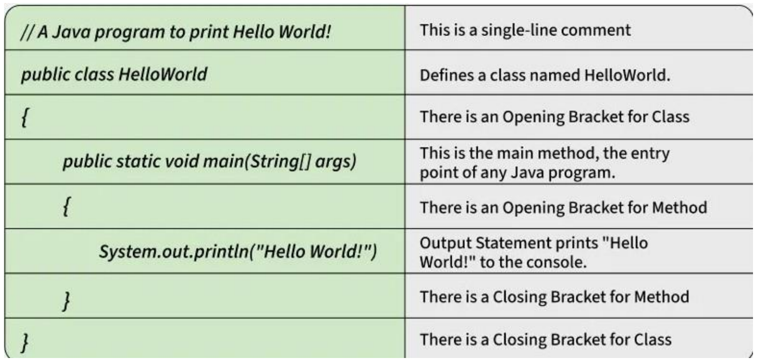
When learning any programming language, the first step is writing a simple program to display “Hello World”.

Listing 1: Hello World Program

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

Output:

```
Hello World!
```



// A Java program to print Hello World!	This is a single-line comment
public class HelloWorld	Defines a class named HelloWorld.
{	There is an Opening Bracket for Class
public static void main(String[] args)	This is the main method, the entry point of any Java program.
{	There is an Opening Bracket for Method
System.out.println("Hello World!")	Output Statement prints "Hello World!" to the console.
}	There is a Closing Bracket for Method
}	There is a Closing Bracket for Class

Figure 2: Hello World Program Explanation

How Java Program Executes

- Write code in a file: HelloWorld.java
- Java Compiler (javac) compiles it into bytecode: HelloWorld.class
- JVM reads the .class file and interprets bytecode
- JVM converts bytecode into machine-readable binary and executes the program

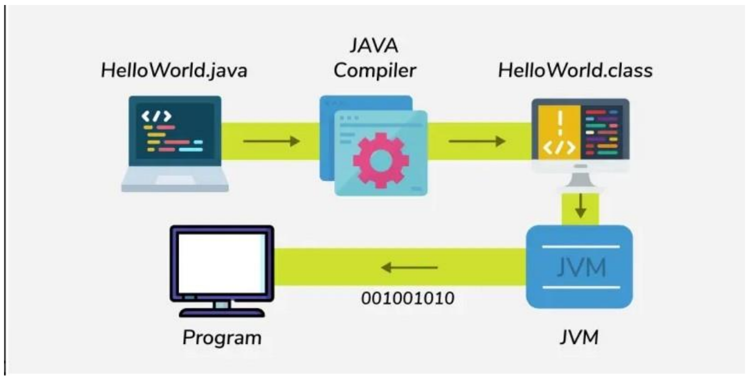


Figure 3: Java Program Execution Flow

Applications of Java

Java is used in:

- Mobile App Development (Android)
- Web Development (Spring, Spring Boot, Hibernate)
- Desktop GUI Applications (JavaFX, Swing)

- Enterprise Applications (Banking, ERP Systems)
- Game Development (LibGDX, jMonkeyEngine)
- Big Data Technologies (Hadoop, Kafka)
- Internet of Things (IoT)
- Cloud Applications (AWS, Azure, Google Cloud)
- Scientific Applications

Comments in Java

Comments are notes inside code that are not executed during runtime.

Single-line Comment

```
// This is a comment
```

Multi-line Comment

```
/*
This is a multi-line comment.
Used for explaining larger sections of code.
*/
```

Curly Braces and Indentation

Curly braces *{}* define blocks of code in Java.

Listing 2: Curly Braces Example

```
public class Geeks{
    public static void main(String[] args) {
        {
            System.out.println("This is inside the block.");
        }
        System.out.println("This is outside the block.");
    }
}
```

Curly braces group multiple statements into a block, even without conditions.

Famous Applications Built Using Java

- Android Applications
- Netflix
- Amazon
- LinkedIn
- Minecraft
- Spotify
- Uber
- NASA WorldWind

Variables

Use short and meaningful camelCase names.

```
int[] marks;
double answer;
```

Constants

Use uppercase with underscores.

```
final double PI = 3.14159;
```

Packages

Always lowercase and follow reverse-domain naming.

```
import java.util.Scanner;
import java.io.*;
```

Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is a virtual machine that executes Java bytecode. It provides platform independence by converting bytecode into machine-specific instructions.

Key Responsibilities of JVM:

- Loads, verifies, and executes bytecode
- Provides memory management and garbage collection
- Ensures platform independence
- Handles security and runtime environment

Working:

1. Java source code is compiled into bytecode (.class file)
2. JVM interprets or compiles bytecode using JIT (Just-In-Time compiler)
3. Executes the program on the target machine

Java Runtime Environment (JRE)

The Java Runtime Environment (JRE) provides the libraries and components required to run Java applications.

Components of JRE:

- JVM (execution engine)
- Core class libraries
- Supporting files and runtime resources

Note: JRE is used only to run Java programs, not to develop them.

Java Development Kit (JDK) / Java Environment

The Java Development Kit (JDK) is a complete development environment used to create, compile, and run Java programs.

JDK = JRE + Development Tools
Important Tools in JDK:

- javac – Java compiler
- java – Executes Java program
- javadoc – Generates documentation
- jar – Packages Java files into archives

Java Source File Structure

A Java program follows a specific structure.

```
package packageName;   // optional
import packageName.*;   //  optional

public class ClassName {
    public static void main(String[] args) {
        //  statements
    }
}
```

Explanation:

- package – Defines the package name (optional)
- import – Imports predefined classes (optional)
- class – Contains program logic
- main() – Entry point of Java program

Java Compilation and Execution Process

Steps:

1. Write source code in HelloWorld.java
2. Compile using: javac HelloWorld.java
3. Compiler generates bytecode: HelloWorld.class
4. Run using: java HelloWorld

Flow:

Source Code (.java) → Compiler (javac) → Bytecode (.class) → JVM → Machine Code

1 Fundamentals of Java

1. Case Sensitivity
Java is case-sensitive. Main and main are different.

2. Class Name Rule
The file name must match the public class name.

3. main() Method
Execution of a Java program begins from the main() method.

4. Object-Oriented Language
Java supports OOP concepts such as:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

5. Platform Independence
Java follows *Write Once, Run Anywhere (WORA)* principle using JVM.

6. Automatic Memory Management
Java uses Garbage Collection to free unused memory automatically.

Session 2

Object-Oriented Programming Concepts in Java

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects. Objects contain both data (attributes) and behavior (methods), and they interact with each other to perform tasks. OOP helps in building modular, reusable, and maintainable software systems.

OOP in Java closely models real-world entities and provides better code organization and scalability.

Key Features of OOP

- Promotes code reusability using classes and objects
- Enhances maintainability and scalability
- Represents real-world entities effectively
- Organizes programs into logical and manageable units

Class

A class is a blueprint or template used to create objects. It defines the properties (variables) and behaviors (methods) common to all objects of that type.

Example:

```
class Car {
    String model;
    int speed;

    void display() {
        System.out.println("Car model: " + model);
    }
}
```

Object

An object is an instance of a class. It represents a real-world entity and contains state, behavior, and identity.

Example:

```
Car c1 = new Car();
c1.model = "BMW";
c1.display();
```

Components of an Object:

- State – represented by attributes
- Behavior – represented by methods
- Identity – unique name of the object

Abstraction

Abstraction is the process of hiding implementation details and showing only essential features. It allows users to focus on what an object does instead of how it does it.

Example:

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

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

Encapsulation

Encapsulation is the wrapping of data and methods into a single unit (class) and restricting direct access to the data. This improves data security and controlled access.

Example:

```
class Student {
    private int marks;

    public void setMarks(int m) {
        marks = m;
    }

    public int getMarks() {
        return marks;
    }
}
```

Inheritance

Inheritance allows one class to acquire properties and methods of another class using the extends keyword. It promotes code reuse and establishes an “is-a” relationship.

Example:

```
class Animal {
    void eat() {
        System.out.println("Animal  eats");
    }
}
```

```
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

Types of Inheritance in Java

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance (through Interfaces)
- Hybrid Inheritance (using Interfaces)

Polymorphism

Polymorphism means “many forms”. It allows the same method to behave differently based on the context.

Types of Polymorphism:

- Compile-time Polymorphism (Method Overloading)
- Runtime Polymorphism (Method Overriding)

Example (Method Overloading):

```
class MathOp {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

Advantages of OOP

- Code reusability and modularity
- Easy maintenance and scalability
- Better data security using encapsulation
- Faster development with reusable components

Disadvantages of OOP

- Steeper learning curve for beginners
- More memory usage due to object creation
- Complex debugging in large applications

Session 3

Defining Classes in Java

A class in Java is a blueprint used to create objects. It defines properties (variables) and behaviors (methods) that the objects created from it will have.

Example:

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

    void display() {
        System.out.println(name + " " + age);
    }
}
```

In the above example, Student is a class with two data members and one method.

Constructors

A constructor is a special method used to initialize objects. It has the same name as the class and does not have a return type.

Example:

```
class Student {
    String name;

    Student(String n) {
        name = n;
    }

    void show() {
        System.out.println(name);
    }

    public static void main(String[] args) {
        Student s = new Student("Sneha");
        s.show();
    }
}
```

The constructor initializes the name when the object is created.

Methods

Methods define the behavior of a class. They contain code that performs specific tasks.

Example:

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(5, 3));
    }
}
```

Session 4

Access Specifiers

Access specifiers control the visibility of classes, variables, and methods.

- **public** – Accessible everywhere
- **private** – Accessible only within the same class
- **protected** – Accessible within package and subclasses
- **default** – Accessible only within the same package

Example:

```
class Demo {
    private int x = 10;

    public void show() {
        System.out.println(x);
    }
}
```

```
class Counter {
    static int count = 0;

    Counter() {
        count++;
    }

    public static void main(String[] args) {
        new Counter();
        new Counter();
        System.out.println("Count: " + Counter.count);
    }
}
```

Data Types

Data types specify the type of data that a variable can store.

Primitive Data Types:

- int
- float
- double
- char
- boolean

Example:

```
int num = 10;
double price = 99.99;
char grade = ’A’;
boolean flag = true;
```

Variables

Variables are containers used to store data values.

Example:

```
class VariableDemo {
    public static void main(String[] args) {
        int age = 22;
        String name = "Sneha";
        System.out.println(name + " is " + age + " years old.");
    }
}
```

Session 5

Control Flow

Control flow statements control the execution order of statements.

If-Else Statement

```
int num = 10;
if(num > 0) {
    System.out.println("Positive");
} else {
    System.out.println("Negative");
}
```

Loop Example

```
for(int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

Strings in Java

A string in Java is a sequence of characters. It is widely used to represent text data such as names, messages, and sentences. In Java, strings are objects of the String class present in the java.lang package.Unlike primitive data types, strings are immutable, which means once a string object is created, its value cannot be changed.

Creating Strings in Java

There are two common ways to create strings in Java:

Using String Literal

```
String s = "Hello World";
```

Using the new Keyword

```
String s = new String("Hello World");
```

Immutability of Strings

Strings in Java are immutable. This means any modification to a string results in the creation of a new object rather than changing the original string.

```
String s1 = "Hello";
s1.concat(" World");
System.out.println(s1); // Output: Hello
```

String Pool

Java maintains a special memory area called the String Constant Pool to store string literals. This helps in memory optimization and reusability.

```
String s1 = "Java";
String s2 = "Java";
System.out.println(s1 == s2); // true (same reference)
```

Commonly Used String Methods

- **length()** : Returns the length of the string.

- **charAt(int index)** : Returns the character at the specified index.
- **substring(int begin, int end)** : Returns a substring.
- **equals(String str)** : Compares two strings.
- **toUpperCase()** : Converts string to uppercase.

- **toLowerCase()** : Converts string to lowercase.
- **trim()** : Removes leading and trailing spaces.
- **replace(char old, char new)** : Replaces characters.

Example Program

```
public class StringExample {
    public static void main(String[] args) {
        String str = "Java Programming";

        System.out.println("Length: " + str.length());
        System.out.println("Character at index 2: " + str.charAt(2));
        System.out.println("Substring: " + str.substring(5, 16));
        System.out.println("Uppercase: " + str.toUpperCase());
        System.out.println("Lowercase: " + str.toLowerCase());
    }
}
```

String Comparison

There are two ways to compare strings in Java:

- Using == operator (compares references)
- Using equals() method (compares values)

```
String s1 = "Java";
String s2 = new String("Java");

System.out.println(s1 == s2);        // false
System.out.println(s1.equals(s2)); // true
```

Mutable Alternatives to String

Since strings are immutable, Java provides mutable classes:

- **StringBuilder**
- **StringBuffer**

Strings are one of the most frequently used classes in Java. They provide powerful methods for manipulation, comparison, and transformation of text data. Understanding string immutability, string pool, and commonly used methods is essential for efficient Java programming.

Session 6

Exception Handling in Java

The Idea Behind Exception

An exception is an abnormal condition that occurs during program execution which disrupts the normal flow of instructions. Exception handling prevents program crashes and separates error handling from normal logic.

```
int a = 10;
int b = 0;
System.out.println(a / b); // ArithmeticException
```

Exceptions and Errors

All exceptions and errors inherit from the Throwable class.

Throwable
  Error
  Exception

Error

Errors are serious problems caused by the JVM and are not recoverable. Examples: OutOfMemoryError, StackOverflowError.

Exception

Exceptions are recoverable problems that can be handled using try-catch. Examples: IOException, ArithmeticException.

Types of Exceptions

Checked Exceptions

Checked at compile time and must be handled. Examples: IOException, SQLException.

Unchecked Exceptions

Occur at runtime and handling is optional. Examples: NullPointerException, ArithmeticException.

Control Flow in Exceptions

Normal Flow:

try -> normal execution -> finally -> continue

Exceptional Flow:

try -> exception -> catch -> finally -> continue

JVM Reaction to Exceptions

When an exception occurs:

1. JVM creates an exception object
2. Searches for matching catch block
3. Executes catch if found
4. Otherwise program terminates

Keywords in Exception Handling

try

Contains risky code.

```
try {
    int x = 10/0;
}
```

catch

Handles exceptions.

```
catch(ArithmeticException e) {
    System.out.println("Division by zero");
}
```

finally

Always executes.

```
finally {
    System.out.println("Always executes");
}
```

throw

Used to explicitly throw an exception.

```
throw new ArithmeticException("Manual exception");
```

throws

Declares exceptions in method signature.

```
void readFile() throws IOException {
    // risky code
}
```

In-built and User Defined Exceptions

In-built Exceptions

Provided by Java library.

```
int arr[] = new int[5];
System.out.println(arr[10]); // ArrayIndexOutOfBoundsException
```

User Defined Exceptions

Custom exceptions created by extending Exception class.

```
class InvalidAgeException extends Exception {
    InvalidAgeException(String msg){
        super(msg);
    }
}
```

Checked vs Unchecked Exceptions

Feature	Checked	Unchecked
Compile-time check	Yes	No
Must handle	Yes	No
Inheritance	Exception	RuntimeException
Example	IOException	NullPointerException

Session 7

Packages

Packages in Java are used to organize classes and interfaces into a hierarchical namespace. They help avoid naming conflicts, provide access protection, and make large projects easier to manage.

Defining a Package

A package is defined using the package keyword at the beginning of a Java source file.

Syntax

```
package  package_name;
```

Example

```
package com.example.utility;

public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Directory Structure:

```
com/
  example/
       utility/
             Calculator.java
```

Compilation:

```
javac -d . Calculator.java
```

Explanation:

- The -d . option creates directory folders according to the package name.
- The compiled class will be placed inside the corresponding folder structure.

CLASSPATH Setting for Packages

CLASSPATH tells the JVM where to look for user-defined classes and packages.

Setting CLASSPATH Temporarily (Windows)

```
set CLASSPATH=.;C:\myprojects
```

Setting CLASSPATH Permanently

System Properties -> Environment Variables -> CLASSPATH

Example Execution

```
java com.example.utility.Calculator
```

Explanation:

- The JVM searches the specified directories to locate the class files.
- The dot (.) represents the current working directory.

Making JAR Files for Library Packages

A JAR (Java Archive) file is used to bundle multiple class files and resources into a single compressed file for distribution.

Steps to Create a JAR File

1. Compile the package classes.

   ```
   javac -d . Calculator.java
   ```

2. Create the JAR file.

   ```
   jar cf utility.jar com
   ```

Using the JAR File

```
java -cp utility.jar com.example.utility.Calculator
```

Explanation:

- jar cf creates a JAR archive.
- JAR files help distribute reusable library packages efficiently.

Import and Static Import

Java provides import statements to access classes from packages.

Session 8

Importing a Package

```
import   com.example.utility.Calculator;

public class Test {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(5, 3));
    }
}
```

Importing All Classes from a Package

```
import com.example.utility.*;
```

Static Import

Static import allows access to static members without class qualification.

```
import static java.lang.Math.*;

public class Demo {
    public static void main(String[] args) {
        System.out.println(sqrt(25));
        System.out.println(pow(2, 3));
    }
}
```

Explanation:

- import is used to access classes.
- static import allows direct use of static methods and variables.

Naming Convention for Packages

Package names should follow standard naming conventions to maintain clarity and avoid conflicts.

Rules

- Use lowercase letters only.
- Use reverse domain naming to ensure uniqueness.
- Avoid special characters and uppercase letters.

Examples

com.company.project
org.apache.commons
in.college.department.module

Best Practices:

- Start with a reversed internet domain name.
- Keep names meaningful and hierarchical.
- Avoid very long or ambiguous names.

Summary: Packages in Java help in organizing classes, improving code reuse, maintaining access control, and supporting modular development. Proper use of CLASSPATH, JAR files, and naming conventions ensures efficient package management in large-scale applications.

Session 9

Input/Output Basics

Java Input/Output (I/O) is used to read input from various sources and write output to destinations. The java.io package provides classes for performing input and output operations.

There are two main types of streams:

- Byte Streams
- Character Streams

Byte Streams

Byte streams handle raw binary data (8-bit bytes). They are suitable for images, audio, video, and binary files.

Base Classes

- InputStream – For reading bytes
- OutputStream – For writing bytes

Reading File Using Byte Stream

```
FileInputStream fis = new FileInputStream("file.txt");
int data;
while((data = fis.read()) != -1){
    System.out.print((char) data);
}
fis.close();
```

Writing File Using Byte Stream

```
FileOutputStream fos = new FileOutputStream("file.txt");
String text = "Hello Java";
fos.write(text.getBytes());
fos.close();
```

Character Streams

Character streams handle text data (16-bit Unicode characters). They are ideal for text files and international characters.

Base Classes

- Reader – For reading characters
- Writer – For writing characters

Reading File Using Character Stream

```
FileReader fr = new FileReader("file.txt");
int ch;
while((ch = fr.read()) != -1){
    System.out.print((char)ch);
}
fr.close();
```

Writing File Using Character Stream

```
FileWriter fw = new FileWriter("file.txt");
fw.write("Hello␣Character␣Stream");
fw.close();
```

Difference Between Byte Streams and Character Streams

Feature	Byte Streams	Character Streams
Data Type	8-bit bytes	16-bit characters
Base Classes	InputStream / OutputStream	Reader / Writer
Suitable For	Binary data	Text data
Encoding Support	No	Yes
Example	FileInputStream	FileReader

Reading Files in Java

Files can be read using BufferedReader for efficient reading line by line.

```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
String line;
while((line = br.readLine()) != null){
    System.out.println(line);
}
br.close();
```

Session 10

Writing Files in Java

BufferedWriter and PrintWriter are used for efficient and formatted writing.

Using BufferedWriter

```java
BufferedWriter bw = new BufferedWriter(new FileWriter("file.txt"));
bw.write("Writing␣using␣BufferedWriter");
bw.close();
```

```java
PrintWriter pw = new PrintWriter("file.txt");
pw.println("Formatted␣writing");
pw.close();
```

Using PrintWriter

- Byte Streams handle binary data.
- Character Streams handle text data.
- FileReader and FileWriter are basic text I/O classes.
- BufferedReader and BufferedWriter improve performance.
- PrintWriter supports formatted output.

Session 11

Multithreading

Thread

A thread is a lightweight sub-process that allows concurrent execution of multiple parts of a program. Java supports multithreading to improve performance and CPU utilization.
Each Java program has at least one thread called the main thread.

Advantages of Multithreading

- Better CPU utilization
- Faster execution
- Concurrent task handling
- Efficient resource sharing

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread␣is␣running...");
    }
}
```

Thread Life Cycle

A thread passes through different states during its execution.

States of Thread

1. New
2. Runnable
3. Running
4. Blocked/Waiting
5. Terminated (Dead)

Description

- New: Thread object created but not started
- Runnable: Ready to run and waiting for CPU
- Running: Currently executing
- Blocked/Waiting: Waiting for resources or another thread
- Terminated: Execution completed

Creating Threads

Method 1: Extending Thread Class

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

public class Test {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

Method 2: Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread using Runnable interface");
    }
}

public class Test {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

Difference

- Extending Thread: Cannot extend another class
- Runnable Interface: Supports multiple inheritance (preferred)

Thread Priorities

Thread priority indicates the importance of a thread for scheduling.

Priority Range

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5 (default)
- MAX_PRIORITY = 10

```
Thread t1 = new Thread();
Thread t2 = new Thread();

t1.setPriority(Thread.MIN_PRIORITY);
t2.setPriority(Thread.MAX_PRIORITY);
```

Higher priority threads get preference but execution order is not guaranteed.

Synchronizing Threads

Synchronization controls access of multiple threads to shared resources and prevents race conditions.

Synchronized Method

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}
```

Synchronized Block

```
synchronized(this) {
    count++;
}
```

Benefits

- Prevents data inconsistency
- Ensures thread safety
- Avoids race conditions

Session 12

Inter-thread Communication

Inter-thread communication allows threads to communicate using:

- wait()
- notify()
- notifyAll()

These methods belong to the Object class and must be used inside synchronized blocks.

Example: Producer-Consumer

```
class Shared {
    synchronized void produce() throws InterruptedException {
        System.out.println("Producing...");
        wait();
        System.out.println("Resumed Production");
    }

    synchronized void consume() {
        System.out.println("Consuming...");
        notify();
    }
}
```

```
}
```

Java New Features

Session 13

Functional Interface

A Functional Interface contains exactly one abstract method.

It is mainly used with Lambda Expressions.

Rules:

- Only one abstract method allowed
- Can have multiple default or static methods
- Annotated using @FunctionalInterface (optional but recommended)

 Example:

```
@FunctionalInterface
interface MyInterface {
   void display();
}
```

Using Traditional Way:

```
class Test implements MyInterface {
   public void display() {
      System.out.println("Hello");
   }
```

}

Using Lambda Expression:

```
MyInterface obj =  () -> {
      System.out.println("Hello using Lambda");};

obj.display();
```

Lambda Expressions

Lambda expressions provide a short and clean way to implement functional interfaces.

```
(parameters) -> { body }
```

Example 1: Without Parameter

```
MyInterface obj = () -> System.out.println("Lambda Example");
obj.display();
```

Example 2: With Parameter

```
interface Add {
   int sum(int a, int b);
}

Add obj = (a, b) -> a + b;
System.out.println(obj.sum(10, 20));
```

}

Advantages:

- Less code
- Improves readability
- Enables functional programming

SESSION 14

Method References

Method reference is a shorter way to call a method using :: operator.

Instead of writing lambda expression, we use method reference.

Syntax:

ClassName::methodName

Example:

```
interface Sayable {
   void say();
}

class Test {
   public static void hello() {
      System.out.println("Hello Students");
   }
}

Sayable s = Test::hello;
s.say();
```

Types of Method References:

1. Static Method Reference
2. Instance Method Reference
3. Constructor Reference

Constructor Reference Example:

```
interface Message {
   Test create();
}

Message msg = Test::new;
```

Stream API

Definition:

Stream API allows functional-style operations on collections.

It processes data using:

- filter()
- map()
- sorted()

- collect()
- forEach()

Example:

```
import java.util.*;
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(10, 20, 30, 40);

        list.stream()
            .filter(n -> n > 20)
            .forEach(System.out::println);
    }
}
```

Output:

```
30
40
```

Common Stream Methods:

- filter() → filters data
- map() → transforms data
- sorted() → sorts data
- count() → counts elements
- collect() → collects result

Default Methods in Interface

Definition:

Interfaces can have methods with implementation using default keyword.

Example:

```
interface MyInterface {
    default void show() {
        System.out.println("Default Method");
    }
}

class Test implements MyInterface {}

Test obj = new Test();
obj.show();
```

Static Methods in Interface

Static methods belong to the interface and are called using interface name.

Example:

```
interface MyInterface {
    static void display() {
        System.out.println("Static Method");
    }
}

MyInterface.display();
```

Base64 Encode and Decode

Java 8 introduced Base64 class in java.util.

Encoding Example:

```
import java.util.Base64;

public class Test {
   public static void main(String[] args) {
      String str = "Hello";

      String encoded = Base64.getEncoder()
                        .encodeToString(str.getBytes());

      System.out.println("Encoded: " + encoded);
   }
}
```

Decoding Example:

```
import java.util.Base64;

public class Test {
   public static void main(String[] args) {
      String encoded = "SGVsbG8=";

      byte[] decodedBytes = Base64.getDecoder().decode(encoded);
      String decoded = new String(decodedBytes);
```

```
        System.out.println("Decoded: " + decoded);
    }
}
```

Java provides Base64 encoder and decoder utilities.

```
String encoded = Base64.getEncoder().encodeToString("Hello".getBytes
    ());
byte[] decoded = Base64.getDecoder().decode(encoded);
System.out.println(new String(decoded));
```

Session 15

ForEach Method

Used to iterate collections with lambda.

```
List<String> names = Arrays.asList("A","B","C");
names.forEach(n -> System.out.println(n));
```

Try-with-Resources

Automatically closes resources implementing AutoCloseable.

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt
    "))) {
     System.out.println(br.readLine());
}
```

Type Annotations

Annotations can be applied anywhere a type is used.

```
List<@NonNull String> list = new ArrayList<>();
```

Repeating Annotations

Allows multiple annotations of the same type.

```
@Repeatable(Authors.class)
@interface Author {
    String name();
}
```

Java Module System

Introduced in Java 9 for modular applications.

```
module com.example.app {
    requires java.base;
}
```

Diamond Syntax with Anonymous Class

Improved type inference with anonymous inner classes.

```
Comparator<String> cmp = new Comparator<>() {
    public int compare(String a, String b) {
        return a.compareTo(b);
    }
};
```

Local Variable Type Inference (var)

Introduced in Java 10.

```
var message = "Hello␣Java";
```

Switch Expressions

Switch can return values using arrow syntax.

```
int day = 2;
String result = switch(day) {
    case 1 -> "Monday";
    case 2 -> "Tuesday";
    default -> "Other";
};
```

Yield Keyword

Used in switch expressions to return values.

```
String res = switch(day) {
    case 1: yield "Mon";
    default: yield "Other";
};
```

Text Blocks

Multi-line strings using triple quotes.

```
String text = """
␣␣␣␣Java␣Text␣Block
␣␣␣␣Multi−line␣String
""";
```

Records

Records are immutable data carrier classes.

```
record Student(int id, String name) {}
```

Sealed Classes

Restrict which classes can extend a class.

```
sealed class Shape permits Circle, Rectangle {}
final class Circle extends Shape {}
final class Rectangle extends Shape {}
```

Session 16

Collection in Java

A **Collection** is a group of objects represented as a single unit. It is part of the java.util package and provides standard methods to manipulate data structures such as lists, sets, and queues.

Collection Framework in Java

The Java Collection Framework (JCF) is a unified architecture for storing and manipulating groups of objects. It provides:

- Interfaces (Collection, List, Set, Queue, Map)
- Implementations (ArrayList, HashSet, HashMap, etc.)
- Algorithms (sorting, searching)

Hierarchy of Collection Framework

```
Iterable
   |
Collection
  |-------- List
  |          |-- ArrayList
  |          |-- LinkedList
  |          |-- Vector -> Stack
  |
  |-------- Set
  |          |-- HashSet
  |          |-- LinkedHashSet
  |          |-- TreeSet (SortedSet)
  |
  |--------- Queue
             |-- PriorityQueue
             |-- LinkedList

Map (separate hierarchy)
  |-- HashMap
  |-- LinkedHashMap
  |-- TreeMap
  |-- Hashtable
```

Iterator Interface

Iterator is used to traverse collections sequentially.

```
ArrayList<String> list = new ArrayList<>();
list.add("A"); list.add("B");

Iterator<String> it = list.iterator();
while(it.hasNext()) {
     System.out.println(it.next());
}
```

Collection Interface

The root interface of all collections except Map. It provides methods like add(), remove(), size(), clear().

List Interface

List represents an ordered collection that allows duplicates and maintains insertion order.

ArrayList

Resizable array implementation.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
System.out.println(list);
```

LinkedList

Doubly linked list implementation.

```
LinkedList<Integer> ll = new LinkedList<>();
ll.add(10);
ll.add(20);
System.out.println(ll);
```

Vector

Thread-safe dynamic array.

```
Vector<String> v = new Vector<>();
v.add("A");
v.add("B");
```

Stack

LIFO structure extending Vector.

```
Stack<Integer> stack = new Stack<>();
stack.push(10);
stack.push(20);
System.out.println(stack.pop());
```

Queue Interface

Queue follows FIFO order.

```
Queue<Integer> q = new LinkedList<>();
q.add(1);
q.add(2);
System.out.println(q.poll());
```

Set Interface

Set represents unique elements with no duplicates.

HashSet

Unordered collection.

```
HashSet<String> set = new HashSet<>();
set.add("A");
set.add("B");
```

LinkedHashSet

Maintains insertion order.

```
LinkedHashSet<String> lhs = new LinkedHashSet<>();
lhs.add("X");
lhs.add("Y");
```

SortedSet Interface and TreeSet

TreeSet stores elements in sorted order.

```
TreeSet<Integer> ts = new TreeSet<>();
ts.add(30);
ts.add(10);
System.out.println(ts); // Sorted
```

Map Interface

Map stores key-value pairs (not part of Collection hierarchy).

HashMap Class

Unordered key-value storage.

```
HashMap<Integer,String> map = new HashMap<>();
map.put(1,"Java");
map.put(2,"Python");
```

LinkedHashMap Class

Maintains insertion order.

```
LinkedHashMap<Integer,String> lhm = new LinkedHashMap<>();
lhm.put(1,"A");
lhm.put(2,"B");
```

TreeMap Class

Sorted key-value pairs.

```
TreeMap<Integer,String> tm = new TreeMap<>();
tm.put(2,"B");
tm.put(1,"A");
System.out.println(tm);
```

Hashtable Class

Thread-safe map (legacy class).

```
Hashtable<Integer,String> ht = new Hashtable<>();
ht.put(1,"X");
ht.put(2,"Y");
```

Sorting in Collections

Sorting can be done using Collections.sort().

```
ArrayList<Integer> list = new ArrayList<>();
list.add(30); list.add(10); list.add(20);
Collections.sort(list);
System.out.println(list);
```

Comparable Interface

Used for natural ordering of objects. Implement compareTo().

```
class Student implements Comparable<Student> {
    int id;
    Student(int id){ this.id=id; }
    public int compareTo(Student s){
        return this.id - s.id;
    }
}
```

Comparator Interface

Used for custom sorting logic.

```
class SortByName implements Comparator<String>{
    public int compare(String a, String b){
        return a.compareTo(b);
    }
}
```

Properties Class in Java

Properties class represents a persistent set of key-value pairs stored in files.

```
Properties p = new Properties();
p.setProperty("username", "admin");
p.setProperty("password", "1234");
System.out.println(p.getProperty("username"));
```

Summary

- Collection Framework provides unified data structures.
- List allows duplicates; Set stores unique elements.
- Map stores key-value pairs.
- Comparable and Comparator help in sorting.
- Properties class manages configuration data.

Session 17

Spring Framework

The Spring Framework is a lightweight, open-source Java framework used to develop enterprise applications. It provides comprehensive infrastructure support for developing Java applications with features such as Dependency Injection (DI), Inversion of Control (IoC), Aspect-Oriented Programming (AOP), and flexible configuration options.

Spring Core Basics

Spring Core is the fundamental part of the Spring Framework and provides IoC and DI capabilities. It manages object creation, wiring, and lifecycle management using a container called the **Spring IoC Container**.

Spring Dependency Injection (DI)

Dependency Injection is a design pattern where the dependencies of a class are provided externally by the Spring container instead of being created inside the class.

Types of Dependency Injection

- Constructor Injection
- Setter Injection

Example: Constructor Injection

```java
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;
    public Car(Engine engine) {
        this.engine = engine;
    }
    public void drive() {
        engine.start();
        System.out.println("Car is running");
    }
}
```

Spring Inversion of Control (IoC)

Inversion of Control means that the control of object creation and lifecycle is transferred from the programmer to the Spring container.

Instead of:

```
Engine e = new Engine();
Car car = new Car(e);
```

Spring Container handles:

```
ApplicationContext context =
 new ClassPathXmlApplicationContext("beans.xml");
Car car = context.getBean(Car.class);
```

Aspect-Oriented Programming (AOP)

AOP is used to separate cross-cutting concerns such as logging, security, and transactions from business logic.

Key Concepts

- Aspect: Modularization of cross-cutting concerns
- Advice: Action taken at a join point
- Join Point: Point during execution (method call)
- Pointcut: Expression selecting join points

Example

```
@Aspect
class LoggingAspect {
    @Before("execution(*␣com.app.service.*.*(..))")
    public void logBefore() {
        System.out.println("Method␣execution␣started");
    }
}
```

Bean Scopes

Spring beans can be defined with different scopes.

Singleton

Single instance per container (default).

Prototype

New instance created every time requested.

Request

One instance per HTTP request (Web apps).

Session

One instance per HTTP session.

Application

One instance per ServletContext.

WebSocket

One instance per WebSocket session.

Auto Wiring

Autowiring automatically injects dependencies without explicit configuration.

```java
@Component
class Engine {}

@Component
class Car {
    @Autowired
    private Engine engine;
}
```

Spring Annotations

Spring provides various annotations to simplify configuration:

- @Component
- @Service
- @Repository

- @Controller
- @Autowired

Life Cycle Callbacks

Spring beans have lifecycle methods:

- Initialization callback
- Destruction callback

```
@Component
class MyBean {
    @PostConstruct
    public void init() {
        System.out.println("Bean Initialized");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Bean Destroyed");
    }
}
```

Bean Configuration Styles

Spring supports multiple configuration styles:

XML Configuration

```
<bean id="engine" class="com.app.Engine"/>
<bean id="car" class="com.app.Car">
    <constructor-arg ref="engine"/>
</bean>
```

Annotation-based Configuration

```
@Component
class Engine {}

@Component
class Car {
    @Autowired
    private Engine engine;
}
```

Java-based Configuration

```
@Configuration
class AppConfig {
    @Bean
    public Engine engine() {
        return new Engine();
    }

    @Bean
    public Car car() {
        return new Car(engine());
    }
}
```

- Spring Core provides IoC and Dependency Injection.
- AOP handles cross-cutting concerns.
- Bean scopes define bean lifecycle and visibility.
- Autowiring and annotations reduce XML configuration.
- Bean configuration can be XML-based, annotation-based, or Java-based.

Session 18

Spring Boot

Spring Boot is an extension of the Spring Framework that simplifies the development of stand-alone, production-ready Spring applications. It eliminates the need for extensive XML configuration and provides embedded servers such as Tomcat or Jetty.

Spring Boot Build Systems

Spring Boot supports popular build tools for dependency management and project building.

Maven

Uses pom.xml to manage dependencies.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Gradle

Uses build.gradle for configuration.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web
    '
}
```

Spring Boot Code Structure

A typical Spring Boot project follows this structure:

```
src/main/java
    com.example.app
        Application.java
        controller/
        service/
        repository/
src/main/resources
    application.properties
```

static/

Main Class

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Spring Boot Runners

Spring Boot provides interfaces to run code after application startup:

- CommandLineRunner
- ApplicationRunner

```
@Component
class MyRunner implements CommandLineRunner {
    public void run(String... args) {
        System.out.println("Application Started!");
    }
}
```

Logger in Spring Boot

Logging is used for debugging and monitoring applications.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Demo {
    private static final Logger logger =
        LoggerFactory.getLogger(Demo.class);

    public void test() {
        logger.info("Information log");
        logger.error("Error log");
    }
}
```

Building RESTful Web Services

Spring Boot makes it easy to create REST APIs using @RestController and request mapping annotations.

Session 19

Rest Controller

A class annotated with @RestController handles HTTP requests and returns JSON responses.

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello Spring Boot";
    }
}
```

Request Mapping

Used to map HTTP requests to handler methods.

```
@RequestMapping("/api")
public class ApiController {}
```

Request Body

Used to bind HTTP request body to method parameters.

```
@PostMapping("/user")
public String addUser(@RequestBody User user) {
    return user.getName();
}
```

Path Variable

Extracts values from URI.

```
@GetMapping("/user/{id}")
public String getUser(@PathVariable int id) {
    return "User ID: " + id;
}
```

Request Parameter

Reads query parameters from URL.

```java
@GetMapping("/search")
public String search(@RequestParam String keyword) {
    return "Searching for " + keyword;
}
```

GET, POST, PUT, DELETE APIs

GET API

```java
@GetMapping("/users")
public List<User> getUsers() {
    return userService.findAll();
}
```

POST API

```java
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return userService.save(user);
}
```

PUT API

```java
@PutMapping("/users/{id}")
public User updateUser(@PathVariable int id,
                       @RequestBody User user) {
    return userService.update(id, user);
}
```

DELETE API

```java
@DeleteMapping("/users/{id}")
public String deleteUser(@PathVariable int id) {
    userService.delete(id);
    return "Deleted Successfully";
}
```

Session 20

Building Web Applications

Spring Boot supports building web applications using MVC architecture.

Controller for Web Page

```
@Controller
public class WebController {
    @GetMapping("/home")
    public String home() {
        return "home"; // returns home.html
    }
}
```

application.properties

```
server.port=8080
spring.mvc.view.prefix=/templates/
spring.mvc.view.suffix=.html
```

Summary

- Spring Boot simplifies Spring application development.
- Build tools: Maven and Gradle.
- Runners execute code after startup.
- Logger helps monitor application events.
- REST APIs created using @RestController and mapping annotations.
- Supports GET, POST, PUT, DELETE operations.
- Can build both RESTful and traditional web applications.