Introduction to Programming and Java

Prepared By

Dr. Kshyama sagar Mahanta

By the end of this lecture, you will understand:

✓ What programming is

✓ What Java is

✓ How Java evolved

✓ Major features of Java

✓ Applications of Java

What is Programming?

Definition

- Programming is the process of writing instructions that tell a computer how to perform specific tasks.
- A set of instructions written in a programming language is called a **program**.

Why Do We Need Programming?

- Programming helps us:
- Solve problems
- Automate tasks
- Process data
- Develop software
- Build websites and mobile applications

All these below applications are developed using programming languages.

Task	Software
Writing documents	MS Word
Web browsing	Chrome
Online meetings	Zoom
Banking	Banking Applications

What is Java?

Definition

- Java is a high-level, object-oriented, platform-independent programming language.
- It was designed to be simple, secure, and portable.

- Java is a programming language and computing platform that enables developers to create applications that can run on different operating systems without modification.

The Naming History

- The programming language **Java** was named after **Java coffee** (a type of espresso bean produced on the Indonesian island of Java).
- The choice was inspired by the massive amounts of coffee the development team drank during their long programming sessions.
- **The Original Name ("Oak"):** In 1991, lead creator James Gosling originally named the language Oak after a large oak tree that stood directly outside his office window.
- **The Trademark Problem:** In 1994, lawyers discovered that a pre-existing company named Oak Technology already held the trademark. The team was legally forced to find a new name before the official public release.
- While sitting in the meeting drinking Peet's Java coffee, engineer Chris Warth suggested "Java". The team loved how unique it sounded, and it easily passed the trademark checks.
- the official **Java logo is a steaming coffee cup**

History and Evolution of Java

The Green Project (1991)

- Java originated from a project called the **Green Project**.
- The project was led by James Gosling and a team of engineers at Sun Microsystems (now Oracle).

Objective

Develop software for consumer electronic devices such as:

- Television
- Set-top boxes
- Smart appliances

Timeline of Java

Year	Event
1991	Green Project Started
1992	Oak Language Developed
1995	Renamed to Java
1996	Java 1.0 Released
2004	Java 5 Introduced
2014	Java 8 Released
2021	Java 17 Released (LTS)
Present	Java continues evolving

Motto of Java

WORA (Write Once, Run Anywhere)

Meaning:

- A Java program written on one operating system can run on another operating system without rewriting the code.

Example:

A Java program written on Windows can run on:

- Linux
- macOS
- Unix

provided a Java Runtime Environment is available.

Features of Java

1. Simple

- Java syntax is easy to learn.

Example:

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

Compared to C++, Java removes complex features such as:

- Pointers
- Operator overloading
- Multiple inheritance through classes

2. Object-Oriented

Java follows OOP principles:

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Benefits

- Code reuse
- Easy maintenance
- Better organization

3. Platform Independent

- Java source code is converted into **Bytecode**.
- The bytecode can run on any system having JVM.

Execution Flow

Java Program

↓

Compiler

↓

Bytecode

↓

JVM

↓

Machine Code

4. Secure

Java provides security through:

- Bytecode verification
- No direct memory access
- Exception handling
- Security manager

Applications such as online banking use Java because of its security.

5. Robust

Robust means strong and reliable.

Java provides:

- Automatic memory management
- Garbage collection
- Exception handling

These reduce program crashes.

6. Multithreading

Java supports execution of multiple tasks simultaneously.

Example:

While downloading a file:

- Download continues
- User can browse other pages

Both activities happen simultaneously.

7. Portable

Programs can be transferred across systems without modification.

Example:

- Windows → Linux
- Linux → macOS

8. Distributed

Java supports network-based applications.

Example:

- Client-Server applications
- Web applications
- Cloud applications

9. Dynamic

- Java can load classes during runtime.

This provides flexibility and extensibility.

Applications of Java

Java is used in almost every domain of software development.

1. Desktop Applications

Examples:

- Calculator
- Notepad
- Inventory Systems

2. Web Applications

Examples:

- E-commerce websites
- Online banking portals

Technologies:

- JSP
- Servlets
- Spring Framework

3. Mobile Applications

Many Android applications are developed using Java.

Examples:

- Educational Apps
- Banking Apps
- Shopping Apps

4. Enterprise Applications

Large organizations use Java for:

- ERP Systems
- Banking Software
- Payroll Systems

5. Cloud Applications

- Java is widely used in cloud platforms and microservices.

Comparison: C++ vs Java

C++	**Java**
Platform dependent	Platform independent
Uses pointers	No explicit pointers
No automatic garbage collection	Automatic garbage collection
Supports operator overloading	Does not support operator overloading
Procedural + OOP	Primarily OOP

Assignment

Short Questions

1.Define Java.

2.What is WORA?

3.List five features of Java.

4.What is platform independence?

Long Questions

1.Explain the history and evolution of Java.

2.Discuss the features of Java in detail.

3.Compare Java and C++.

4.Explain various application areas of Java.

Lecture 2:
JDK, JRE, JVM and Java Program Development Process

Prepared By

Dr. Kshyamasagar Mahanta

Learning Objectives

- After this lecture, students will be able to:
- Define JDK, JRE, and JVM.
- Explain the architecture of Java.
- Understand the Java program execution process.
- Install and configure Java Development Kit (JDK).
- Compile and execute a Java program using command prompt.
- Differentiate between JDK, JRE, and JVM.

Java Architecture Overview

- When we write a Java program, the computer cannot directly understand it. The Java program passes through several stages before execution.

Java Architecture

Java Source Code (.java)

↓

Compiler

(javac)

↓

Bytecode (.class)

↓

JVM

↓

Machine Code

↓

Output

Java Virtual Machine (JVM)

Definition

- JVM stands for **Java Virtual Machine**.
- It is a virtual environment that executes Java bytecode.

Key Function

- Convert bytecode into machine language understood by the operating system.

The **Java Virtual Machine (JVM)** is the core execution engine of the Java ecosystem. It acts as an abstract computing machine that loads, verifies, and executes Java bytecode, bridging the gap between compiled source code and the underlying computer hardware.

Responsibilities of JVM

1. Loading Classes

Loads required class files.

2. Verifying Bytecode

Checks bytecode for security and correctness.

3. Executing Code

Converts bytecode to machine code.

4. Memory Management

Allocates and deallocates memory.

5. Garbage Collection

Automatically removes unused objects.

JVM Memory Areas (Basic Introduction)

The Java Virtual Machine (JVM) divides its memory into five distinct runtime data areas to efficiently handle program execution, thread isolation, and memory cleanup . These core regions are the Method Area, Heap, Thread Stacks, Program Counter (PC) Registers, and Native Method Stacks .

The 5 JVM Memory Areas

- **Method Area:** Stores class-level data, including metadata, method code, static variables, and runtime constant pools . It is shared among all threads and is implemented as **Metaspace** in modern Java versions.
- **Heap:** The largest memory area, where all dynamically created object instances and arrays are stored . It is shared across all threads and is automatically managed by the **Garbage Collector** to reclaim unused memory .
- **Stack Area (Thread Stacks):** Every thread gets its own private stack . Each method call creates a "stack frame" in this area, which temporarily holds local variables, method parameters, and references to objects on the Heap.
- **Program Counter (PC) Register:** A tiny, per-thread register that stores the memory address of the JVM instruction currently being executed .
- **Native Method Stack:** Dedicated to executing methods written in other languages (such as C or C++) via the Java Native Interface (JNI).

Java Runtime Environment (JRE)

Definition

- JRE stands for **Java Runtime Environment**.
- It provides everything needed to run Java programs.

JRE
├── JVM
└── Java Class Libraries

Purpose

- If a user only wants to run Java applications, JRE is sufficient.

Example:

- A person using a Java-based banking application does not need JDK.
- Only JRE is required.

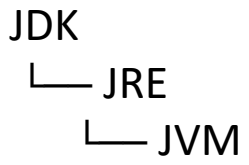
Java Development Kit (JDK)

Definition

- JDK stands for **Java Development Kit**.
- It is used by programmers to develop Java applications.

JDK

```
├── JRE
│   ├── JVM
│   └── Libraries
├── Compiler (javac)
├── Debugger
├── Documentation Tool
└── Other Development Tools
```



Installing JDK

- **Step 1** Download JDK
- **Step 2** Install JDK.
- **Step 3 Set Environment Variable** Add to PAT
- **Step 4** Verify Installation

Open Command Prompt: java –version

Check compiler:javac -version

Java Program Development Process

Step 1: Create Program Using Notepad or IDE.

File: Hello.java

Program:

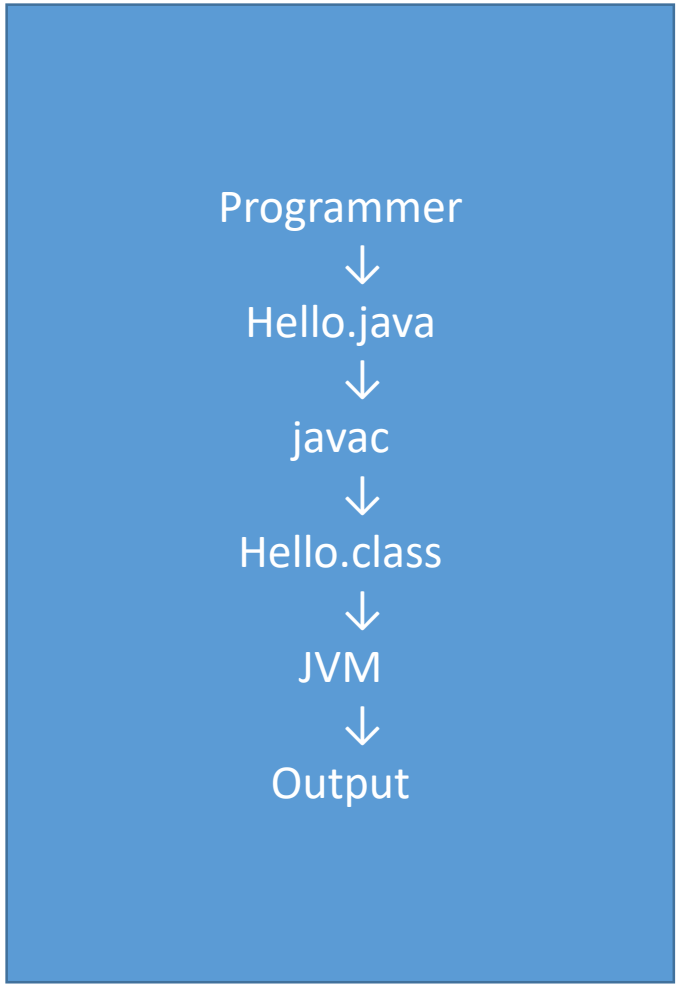
```
class Hello
{
   public static void main(String args[])
   {
      System.out.println("Hello Java");
   }
}
```

Step 2: Compile Program

Command: ***javac Hello.java***

Step 3: Execute Program

Command: java Hello



Introduction to Java Program Structure

Prepared by

Dr. Kshyamasagar Mahanta

Every Java application consists of:

- Class
- Method
- Statements
- Variables
- Objects (later)

Component	Purpose
class	Defines a class
ClassName	User-defined name
main()	Starting point of program
Statements	Instructions to execute
{}	Block delimiters

Writing the First Java Program

Program 1: Hello World

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

Understanding Each Line of the Program

class Hello

class is a keyword.

Hello is the class name.

A class is a blueprint for creating objects.

{

Opening brace indicates beginning of class body.

public static void main(String args[])

This is called the **main method**.

Execution starts from this method.

Without main(), the program cannot run as an application.

Breakdown of main()

public

Accessible from anywhere.

static

Can be called without creating an object.

void

Returns nothing.

main

Special method name recognized by JVM.

String args[]

Stores command-line arguments.

System.out.println("Hello World");

Used for displaying output.

System

Predefined class.

out

Output stream.

println()

Method used to print output and move cursor to next line.

Output Statements in Java

println()

Prints output and moves cursor to next line.

```
class Test
{
   public static void main(String args[])
   {
      System.out.println("Java");
      System.out.println("Programming");
   }
}
```

print()

Prints output on same line.

```
class Test
{
   public static void main(String args[])
   {
      System.out.print("Java ");
      System.out.print("Programming");
   }
}
```

Lecture 4: Data Types and Variables in Java

Prepared by

Dr. Kshyamasagar Mahanta

Learning Objectives

After this lecture, students will be able to:

- Understand the concept of data types in Java.
- Differentiate between primitive and non-primitive data types.
- Declare and initialize variables.
- Use different primitive data types appropriately.
- Understand memory requirements of data types.
- Follow variable naming conventions.
- Write simple Java programs using variables.

Introduction to Data Types

What is Data?

Data refers to any value that can be processed by a computer.

Examples: 25 98.5 'A‘ "Java" true

Why Do We Need Data Types?

Different types of data require different amounts of memory.

The computer must know what type of data is being stored.

This information is provided by **data types**.

Definition

A **Data Type** specifies:

- What type of value can be stored
- How much memory is required
- What operations can be performed

Categories of Data Types in Java

Java data types are divided into two categories: primitive and non-primitive

Primitive Data Types

Built-in data types provided by Java. Examples:int ,float ,double ,char ,Boolean

Non-Primitive Data Types

Created by programmers. Examples:String ,Arrays ,Classes ,Interfaces

Primitive Data Types

Java provides 8 primitive data types.

Data Type	Size	Range/Value
byte	1 Byte	-128 to 127
short	2 Bytes	-32,768 to 32,767
int	4 Bytes	Large Integers
long	8 Bytes	Very Large Integers
float	4 Bytes	Decimal Numbers
double	8 Bytes	High Precision Decimal Numbers
char	2 Bytes	Single Character
boolean	JVM Dependent	true/false

Byte Stores small integer values.

Example:

```
byte age = 25;
System.out.println(age);
```

Short Stores larger integer values than byte.

Example:

```
short marks = 32000;
System.out.println(marks);
```

Int Most commonly used integer type.

Example

```
int salary = 50000;
System.out.println(salary);
```

Long Stores very large integer values.

Example:

```
long population = 1400000000L;
System.out.println(population);
// use suffix L for Long values
```

Float Stores decimal values.

Example

```
float percentage = 85.5f;
System.out.println(percentage);
```

Double Stores high precision decimal values.

Example

```
double pi = 3.1415926535;
System.out.println(pi);
```

Feature	float	double
Size	4 Bytes	8 Bytes
Precision	Lower	Higher
Suffix Required	Yes (f)	No
Memory Usage	Less	More
Usage	Simple calculations	Scientific calculations

Char Stores a single character.

char grade = 'A';

System.out.println(grade);

Boolean *Stores logical values.*

Possible Values : true false

Example

boolean result = true;

System.out.println(result);

Variables in Java

What is a Variable?

A variable is a named memory location used to store data.

Real-Life Example

Consider a water bottle.

Bottle Name = Variable

Water = Value

Syntax

datatype variableName;

Example: int age;

Initialization

Assigning a value to a variable.

```
int age;
age = 20;
```

Declaration + Initialization

```
int marks = 85;
float percentage = 89.5f;
char grade = 'A';
```

Example:

```
class Student
{
    public static void main(String args[])
    {
        int age = 20;
        float percentage = 85.5f;
        char grade = 'A';

        System.out.println(age);
        System.out.println(percentage);
        System.out.println(grade);
    }
}
```

Rules for Naming Variables

If you break any of these rules, your Java program will fail to compile:

- **First Character:** Must begin with a letter (A-Z or a-z), a currency character ($), or an underscore (_). It cannot start with a number.
- **Subsequent Characters**: Can contain letters, numbers (0-9), underscores, or dollar signs.
- **No Special Characters**: Symbols like @, #, !, or % are strictly illegal.
- **No White Spaces**: You cannot include empty spaces inside a variable name.
- **Case Sensitivity:** Java treats uppercase and lowercase characters differently. For example, myVar and myvar are completely unique variables.
- **No Reserved Keywords**: You cannot use Java's built-in keywords (such as int, class, public, if, or void) as variable names.
- **Use meaningful names**

Program 1: Store and display student information.

Program 2: Store marks and percentage.

1.

```
class Student
{
   public static void main(String args[])
   {
      String name = "Rahul";
      int age = 20;

      System.out.println(name);
      System.out.println(age);
   }
}
```

2.

```
class Marks
{
   public static void main(String args[])
   {
      int marks = 450;
      float percentage = 90.0f;

      System.out.println(marks);
      System.out.println(percentage);
   }
}
```

Lecture 5: Operators and Control Statements in Java

Prepared by

Dr. Kshyamasagar Mahanta

Learning Objectives

After completing this lecture, students will be able to:

- Understand the purpose of operators in Java.
- Use arithmetic, relational, and logical operators.
- Write programs using operators.
- Understand decision-making statements.
- Implement if, if-else, and switch statements.
- Use looping statements (for, while, do-while).
- Choose appropriate control structures for different problems.

An **operator** is a symbol that performs a specific operation on one or more operands (values or variables).

```
int a = 10;
int b = 5;
int c = a + b;
```

Why Do We Need Operators?

- Operators help us:
- Perform calculations
- Compare values
- Make decisions
- Control program flow

Arithmetic Operators

Relational Operators

Logical Operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (Remainder)

```
class ArithmeticDemo
{
   public static void main(String args[])
   {
      int a = 20;
      int b = 10;

      System.out.println("Addition = " + (a+b));
      System.out.println("Subtraction = " + (a-b));
      System.out.println("Multiplication = " + (a*b));
      System.out.println("Division = " + (a/b));
      System.out.println("Remainder = " + (a%b));
   }
}
```

Relational Operators

Relational operators compare two values.

Operator	Meaning
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To
==	Equal To
!=	Not Equal To

```
class RelationalDemo
{
   public static void main(String args[])
   {
      int a = 10;
      int b = 20;

      System.out.println(a>b);
      System.out.println(a<b);
      System.out.println(a==b);
      System.out.println(a!=b);
   }
}
```

Logical Operators

Logical operators combine multiple conditions.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

```
if (a > b || c == d)
        System.out.println("One or both  the conditions are true");
    else
        System.out.println("Both the  conditions are false");
```

Assignment Operator

=, +=,-=,*=,/=,%=

Inrement and Decrement Operator(Unary Operator)

++, --

Ex:

```
Class IncrementOpr
{
Public static void main(String args[])
{
int m=10, n=20;
System.out.println(“m=”+ m);
System.out.println(“n=”+ n);
System.out.println(“++m=”+ ++m);
System.out.println(“n++ =”+ n++);
System.out.println(“m=”+ m);
System.out.println(“m=”+ m);
}
}
```

Conditional Operator

Exp1? Exp2:exp3

Ex: a=10; b=20;

X=(a>b)?a:b;

BITWISE Operator

&,|,^,~,<<,>>

Control Statements

Control statements determine the flow of execution of a program.

Without control statements, programs execute sequentially.

Control Statements: if, if-else, switch

if Statement

Used when a condition must be checked.

Syntax:

```
if(condition)
{
  statements;
}
```

Ex:

```
int age = 20;
if(age >= 18)
{
   System.out.println("Eligible to Vote");
}
```

if-else Statement

Used when two alternatives exist.

Example:

```
class Voting
{
   public static void main(String args[])
   {
      int age = 16;
      if(age >= 18)
      {
         System.out.println("Eligible");
      }
      else
      {
         System.out.println("Not Eligible");
      }
   }
}
```

switch Statement

Used when multiple choices are available.

```
switch(expression)
 {
 case value1:
      statement;
      break;
case value2:
     statement;
     break;
default:
     statement;
 }
```

```
class Day
{
   public static void main(String args[])
   { int choice = 2;
      switch(choice)
      {
         case 1:
            System.out.println("Monday");
            break;
         case 2:
            System.out.println("Tuesday");
            break;
         case 3:
            System.out.println("Wednesday");
            break;
         default:
            System.out.println("Invalid Choice");
      }
   }
}
```

Loops in Java

for Loop

Used when number of repetitions is known.

```
for(initialization; condition; increment)
{
    statements;
}
```

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

while Loop

Condition is checked before execution.

```
while(condition)
{
    statements;
}
```

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

do-while Loop

Condition is checked after execution.

Therefore, loop executes at least once.

```
do
{
   statements;
}
while(condition);
```

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

Feature	`while` Loop	`do-while` Loop
Condition Check	Evaluated at the **entry** of the loop.	Evaluated at the **exit** of the loop.
Minimum Iterations	Can execute **0 times** if the initial condition is false.	Will always execute **at least 1 time**.
Loop Type	Known as an **entry-controlled** loop.	Known as an **exit-controlled** loop.
Syntax Ending	No semicolon after the condition: `while(cond) { ... }`	Requires a semicolon after the condition: `while(cond);`

1. Find whether a number is even or odd.
2. Print numbers from 1 to 10 using for loop.
3. Display day name using switch.
4. Print multiplication table of 5 using while loop.
5. Check Whether a Number is Divisible by 5 and 11
6. Find the Greatest Among Three Numbers
7. Display Grade Based on Marks

Lecture 6: Input/Output Operations and Packages in Java

Learning Objectives

After completing this lecture, students will be able to:

- Understand Input and Output operations in Java.
- Use the Scanner class to accept user input.
- Read integers, float values, characters, and strings from the keyboard.
- Develop programs using Scanner class.
- Understand the concept of packages in Java.
- Create user-defined packages.
- Import and use package classes in Java programs.

Input and Output in Java

Output

System.out.println();

Example:

System.out.println("Hello Java");

Input

Java provides several ways to accept input.

Most commonly used: **Scanner Class**

Scanner is a predefined class available in the package: java.util

It is used to read data from:

- Keyboard
- Files
- Input streams

Importing Scanner

Before using Scanner, import it:

```
import java.util.Scanner;
```

Creating Scanner Object

Syntax:

```
Scanner sc = new Scanner(System.in);
```

Reading Different Types of Data

Integer Input

```
int n;
n = sc.nextInt();
```

Ex:

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter Number: "); int n = sc.nextInt();
System.out.println("Number = " + n);
```

Float Input

```
float salary = sc.nextFloat();
```

Double Input

```
double price = sc.nextDouble();
```

Character Input

Scanner has no direct nextChar() method.

Use:

```
char ch = sc.next().charAt(0);
```

Example:

```
char grade;
grade = sc.next().charAt(0);
```

String Input

Single Word

```
String name = sc.next();
```

Full Sentence

```
String address = sc.nextLine();
```

Example Program

```
import java.util.Scanner;
class Student
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Name: ");
        String name = sc.next();
        System.out.print("Enter Age: ");
        int age = sc.nextInt();
        System.out.println("Name = " + name);
        System.out.println("Age = " + age);
    }
}
```

Practice Programs Using Scanner

Program 1: Addition of Two Numbers

Program 2: Largest Among Three Numbers

Program 3: Multiplication Table

Program 4: Area of Rectangle

Program 5: Check Even or Odd

Program 6: Simple Interest

1. Addition of Two Numbers

```
import java.util.Scanner;
class Addition
{
   public static void main(String args[])
   {
      Scanner sc = new Scanner(System.in);
      int a,b,sum;
     System.out.print("Enter First Number: ");
      a = sc.nextInt();
      System.out.print("Enter Second Number: ");
      b = sc.nextInt();
      sum = a + b;
      System.out.println("Sum = " + sum);
   }
}
```

Program 2: Largest Among Three Numbers

```
import java.util.Scanner;
class Largest
{
   public static void main(String args[])
   {
      Scanner sc = new Scanner(System.in);
      int a,b,c;
      System.out.print("Enter Three Numbers: ");
      a = sc.nextInt();
      b = sc.nextInt();
      c = sc.nextInt();
      if(a>b && a>c)
         System.out.println(a + " is Largest");
      else if(b>c)
         System.out.println(b + " is Largest");
      else
         System.out.println(c + " is Largest");
   }
}
```

Program 3: Multiplication Table

```
import java.util.Scanner;
class Table
{
   public static void main(String args[])
   {
      Scanner sc = new Scanner(System.in);
      int n;
      System.out.print("Enter Number: ");
      n = sc.nextInt();
      for(int i=1;i<=10;i++)
      {
         System.out.println(n + " x " + i + " = " + (n*i));
      }
   }
}
```

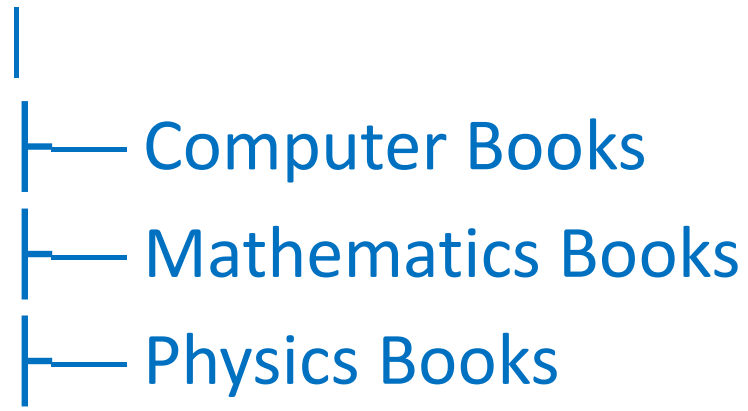
Introduction to Packages

A package is a collection of related classes and interfaces.

Real-Life Analogy

College Library

Library



Books are organized into sections.

Similarly, Java organizes classes into packages.

Advantages of Packages

1. Organization

Keeps related classes together.

2. Reusability

Classes can be reused.

3. Avoids Naming Conflicts

Two classes can have the same name in different packages.

4. Security

Provides access control.

Package	Purpose
java.lang	Core Java classes
java.util	Scanner, Collections
java.io	File handling
java.net	Networking
java.sql	Database connectivity

UNIT II: Introduction to Object-Oriented Programming (OOP)

Learning Objectives

After completing this lecture, students will be able to:

- Understand the need for Object-Oriented Programming (OOP).
- Differentiate between Procedural Programming and Object-Oriented Programming.
- Explain the six major OOP principles.
- Define Class and Object with real-life examples.
- Understand how Java implements OOP concepts.
- Design simple classes in Java.
- Identify applications of OOP in software development.

Programming Paradigms

Programming languages follow different styles.

1. Procedural Programming (POP)

Examples: C, Pascal

Focuses on:

Functions

Procedures

2. Object-Oriented Programming (OOP)

Examples:

Java

C++

C#

Python (supports OOP)

Focuses on:

Objects

Classes

Advantages of OOP

- Easy maintenance
- Reusable code
- High security
- Real-world modelling
- Better organization
- Easier debugging
- Scalable software

Procedural Programming	**Object-Oriented Programming**
Focus on Functions	Focus on Objects
Data is less secure	Data is protected
Difficult for large projects	Suitable for large projects
Less reusable	Highly reusable
Top-down approach	Bottom-up approach
Example: C	Example: Java

Principles of OOP

The six major concepts are:

— Class

— Object

— Encapsulation

— Inheritance

— Polymorphism

— Abstraction

Class

A class is a blueprint or template used to create objects.

Real-Life Example

Blueprint of a House

Blueprint

↓

House 1

House 2

House 3

The blueprint is the class. The houses are objects

Another Example

Student Admission Form

Roll Number:

Name:

Branch:

Age:

The admission form acts as a template.

Each student fills the same form.

Therefore,

Admission Form = Class

Students = Objects

Components of a Class

A class contains:

- Variables (Data Members)
- Methods (Functions)
- Constructors
- Objects

Ex:

class Employee { int id; String name; float salary; }

Q. Design class for

1. Laptop
2. Fan
3.Bus

Object

An object is an instance of a class.

Real-Life Example

Class

Car

Objects

BMW

Audi

Hyundai

Toyota

All are objects of Car class.

Class Student

Objects

Rahul

Amit

Priya

Anjali

Object Creation

Student s1 = new Student();

Student → Class

s1 → Reference Variable

new → Memory Allocation

Student() → Constructor

Difference Between Class and Object

Class	Object
Blueprint	Real Entity
No Memory	Occupies Memory
Logical	Physical
Definition	Instance
One Class	Many Objects

Encapsulation

Wrapping data and methods together into one unit is called Encapsulation.

Example

Student class

↓

Roll

Name

Marks

Display()

All inside one class.

Benefits

- Data security
- Easy maintenance
- Better organization

Inheritance

Inheritance means acquiring properties from another class.

Ex: Vehicle -> Car ->Bike

Polymorphism

Polymorphism means “many forms.” In Java, polymorphism allows the same method to perform different behaviors based on the object or parameters.

Real-Life Example: Person

- Teacher at College
- Father at Home
- Customer in Shop

One person. Different roles.

Abstraction

Showing essential information while hiding implementation details.

Real-Life Example

Driving a Car

Driver uses: Steering, Brake, Accelerator

Driver does not need to know:Engine combustion,Fuel injectionGear mechanics

ATM Example

Customer sees: Withdraw, Deposit, Balance check

Customer does not see:Database, Server, Banking logic

Benefits

- Simplicity
- Security
- Reduced complexity

Identify the following:

Item	Class	Object
Student		
Rahul		
Car		
Honda City		
Mobile		
Samsung Galaxy		

Design classes for:

- Bank Account
- Hospital Patient
- Library Book

Write at least **four data members** for each.

Quick Quiz

Fill in the Blanks

1. Java is a _____ programming language.

2. A class is a _____ of an object.

3. An object is an _____ of a class.

4. Wrapping data and methods together is called _____.
5. Acquiring properties from another class is called _____.

6. One interface, many forms is called _____.

7. Hiding implementation details is called _____.

Assignment

1. Define Object-Oriented Programming. Explain its advantages.
2. Differentiate between Procedural Programming and Object-Oriented Programming.
3. Explain the six principles of OOP with suitable examples.
4. Differentiate between a class and an object.
5. Explain encapsulation, inheritance, polymorphism, and abstraction using real-life examples.

Lecture 8 & 9 Classes, Objects, new Keyword and Reference Variables

Introduction to Classes

- A class is a **user-defined data type** that groups data (variables) and functions (methods) into a single unit.
- It acts as a **blueprint** for creating objects.

Example

Suppose a college has 500 students. Every student has: Roll Number, Name, Branch, Semester

Instead of writing these variables 500 times, we define one class.

Student

Roll Number

Name

Branch

Semester

Each student becomes an object.

Syntax of a Class

```
class ClassName
{
    // Variables
    // Methods
}
```

Ex:

```
class Student { int roll; String name; }
```

A class can contain:

- Data Members (Variables)
- Methods (Functions)
- Constructors
- Blocks
- Nested Classes

```
class Student
{ int roll;
String name;
void display()
{
System.out.println("Roll = " + roll);
System.out.println("Name = " + name); }
public static void main(String args[])
{
Student s1 = new Student();
s1.roll = 101;
s1.name = "Rahul";
s1.display();
}
}
```

Multiple Objects

Student s1 = new Student();

Student s2 = new Student();

Student s3 = new Student();

Each object has separate memory.

Example

s1.roll = 101;

s2.roll = 102;

s3.roll = 103;

New

The new keyword is used to allocate memory dynamically for an object.

Syntax

Student s = new Student();

What Happens Internally?

Step 1: Memory is allocated.

Step 2: Constructor is called.

Step 3: Reference variable stores the address.

Without new

Student s; //Only reference variable exists. No object.

With new

Student s = new Student(); Memory allocated. Object created.

```
class Car
{
String brand;
}
class Demo
{
public static void main(String args[])
{
Car c = new Car();
c.brand = "Honda";
System.out.println(c.brand);
}
}
```

Reference Variables

A reference variable stores the memory address (reference) of an object.

Example

```
Student s;
```

Here s is only a reference variable. No memory allocated.

```
Student s = new Student();
```

Now s refers to the object.

Reference variable points to object.

Memory Allocation in Java

Java **memory** *is divided into:* ***Stack Memory and Heap Memory***

Stack Stores: Local variables, Reference variables

Heap Stores: Objects

```
class Employee
{
   int id;
   String name;
   void display()
   {
      System.out.println(id);
      System.out.println(name);
   }
   public static void main(String args[])
   {
      Employee e = new Employee();
      e.id = 1;
      e.name = "Rakesh";
      e.display();
   }}
```

Q1. Create a Book class.

Members: Book ID, Book Name, Price

Display all information.

Q2. Create a Car class.

Members: Model, Price, Colour

Create two objects and display details.

Q3. Create three employee objects and assign different values.

Display all employee details.

Quiz

1. A _____ is a blueprint of an object.
2. An object is an _____ of a class.

3. The _____ keyword allocates memory.

4. Variables inside a class are called _____.

5. A variable that stores the address of an object is called a _____.

Lecture 10: Access Modifiers

What is Access Control?

In Java, not every variable or method should be accessible by everyone.

Some data should be:

- Accessible only inside the class.
- Accessible inside the package.
- Accessible by subclasses.
- Accessible from anywhere.

Java provides Access Modifiers to control this accessibility.

Why Do We Need Access Modifiers?

Suppose we have a Bank Account class.

```
class BankAccount
{
   double balance = 50000;
}
```

If every program can directly change the balance,

```
account.balance = -100000;
```

the data becomes invalid.

Access modifiers help protect data from unauthorized access.

Advantages of Access Modifiers

- Data security
- Controlled access
- Better program design
- Supports encapsulation
- Prevents accidental modification
- Improves maintainability

Access modifiers in Java are used to control the visibility and accessibility of classes, methods, and variables.

They help enforce encapsulation by restricting access to different parts of a program.

Types of Access Modifiers

Java provides four access modifiers.

Modifier	**Accessibility**
private	Only inside the same class
default	Same class and same package
protected	Same package and subclasses
public	Accessible everywhere

private

↓

default

↓

protected

↓

public

Accessibility increases from top to bottom.

Private Access Modifier

The private modifier allows access only within the same class.

No other class can directly access private members.

Syntax

```
private int age;
```

Example 1

```
class Student
{
   private int roll = 101;
   void display()
   {
      System.out.println(roll);
   }
   public static void main(String args[])
   {
      Student s = new Student();
      s.display();
   }
}
```

Example 2

```
class Student
{
   private int roll = 101;
}
class Demo
{
   public static void main(String args[])
   {
      Student s = new Student();
      System.out.println(s.roll);
   }
}
```

Output

Compile Time Error

roll has private access in Student

Default Access Modifier

If no access modifier is specified, Java automatically assigns default access.

Syntax

```
class Student
{
    int roll;
}
```

No modifier is written.

Accessibility: Accessible only inside the same package.

Example

Package college

```
├── Student.java
└── Test.java
```

Student.java

```
class Student
{
   int roll = 101;
}
```

Test.java

```
class Test
{
   public static void main(String args[])
   {
      Student s = new Student();
      System.out.println(s.roll);
   }
}
```

Output: 101

Another package

school

Cannot access roll.

Protected Access Modifier

Protected members are accessible Inside the same package and Outside the package through inheritance.

Syntax

```
protected int age;
```

Example

```
class Animal
{
   protected void sound()
   {
      System.out.println("Animal Sound");
   }
}
class Dog extends Animal
{
   public static void main(String args[])
   {
      Dog d = new Dog();
      d.sound();
   }
}
```

Public Access Modifier

Public members are accessible from anywhere.

Syntax

```
public int age;
```

Example

```
public class Student
{
   public int roll = 101;
}
```

Another class

```
class Test
{
   public static void main(String args[])
   {
      Student s = new Student();
      System.out.println(s.roll);
   }
}
```

Difference Between Access Modifiers

Feature	private	default	protected	public
Same Class	Ye	Yes	Yes	Yes
Same Package	No	Yes	Yes	Yes
Different Package	No	No	Through inheritance	Yes
Security	Highest	Medium	Medium	Lowest

```
class Student
{
   private int roll = 101;
   String branch = "CSE";
   protected String college = "CV Raman Polytechnic";
   public String university = "CGU";
   void display()
   {
      System.out.println(roll);
      System.out.println(branch);
      System.out.println(college);
      System.out.println(university);
   }
}
class Demo
{
   public static void main(String args[])
   {
      Student s = new Student();
      s.display();
   }
}
```

Example: private

```
class Person {
    // private variable
    private String name;
    public void setName(String name)  {
        this.name = name; // accessible within class
    }
    public String getName() { return name; }
}
public class CVRP {
    public static void main(String[] args)
    {
        Person p = new Person();
        p.setName("Alice");
        // System.out.println(p.name); // Error: 'name'
        // has private access
        System.out.println(p.getName());
    }
}
```

Eaxample: Default ---- withing same package

```
class Car {
    String model; // default access
}

public class Main {

    public static void main(String[] args){

        Car c = new Car();
        c.model = "Tesla"; // accessible within the same package
        System.out.println(c.model);
    }
}
```

Example: Default ---- from other package

cse.java // Class cse is having Default access modifier

```
package p1;
class cse
{
   void display()
   {
      System.out.println("Hello World!");
   }
}
```

csenew.java

```
package p2;
import p1.*;    // importing package p1
// This class is having   default access modifier
class csenew {
   public static void main(String args[]) {
         cse o = new cse();            // Accessing class cse from package p1
      o.display();
   }
}
```

Explanation: In this example, the program will show the compile-time error when we try to access a default modifier class from a different package.

Example: Protected Access Modifier

```
package p1;                          // File: Vehicle.java in package p1
public class Vehicle {
   protected int speed;
}
package p2;                          // File: Bike.java in package p2
import p1.Vehicle;
public class Bike extends Vehicle {
   void showSpeed() {
      speed = 100;                   // allowed: subclass in different package
      System.out.println(speed);
   }
}
package p2;                          // File: Test.java in package p2
import p1.Vehicle;
public class Test {
   public static void main(String[] args) {
      Bike b = new Bike();
      b.showSpeed();                 // prints 100
      Vehicle v = new Vehicle();
      // System.out.println(v.speed); // ERROR: cannot access protected outside package & non-subclass
   }}
```

Eaxmple: public

```
class MathUtils {
    public static int add(int a, int b) {
        return a + b;
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println(MathUtils.add(5, 10)); // accessible anywhere
    }
}
```

Static Members and Static Block

What is static?

In Java, the static keyword means that a member belongs to the class itself rather than to individual instances (objects) of that class.

Memory for static members is allocated only once when the class loader loads the class into memory.

The static keyword belongs to the class, not to any individual object.

A static member is shared by all objects of the class.

Syntax

```
static dataType variableName;
static returnType methodName()
{
   // statements
}
```

Why Do We Need Static Members?

Suppose a college has 500 students.

Every student belongs to the same college.

Without static:

Student 1 → College Name = CV Raman Polytechnic

Student 2 → College Name = CV Raman Polytechnic

Student 3 → College Name = CV Raman Polytechnic

...

The college name is duplicated for every object.

Using static:

Student Class

College Name = CV Raman Polytechnic

Only one copy is created and shared.

Real-Life Example

College

Every student has:

Roll Number ✓ (Different)

Name ✓ (Different)

Branch ✓ (Different)

But all students share:

College Name ✓

University Name ✓

Shared information should be declared as static.

Static Variables

A static variable is a class variable.

Only one copy exists regardless of how many objects are created.

Syntax

```
static int count;
```

Example 1

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

   public static void main(String args[])
   {
      System.out.println(count);
   }
}
```

Example 3

```
class Student
{
   int roll;
   String name;
   static String college = "CV Raman Polytechnic";
   void display()
   {
      System.out.println(roll + " " + name + " " + college);
   }
   public static void main(String args[])
   {
      Student s1 = new Student();
      Student s2 = new Student();
      s1.roll = 101;
      s1.name = "Rahul";
      s2.roll = 102;
      s2.name = "Amit";
      s1.display();
      s2.display();
   }
}
```

Changing Static Variable

```
class Student
{
   static String college = "CV Raman Polytechnic";
   public static void main(String args[])
   {
      Student.college = "CGU";
      System.out.println(Student.college);
   }
}
```

Advantages of Static Variables

- Saves memory
- Shared among all objects
- Easy to update common information
- Improves efficiency

Static Methods

A static method belongs to the class.

It can be called without creating an object.

Syntax

```
static void show()
{

}
```

Example

```
class Test
{
   static int count = 10;
   static void show()
   {
      System.out.println(count);
   }
   public static void main(String args[])
   {
      show();
   }}
```

Calling Static Method

Method 1

show();

Method 2 using class name

Test.show();

Example

```
class Calculator
{
   static int add(int a, int b)
   {
      return a + b;
   }
   public static void main(String args[])
   {
      int sum = Calculator.add(10,20);
      System.out.println(sum);
   }
}
```

Rules of Static Methods

Static methods

✓ Can access static variables

✓ Can call static methods

✗ Cannot directly access instance variables

✗ Cannot use *this*

Example

```
class Demo
{
   int x = 10;
   static void show()
   {
      System.out.println(x);
   }
}
```

Output: Compile Time Error

Reason:

Static methods belong to the class.

Instance variables belong to objects.

Correct Version

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

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

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

Static Block

A static block is executed automatically once, when the class is loaded into memory.

Syntax

```
static
{
   // statements
}
```

Example 1

```
class Test
{
   static
   {
      System.out.println("Static Block");
   }
   public static void main(String args[])
   {
      System.out.println("Main Method");
   }
}
```

output: Static Block
Main Method

Execution Order

Class Loading

↓

Static Block

↓

Main Method

Example 2

```
class Demo{
static
   {
      System.out.println("Block 1");
   }
   static
   {
      System.out.println("Block 2");
   }
   public static void main(String args[])
   {
      System.out.println("Main");
   }}
```

Multiple Static Blocks

Java executes them in the order they appear.

Static Variable + Static Block

```
class Test
{
    static int x;
    static
    {
        x = 100;
    }
    public static void main(String args[])
    {
        System.out.println(x);
    }
}
```

Uses of Static Blocks

- Initialize static variables
- Load configuration values
- Initialize database connections
- Execute one-time setup code

Static vs Instance Members

Static Member	Instance Member
Belongs to class	Belongs to object
One copy	Separate copy for each object
Loaded once	Loaded with every object
Accessed using class name	Accessed using object
Memory efficient	Uses more memory

Uses of Static Blocks

- Initialize static variables
- Load configuration values
- Initialize database connections
- Execute one-time setup code

Static vs Instance Members

Static Member	Instance Member
Belongs to class	Belongs to object
One copy	Separate copy for each object
Loaded once	Loaded with every object
Accessed using class name	Accessed using object
Memory efficient	Uses more memory

Program 1: Count Number of Objects

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

   Student()
   {
      count++;
   }
   public static void main(String args[])
   {
      new Student();
      new Student();
      new Student();

      System.out.println("Objects Created = " + count);
   }
}
```

Program 2: Employee Details

```
class Employee
{
   int id;
   String name;
   static String company = "Infosys";
   Employee(int i,String n)
   {
      id=i;
      name=n;
   }
   void display()
   {
      System.out.println(id+" "+name+" "+company);
   }
   public static void main(String args[])
   {
      Employee e1=new Employee(1,"Rahul");
      Employee e2=new Employee(2,"Amit");
      e1.display();
      e2.display();
   }
}
```

Program 3: Static Method

```
class MathDemo
{
    static int square(int x)
    {
        return x*x;
    }

    public static void main(String args[])
    {
        System.out.println(square(5));
    }
}
```

Program 4: Static Block

```
class Welcome
{
    static
    {
        System.out.println("Welcome to Java");
    }

    public static void main(String args[])
    {
        System.out.println("Programming with Java");
    }
}
```

Program 5: Static Variable Shared by Objects

```
class Test
{
   static int x = 10;
   public static void main(String args[])
   {
      Test t1 = new Test();
      Test t2 = new Test();
      t1.x = 50;
      System.out.println(t2.x);
   }
}
```

Output

50

Explanation:

Since x is static, it is shared by all objects.

Quick Quiz

1. A static variable belongs to the _____.

2. Only _____ copy of a static variable exists.

3. Static methods can be called without creating an _____.

4. A static block executes when the _____ is loaded.

5. The main() method is declared _____.

Constructors

Why Do We Need Constructors?

Consider this class:

```
class Student
{
   int roll;
   String name;
}
```

We create an object:

```
Student s = new Student();
```

Now suppose we want to initialize:

```
roll = 101;
name = "Rahul";
```

We could write:

```
s.roll = 101;
s.name = "Rahul";
```

This works, but there is a better way.

We can initialize the object at the time of object creation using a constructor.

```
Student s = new Student(101, "Rahul");
```

This makes object initialization easier and more organized.

What is a Constructor?

Definition: A constructor is a special member of a class that is automatically invoked when an object is created.

Its primary purpose is to initialize an object.

Example:

```
class Student
{
   int roll;
   Student()
   {
      roll = 101;
   }
}
```

Object creation:

```
Student s = new Student();
```

When this statement executes:

new Student()

↓

Constructor called

↓

roll initialized

↓

Object created

Constructor Syntax

```
class ClassName
{
   ClassName()
   {
      // initialization
   }
}
```

Example:

```
class Student
{
   Student()
   {
      System.out.println("Constructor Called");
   }
}
//Creating an object:
Student s = new Student();
```

//Output:

Constructor Called

Important Characteristics of Constructors

A constructor has several important characteristics.

1. Constructor name must be the same as the class name

Ex:

```
class Student
{
   Student()
   {
   }
}
```

2. Constructor does not have a return type

Correct:

```
Student()
{
}
```

Incorrect:

```
void Student()
{
}
```

The second one is a method, not a constructor.

3. Constructor is automatically called

```
Student s = new Student();
```

The constructor executes automatically.

We don't normally call it like: `s.Student();`

4. Constructors are mainly used for initialization

Example:

```
Student()
{
   roll = 101;
   name = "Rahul";
}
```

5. Constructors can be overloaded

A class can have multiple constructors with different parameter lists.

Example:

```
Student()
{
}
Student(int r)
{
   roll = r;
}
```

Default Constructor / No-Argument Constructor

A no-argument constructor is a constructor that takes no arguments:

```
Student()
{
}
```

Example 1: Explicit No-Argument Constructor

```
class Student
{
   int roll;     String name;
   Student()
   {
      roll = 101;      name = "Rahul";
   }
   void display()
   {
      System.out.println("Roll = " + roll);
      System.out.println("Name = " + name);
   }
   public static void main(String args[])
   {
      Student s = new Student();
      s.display();
   }
}
```

What Happens When new Student() Executes?

Consider:

Student s = new Student();

Execution:

Student s

↓

Reference variable created

↓

new Student()

↓

Memory allocated

↓

Constructor invoked

↓

Object initialized

↓

Reference stored in s

Compiler-Provided Default Constructor

Consider:

```
class Student
{
   int roll;
   String name;
}
```

There is no constructor written by the programmer.

Java automatically provides a default constructor conceptually equivalent to:

```
Student()
{
}
```

Therefore: `Student s = new Student();` is valid.

The instance variables receive their default values:

int → 0

String → null

boolean → false

double → 0.0

Example

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

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

      System.out.println(s.roll);
      System.out.println(s.name);
   }
}
```

Output:

0

null

Important Examination Point

If the programmer writes any constructor, Java does not automatically provide another no-argument constructor.

Example:

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

This causes a compilation error because only:

Student(int)

exists.

There is no:

Student()

Parameterized Constructor

A constructor that accepts one or more parameters is called a parameterized constructor.

Example

```
class Student
{
   int roll;
   Student(int r)
   {
      roll = r;
   }
   void display()
   {
      System.out.println("Roll = " + roll);
   }
   public static void main(String args[])
   {
      Student s = new Student(101);
      s.display();
   }
}
```

Why Use Parameterized Constructors?

Suppose we want to create three students.

Without constructor:

```
Student s1 = new Student();
s1.roll = 101;
Student s2 = new Student();
s2.roll = 102;
Student s3 = new Student();
s3.roll = 103;
```

Using a parameterized constructor:

```
Student s1 = new Student(101);
Student s2 = new Student(102);
Student s3 = new Student(103);
```

The second approach is cleaner.

Parameterized Constructor with Multiple Parameters

```
class Student
{
   int roll;
   String name;
   String branch;
   Student(int r, String n, String b)
   {
      roll = r;
      name = n;
      branch = b;
   }
   void display()
   {
      System.out.println("Roll   = " + roll);
      System.out.println("Name   = " + name);
      System.out.println("Branch = " + branch);
   }
   public static void main(String args[])
   {
      Student s = new Student(101, "Rahul", "CSE");
      s.display();
   }
}
```

Creating Multiple Objects Using a Parameterized Constructor

```
class Student
{
   int roll;
   String name;
   Student(int r, String n)
   {
      roll = r;
      name = n;
   }
   void display()
   {
      System.out.println(roll + " " + name);
   }
   public static void main(String args[])
   {
      Student s1 = new Student(101, "Rahul");
      Student s2 = new Student(102, "Amit");
      Student s3 = new Student(103, "Priya");
      s1.display();
      s2.display();
      s3.display();
   }
}
```

Constructor with this Keyword

A common situation is:

```
class Student
{
   int roll;
   Student(int roll)
   {
      roll = roll;
   }
}
```

This creates a problem because the parameter and instance variable have the same name.

We use this to distinguish them.

```
class Student
{
   int roll;
   Student(int roll)
   {
      this.roll = roll;
   }
}
```

Meaning of *this*

this refers to the current object.

Therefore:

this.roll

means:

The roll variable belonging to the current object.

And:

roll

means the constructor parameter.

Complete Example Using this

```
class Student
{
   int roll;
   String name;
   Student(int roll, String name)
   {
      this.roll = roll;
      this.name = name;
   }
   void display()
   {
      System.out.println("Roll = " + roll);
      System.out.println("Name = " + name);
   }
   public static void main(String args[])
   {
      Student s1 = new Student(101, "Rahul");
      Student s2 = new Student(102, "Amit");
      s1.display();
      s2.display();
   }
}
```

Constructor vs Method

Constructor	Method
Initializes an object	Performs an operation
Same name as class	Can have any valid name
Cannot have a return type	May have a return type
Automatically invoked during object creation	Usually explicitly invoked
Called when object is created	Called when required
Can be overloaded	Can be overloaded

Example

Constructor:

```
Student()
{
   roll = 101;
}
```

Method:

```
void display()
{
   System.out.println(roll);
}
```

Constructor Overloading

A class can have more than one constructor with different parameter lists. This is called constructor overloading.

```
class Student {
int roll; String name;
Student() {
roll = 0; name = "Unknown"; }
Student(int r) {
 roll = r; name = "Unknown";
 } Student(int r, String n) {
roll = r; name = n;
}
void display() {
System.out.println(roll + " " + name); }
public static void main(String args[])
{
Student s1 = new Student();
Student s2 = new Student(101);
Student s3 = new Student(102, "Rahul");
s1.display();
s2.display();
s3.display(); } }
```

Important Points to Remember

1. Constructor name must be exactly the same as the class name.

2. Constructor cannot have a return type.

3. Constructors are automatically called during object creation.

4. Constructors are mainly used for object initialization.

5. Constructors can accept parameters.

6. A class can have multiple constructors.

7. If no constructor is declared, the compiler provides a default constructor.

8. If a programmer declares a constructor, the compiler does not automatically add a no-argument constructor.

Q. Create an Employee class containing:

Employee ID

Name

Salary

Use a parameterized constructor.

Q. Create a Rectangle class.

Data members:

length

breadth

Initialize them using a parameterized constructor.

Calculate area:

Q. Create a Book class with:

Book ID

Title

Author

Price

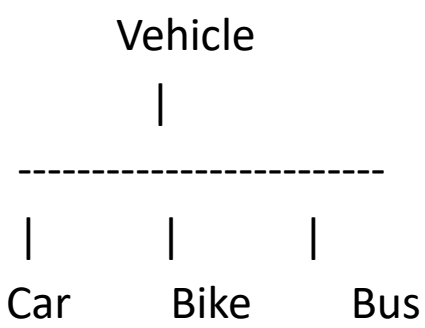
Use a parameterized constructor to initialize all values.

Inheritance

Inheritance is an OOP mechanism through which one class acquires properties and behaviors of another class.
In Java, inheritance between classes is implemented using the: **extends**
keyword.

Real-Life Example

Consider:



A vehicle generally has:
Speed
Colour
Engine
Start()
Stop()

Instead of defining these again in every class, we can put common features in Vehicle.
Then Car, Bike, and Bus can inherit them.

Superclass

The class whose properties are inherited.

Also called:

Parent class

Base class

Subclass

The class that inherits properties.

Also called:

Child class

Derived class

Example:

```
class Vehicle
{
}
```

Vehicle → Superclass

```
class Car extends Vehicle
{
}
```

Car → Subclass

```
class Animal
{
   void eat()
   {
      System.out.println("Animal eats");
   }
}
class Dog extends Animal
{
   void bark()
   {
      System.out.println("Dog barks");
   }
   public static void main(String args[])
   {
      Dog d = new Dog();
      d.eat();
      d.bark();
   }
}
```

Advantages of Inheritance

1. Code Reusability

Existing code can be reused.

2. Reduces Duplication

Common functionality is written once.

3. Easier Maintenance

Changes to common functionality can be made in the parent class.

4. Extensibility

A child can add its own features.

5. Supports Polymorphism

Inheritance is fundamental to runtime polymorphism in Java.

```
class Employee
{
   void work()
   {
      System.out.println("Employee works");
   }
}

class Manager extends Employee
{
   void manage()
   {
      System.out.println("Manager manages");
   }

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

      m.work();
      m.manage();
   }
}
```

What Can a Child Class Access?

A child class can access appropriate members of its parent class.

For example:

```
class Parent
{
   int x = 10;
   void show()
   {
      System.out.println("Parent method");
   }
}
class Child extends Parent
{
   void display()
   {
      System.out.println(x);
      show();
   }
}
```

The child can access the inherited x and show().

Accessibility of Members in Inheritance

Consider:

```
class Parent
{
   private int a = 10;
   int b = 20;
   protected int c = 30;
   public int d = 40;
}
```

If:

```
class Child extends Parent
{
}
```

then:

Member	**Child Can Directly Access?**
private	No
default	Yes, subject to package
protected	Yes
public	Yes

Important Point About private

Private members belong to the parent class but are not directly accessible from the child class.

Example:

```
class Parent
{
   private int x = 10;
}
class Child extends Parent
{
   void show()
   {
      // System.out.println(x);  // Error
   }
}
```

The child cannot directly access x.

Protected Member

```
class Parent
{
    protected int x = 10;
}
class Child extends Parent
{
    void show()
    {
        System.out.println(x);
    }
}
```

This is allowed.

Types of Inheritance

For class inheritance, three common forms are directly supported:

- Single inheritance
- Multilevel inheritance
- Hierarchical inheritance

Java does not support multiple inheritance of classes.

Interfaces provide a mechanism for achieving multiple-type inheritance.

Single Inheritance

One parent class and one child class.

Parent

↓

Child

Example:

```
class Animal
{
   void eat()
   {
      System.out.println("Eating");
   } }
class Dog extends Animal
{
   void bark()
   {
      System.out.println("Barking");
   }
   public static void main(String args[])
   {
      Dog d = new Dog();
      d.eat();
      d.bark();
   }  }
```

Multilevel Inheritance

Inheritance occurs through multiple levels.

Grandparent

↓

Parent

↓

Child

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

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

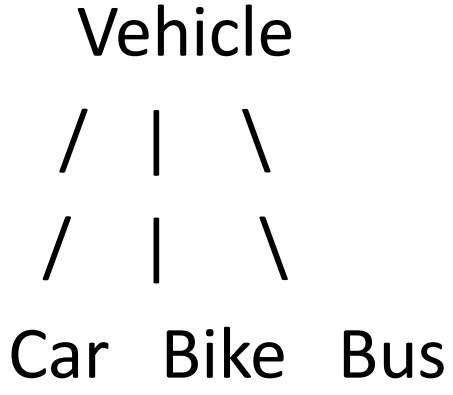
```
class Puppy extends Dog
{
   void play()
   {
      System.out.println("Playing");
   }

   public static void main(String args[])
   {
      Puppy p = new Puppy();

      p.eat();
      p.bark();
      p.play();
   }
}
```

Hierarchical Inheritance

Multiple child classes inherit from the same parent class.




```
class Vehicle
{
   void start()
   {
      System.out.println("Vehicle starts");
   }  }
class Car extends Vehicle
{
   void drive()
   {
      System.out.println("Car drives");
   }   }
class Bike extends Vehicle
{
   void ride()
   {
      System.out.println("Bike rides");
   }  }
```

```
class Demo
{
   public static void main(String args[])
   {
      Car c = new Car();
      Bike b = new Bike();
      c.start();
      c.drive();
      b.start();
      b.ride();
   }
}
```

Multiple Inheritance of Classes not possible in java

Java **does not allow** a class to extend more than one class.

This is invalid:

```
class A
{
}
class B
{
}
class C extends A, B
{
}
```

This produces a compilation error.

Why?

One major reason is ambiguity.

Suppose:



If both B and C contain a method with the same name, which implementation should D inherit?

This type of ambiguity is one reason Java avoids multiple inheritance of classes.

Java interfaces provide a different mechanism for supporting multiple inheritance .

super Keyword

The super keyword refers to the immediate parent class object/context.

It is commonly used to:

- Access parent class variables.
- Call parent class methods.
- Call parent class constructors.

super to Access Parent Variable

```
class Parent
{
   int x = 10;
}
class Child extends Parent
{
   int x = 20;
   void display()
   {
      System.out.println(x);
      System.out.println(super.x);
   }
   public static void main(String args[])
   {
      Child c = new Child();
      c.display();
   }
}
```

super to Call Parent Method

```
class Parent
{
   void display()
   {
      System.out.println("Parent display()");
   }
}
class Child extends Parent
{
   void display()
   {
      System.out.println("Child display()");
      super.display();
   }
   public static void main(String args[])
   {
      Child c = new Child();
      c.display();
   }
}
```

Method Overriding

Method overriding in Java occurs when a subclass provides its own specific implementation of a method defined in its parent class, sharing the identical name, parameter list, and a compatible return type. This mechanism forms the basis of runtime polymorphism.

Example:

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

Child:

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

The child overrides the parent's sound() method.

Complete Method Overriding Program

```
class Animal
{
   void sound()
   {
      System.out.println("Animal makes sound");
   }
}
class Dog extends Animal
{
   @Override
   void sound()
   {
      System.out.println("Dog barks");
   }
   public static void main(String args[])
   {
      Dog d = new Dog();
      d.sound();
   }
}
```

Why @Override?

@Override is an annotation that tells the compiler:

"This method is intended to override a superclass method."

It helps detect mistakes in the method signature.

Example:

```
@Override
void sound()
```

If the parent does not have a compatible method, the compiler can report an error.

Overloading vs Overriding

Method Overloading	Method Overriding
Same class commonly	Parent and child classes
Same method name	Same method name
Different parameter list	Same method signature
Compile-time polymorphism	Runtime polymorphism
Inheritance not required	Inheritance required

Program 1 — Single Inheritance

Create class :

Person
↓
Student

Person should contain:
name
age

Student should contain:
roll
branch

Display all information.

Program 2 — Multilevel Inheritance

Create class:

Person

↓

Employee

↓

Manager

Add suitable data members and methods at each level.

Q. Write a Java program to demonstrate hierarchical inheritance using Employee as the superclass and Manager and Developer as subclasses.

Requirements:

1. Create a superclass Employee with:

name

salary

a method displayEmployee()

2. Create two subclasses:

Manager with a method manageTeam()

Developer with a method writeCode()

3. Create objects of both Manager and Developer.

Dynamic Method Dispatch in Java

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in its superclass.

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

- **Dynamic Method Dispatch** is the mechanism by which a call to an **overridden method** is resolved at **runtime** rather than compile time.
- It is one of the most important mechanisms used to achieve **runtime polymorphism** in Java.

When a superclass reference refers to a subclass object, and an overridden method is called, Java executes the method belonging to the actual object at runtime.

Consider:

Animal a; // The variable a is a reference of type Animal.

Now:

a = new Dog(); // The reference a is pointing to a Dog object.

Therefore:

Reference Type Animal

Object Type Dog

Now suppose we call:

a.sound();

Which sound() method should execute?

Although the reference is of type Animal, the actual object is a Dog.

Therefore Java executes: Dog.sound()

This decision is made at runtime.

Consider:

Animal a; // The variable a is a reference of type Animal.

Now:

a = new Dog(); // The reference a is pointing to a Dog object.

Therefore:

Reference Type Animal

Object Type Dog

Now suppose we call:

a.sound();

Which sound() method should execute?

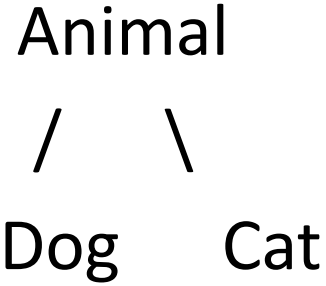
Although the reference is of type Animal, the actual object is a Dog.

Therefore Java executes: Dog.sound()

This decision is made at runtime.

```
class Animal
{
   void sound()
   {
      System.out.println("Animal makes sound");
   }
}
class Dog extends Animal
{
   @Override
   void sound()
   {
      System.out.println("Dog barks");
   }
}
public class DynamicDemo
{
   public static void main(String args[])
   {
      Animal a = new Dog();
      a.sound();
   }
}
//   In Dynamic Method Dispatch, the method executed depends on the actual object, not the reference type.
```

Dynamic Method Dispatch with Multiple Classes



Each class overrides sound().

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

Animal a;

We can assign different objects to the same reference.

a = new Dog();
a.sound();

Output:
Dog barks

Then:
a = new Cat();
a.sound();

Output:
Cat meows

```
class Animal
{
   void sound()     {
      System.out.println("Animal makes sound");
   }    }
class Dog extends Animal
{    @Override
   void sound()
   {
      System.out.println("Dog barks");
   }     }
class Cat extends Animal
{    @Override
   void sound()   {
      System.out.println("Cat meows");
   }    }
public class DynamicMethodDemo
{   public static void main(String args[])   {
      Animal a;
      a = new Dog();
      a.sound();
      a = new Cat();
      a.sound();
      a = new Animal();
      a.sound();
   }      }
```

Why is it Called "Dynamic"?

The method selection is dynamic because the decision is made during program execution.

Compare:

Animal a = new Dog();

a.sound();

The compiler knows:

a → Animal reference

But at runtime, Java knows:

a → Dog object

Therefore the overridden Dog.sound() method is selected.

Dynamic Method Dispatch is also called Runtime Polymorphism

```
class Vehicle
{
   void start()
   {
      System.out.println("Vehicle starts");
   }
}
class Car extends Vehicle
{
   @Override
   void start()
   {
      System.out.println("Car starts with key");
   }
}
class Bike extends Vehicle
{
   @Override
   void start()
   {
      System.out.println("Bike starts with self-start");
   }
}
```

```
Vehicle v;
v = new Car();
v.start();
v = new Bike();
v.start();
```

Outout:

```
car starts with key
Bike starts with self-start
```

Role of @Override

We generally write:

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

@Override is an annotation that tells the compiler:

This method is intended to override a superclass method.

It helps detect mistakes.

Important Rules of Dynamic Method Dispatch

Rule 1 : Dynamic Method Dispatch requires inheritance.

Example:

class Dog extends Animal

Rule 2 : The method must be overridden.

Example:

Animal.sound()

Dog.sound()

Rule 3 : The reference can be of superclass type.

Animal a = new Dog();

Rule 4 : The actual object can be a subclass object.

new Dog()

Rule 5 : The overridden method is selected according to the actual object.

Animal a = new Dog();

a.sound();

calls:

Dog.sound();

Rule 6 : Dynamic dispatch applies to instance methods, not static methods.

Advantages of Dynamic Method Dispatch

1. Runtime Polymorphism

Different objects can respond differently to the same method call.

2. Flexibility

A superclass reference can represent different subclass objects.

3. Extensibility

New subclasses can be added without changing much existing code.

4. Maintainability

Common interfaces can be defined in the superclass.

5. Loose Coupling

Code can work with general superclass types instead of specific implementations.

Abstract Class, Preventing Overriding & Preventing Inheritance

Learning Objectives

By the end of this lecture, students will be able to:

- Define an abstract class.
- Explain abstract methods and concrete methods.
- Understand why and when abstract classes are used.
- Create abstract classes and subclasses in Java.
- Explain why objects of an abstract class cannot be created directly.
- Understand the use of the abstract keyword.
- Explain how the final keyword prevents method overriding.
- Explain how the final keyword prevents inheritance.
- Differentiate abstract and final.
- Write Java programs using abstract classes and final methods/classes.

ABSTRACT CLASS

Why Do We Need Abstraction?

Suppose we have:



Every shape has an area, but the formula for calculating area is different.

For a circle: Area = π × r × r

For a rectangle: Area = length × breadth

So we may want the superclass to say:

Every Shape must have an area() method.

But the superclass itself may not provide the actual implementation.

This is where an abstract class is useful.

What is an Abstract Class?

An abstract class is a class declared using the abstract keyword that may contain abstract methods as well as concrete methods.

Syntax

```
abstract class ClassName
{
   // members
}
```

Example:

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

What is an Abstract Method?

An abstract method is a method declared without a body.

Syntax

abstract returnType methodName();

Example:

```
abstract void area();
```

Notice that there is no method body

no { }

a semicolon(;) at the end.

Abstract Method vs Normal Method

Normal method

```
void display()
{
   System.out.println("Hello");
}
```

It has a body.

Abstract method

```
abstract void display();
```

It does not have a body.

Example of Abstract Class

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

Now create a subclass:

```
class Circle extends Shape
{
   double radius = 5;
   void area()
   {
      double result = Math.PI * radius * radius;
      System.out.println("Area of Circle = " + result);
   }
}
```

Main class:

```
public class AbstractDemo
{
   public static void main(String args[])
   {
      Circle c = new Circle();
      c.area();
   }
}
```

Important Point

The subclass must provide an implementation of the abstract method.

For example:

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

If:

class Circle extends Shape then Circle must implement:

```
void area()
```

Otherwise, Circle must also be declared abstract.

```
abstract class Shape
{
   abstract void area();
}
class Circle extends Shape
{
   double radius;
   Circle(double radius)
   {
      this.radius = radius;
   }
   @Override
   void area()
   {
      double result = Math.PI * radius * radius;
      System.out.println("Area of Circle = " + result);
   }
}
```

```
class Rectangle extends Shape    {
   double length;
   double breadth;
   Rectangle(double length, double breadth)
   {
      this.length = length;
      this.breadth = breadth;
   }
   @Override
   void area()    {
      double result = length * breadth;
      System.out.println("Area of Rectangle = " + result);
   }    }
public class ShapeDemo    {
   public static void main(String args[]    {
      Circle c = new Circle(5);
      c.area();
      Rectangle r = new Rectangle(10, 5);
      r.area();
   }   }
```

Can We Create an Object of an Abstract Class? No.

The following is invalid:

```
Shape s = new Shape();
```

It produces a compilation error because Shape is abstract.

Why?

An abstract class may contain methods without implementation.

Therefore Java does not allow direct object creation of an abstract class.

But Can We Create a Reference of an Abstract Class? Yes.

This is valid:

```
Shape s;
```

We can also write:

```
Shape s = new Circle(5);
```

This is valid because the reference is of type Shape, but the actual object is Circle.

Abstract Class with Concrete Method

An abstract class can contain both:

- Abstract methods
- Concrete methods

Example:

```
abstract class Animal
{
   abstract void sound();

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

```
abstract class Animal
{
   abstract void sound();

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

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

```
public class AnimalDemo
{
   public static void main(String args[])
   {
      Dog d = new Dog();

      d.sound();
      d.eat();
   }
}
```

```
//Abstract Class with Constructor
//An abstract class can have a constructor.
abstract class Animal
{
   Animal()
   {
      System.out.println("Animal Constructor");
   }
   abstract void sound();
}
class Dog extends Animal
{
   Dog()
   {
      System.out.println("Dog Constructor");
   }

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

```
public class Test
{
   public static void main(String args[])
   {
      Dog d = new Dog();
      d.sound();
   }
}
```

Output

```
Animal Constructor
Dog Constructor
Dog barks
```

We cannot create: new Animal();
but the constructor of Animal executes when a Dog object is created.

Rules of Abstract Classes

Rule 1 An abstract class is declared using: abstract

Rule 2 An abstract class may contain abstract methods.

Rule 3 An abstract class may contain concrete methods.

Rule 4 An abstract class may contain constructors.

Rule 5 An abstract class may contain variables.

Rule 6 We cannot directly create an object of an abstract class.

Rule 7 A subclass must implement all inherited abstract methods unless the subclass is also abstract.

PREVENTING METHOD OVERRIDING

Why Prevent Overriding?

Sometimes a superclass contains a method whose implementation must not be changed by subclasses.

For example:

```
class Bank
{
    void calculateInterest()
    {
        // important calculation
    }
}
```

If a subclass changes this method, it could produce unwanted behavior.

Java provides the final keyword to prevent method overriding.

final Method

A method declared using final cannot be overridden by a subclass.

Example of final Method

```
class Parent
{
   final void display()
   {
      System.out.println("Parent display method");
   }
}
class Child extends Parent
{
}
```

This is valid.

The child can inherit and use display();

But it cannot redefine the method.

Attempting to Override a Final Method

```
class Parent
{
   final void display()
   {
      System.out.println("Parent");
   }
}
class Child extends Parent
{
   void display()
   {
      System.out.println("Child");
   }
}
```

This produces a compile-time error.

Reason display() is declared as: final

Therefore it cannot be overridden.

PREVENTING INHERITANCE

Introduction to Interfaces, Implementing Interfaces & Interface Reference Variables

An **interface** in Java is a reference type that defines a contract that implementing classes must follow.

It is mainly used to achieve:

- Abstraction
- Multiple inheritance of type
- Loose coupling
- Polymorphism

Definition

An interface is a blueprint that specifies what a class must do, but generally does not specify how it should do it.

For example, suppose we want different classes to provide a print() operation.

We can define:

```
interface Printable
{
   void print();
}
```

Here:

Printable → interface name

void print() → abstract method

The interface specifies that an implementing class must provide print().

Basic Syntax of an Interface

```
interface InterfaceName
{
   // abstract methods
}
```

Example:

```
interface Printable
{
   void print();
}
```

The method `void print();` does not have a body.

It only tells the implementing class.

"You must provide a print() method."

Interface Methods

Traditionally, interface methods without a body are abstract methods.

consider:

```
interface Printable
{
    void print();
}
```

as equivalent to:

```
interface Printable
{
    public abstract void print();
}
```

Therefore

```
void print();
```

implicitly means:

```
public abstract void print();
```

Important Rules of Interfaces

Rule 1: Interface is declared using interface

Rule 2: Interface methods are public and abstract by default

Rule 3: We cannot directly create an object of an interface

Printable p = new Printable(); //This is invalid

because an interface does not provide the complete implementation required to create an object.

Rule 4: A class implements an interface using **implements**

```
class A implements Printable
{
}
```

Rule 5: The implementing class must provide the interface methods

```
class A implements Printable
{
   public void print()
   {
      System.out.println("Printing");
   }
}
```

Why Do We Need Interfaces?

Consider this situation of Printer. There are may brands printers like HPPrinter, CanonPrinter, EpsonPrinter.

All printers should provide print(). So Instead of defining the requirement separately for every class, we can define a common interface:

```
interface Printable
{
   void print();
}
```

Then:

```
class HPPrinter implements Printable
{
   public void print()
   {
      System.out.println("Printing using HP Printer");
   }
}
class CanonPrinter implements Printable
{
   public void print()
   {
      System.out.println("Printing using Canon Printer");
   }
}
```

The interface provides a common contract.

Interface vs Class

Class	Interface
Declared using class	Declared using interface
Objects can normally be created	Cannot directly create interface objects
Can have constructors	Cannot have constructors
Can contain instance variables	Interface fields are implicitly public static final
Can contain concrete methods	Can declare abstract methods
Extended using extends	Implemented using implements
A class can extend one class	A class can implement multiple interfaces

Program 1: Simple Interface

```
interface Printable
{
   void print();
}
class A implements Printable
{
   public void print()
   {
      System.out.println("Printing");
   }
}
public class Main
{
   public static void main(String[] args)
   {
      A obj = new A();
      obj.print();
   }
}
```

Implementing Interfaces

The keyword *implements* is used when a class wants to follow an interface.

Syntax:

```
class ClassName implements InterfaceName
{
   // implementation
}
```

Example:

```
class A implements Printable
{
   public void print()
   {
      System.out.println("Printing");
   }
}
```

Program 2 – Student Interface

```
interface Student
{
   void study();
}
class CollegeStudent implements Student
{
   public void study()
   {
      System.out.println("Student is studying");
   }
}
public class Main
{
   public static void main(String[] args)
   {
      CollegeStudent s = new CollegeStudent();
      s.study();
   }
}
```

Program 3 – Interface with Multiple Methods

An interface can contain multiple methods.

```
interface Vehicle
{
   void start();
   void stop();
}
class Car implements Vehicle
{
   public void start()
   {
      System.out.println("Car started");
   }
   public void stop()
   {
      System.out.println("Car stopped");
   }
}
public class Main
{
   public static void main(String[] args)
   {
      Car c = new Car();
      c.start();
      c.stop();
   }}
```

What Happens If We Do Not Implement All Methods?

Suppose:

```
interface Vehicle
{
   void start();
   void stop();
}
```

Now:

```
class Car implements Vehicle
{
   public void start()
   {
      System.out.println("Car started");
   }
}
```

This produces a compilation error because stop() has not been implemented.

Multiple Interfaces One major advantage of interfaces is that a class can implement multiple interfaces.

Example:

```
interface Printable     {
  void print();              }
interface Showable     {
   void show();              }
class Demo implements Printable, Showable
{         public void print()
   {
      System.out.println("Printing");
   }
   public void show()
   {     System.out.println("Showing");
   }
}
public class Main
{      public static void main(String[] args)
   {
      Demo d = new Demo();
      d.print();
      d.show();
   }     }
```

What is an Interface Reference Variable?

A reference variable whose type is an interface is called an interface reference variable.

Example:

Printable p;

Here:

Printable → reference type

p → reference variable

But we cannot do:

Printable p = new Printable();

because an interface cannot be directly instantiated.

Instead:

Printable p = new A();

where A implements Printable.

Example

```
interface Printable
{
   void print();
}
class A implements Printable
{
   public void print()
   {
      System.out.println("Printing from A");
   }
}
public class Main
{
   public static void main(String[] args)
   {
      Printable p = new A();
      p.print();
   }
}
```

Why Use Interface References?

Interface **references are useful for achieving polymorphism**

Example:

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

Two classes:

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

Now:

```
Animal a;
a = new Dog();
a.sound();
a = new Cat();
a.sound();
```

The same interface reference a refers to different objects.

Interface Reference with Method Parameter

Interface references are especially useful as method parameters.

```
interface Printable
{       void print();
}
class Document implements Printable
{       public void print()
   {
      System.out.println("Printing Document");
   }     }
class Photo implements Printable
{       public void print()
   {
      System.out.println("Printing Photo");
   }    }
public class Main
{     static void performPrint(Printable p)
   {
      p.print();
   }
   public static void main(String[] args)     {
      Document d = new Document();
      Photo ph = new Photo();
      performPrint(d);
      performPrint(ph);
   }     }
```

Assignment

Q. Define an interface Bank with methods deposit() and withdraw(). Implement it using SBI and HDFC classes.

Interface Variables & Extending Interfaces

1. Learning Objectives

By the end of this class, students will be able to:

1. Define interface variables in Java.
2. Explain why variables declared inside interfaces are constants.
3. Understand the implicit modifiers of interface variables.
4. Access interface variables using the interface name.
5. Explain why interface variables cannot be modified.
6. Use interface variables in Java programs.
7. Explain how one interface can extend another interface.
8. Differentiate extends and implements.
9. Implement an extended interface in a class.
10. Demonstrate multiple interface inheritance.
11. Use interface inheritance to achieve abstraction and polymorphism.

Recap of Interfaces

An interface is declared using:

```
interface Printable
{
   void print();
}
```

A class implements an interface using:

```
class Printer implements Printable
{
   public void print()
   {
      System.out.println("Printing");
   }
}
```

An interface reference can refer to an implementing class object:

```
Printable p = new Printer();
```

Now the question is:

Can an interface contain variables?

Yes.

Interface Variables

Consider:

```
interface Test
{
   int x = 10;
}
```

Here, x is an interface variable.

However, an important point is:

Variables declared inside an interface are implicitly public, static, and final.

Therefore:

```
interface Test
{
   int x = 10;
}
```

is equivalent to:

```
interface Test
{
   public static final int x = 10;
}
```

Understanding public static final

Public : The variable can be accessed from anywhere where the interface is accessible.

Test.x

Static: The variable belongs to the interface itself rather than to an object.

Therefore, we do not need to create an object to access it.

Correct:

Test.x

There is no need for:

Test t = new Test();

which is invalid anyway because an interface cannot be instantiated.

final

The value cannot be changed after initialization.

Therefore:

Test.x = 20;

is invalid.

Example

```
interface Test
{
    int x = 10;
}

public class Main
{
    public static void main(String[] args)
    {
        System.out.println(Test.x);
    }
}
```

Interface Variables Are Constants

An interface variable is effectively a constant.

For example:

```
interface Constants
{
   int MAX = 100;
   double PI = 3.14159;
   String COLLEGE = "CV Raman Polytechnic";
}
```

These are all constants.

They are equivalent to:

```
public static final int MAX = 100;
public static final double PI = 3.14159;
public static final String COLLEGE = "CV Raman Polytechnic";
```

```
interface Test
{
    int x = 10;
}

public class Main
{
    public static void main(String[] args)
    {
        System.out.println(Test.x);

        // Test.x = 20;   // Compilation error
    }
}
```

can we write this?

```
interface Test
{
    public static final int x = 10;
}
```

Yes.

This is completely valid.
But normally we write:

```
interface Test
{
    int x = 10;
}
```

Accessing Interface Variables

There are several contexts where the constant can be accessed.

The clearest form is:

Test.x

Example:

```
interface Test
{
   int x = 10;
}
class Demo
{
   void show()
   {
      System.out.println(Test.x);
   }
}
public class Main
{
   public static void main(String[] args)
   {
      Demo d = new Demo();
      d.show();
   }
}
```

Interface Variable Through an Implementing Class

```
interface Test
{
   int x = 10;
}
class Demo implements Test
{
   void show()
   {
      System.out.println(x);
   }
}
public class Main
{
   public static void main(String[] args)
   {
      Demo d = new Demo();
      d.show();
   } // Because x is inherited as an interface constant and can be referred to directly within the implementing class
}.
```

Example: Mathematical Constants

```
interface MathConstants
{
    double PI = 3.14159;
    int ZERO = 0;
}

public class Main
{
    public static void main(String[] args)
    {
        double radius = 5;

        double area = MathConstants.PI * radius * radius;

        System.out.println("Area = " + area);
    }
}
```

Example: Interface Constants in a Program

```
interface College
{
   String NAME = "CV Raman Polytechnic";
   int ESTABLISHED = 2006;
}

public class Main
{
   public static void main(String[] args)
   {
      System.out.println("College: " + College.NAME);
      System.out.println("Established: " + College.ESTABLISHED);
   }
}
```

Rules for Interface Variables

Rule 1 Every interface variable is implicitly public, static, final

Rule 2 It must be initialized when declared.

Correct:

int x = 10;

Rule 3 this is not valid for an interface field, because interface fields are constants and must be initialized.

int x;

Rule 4 It cannot be modified.

Test.x = 20; // Error

Rule 5 It can be accessed without creating an object.

Test.x

EXTENDING INTERFACES

An interface can inherit another interface. This is called **interface inheritance**.

The keyword used is **extends.**

Example:

```
interface A
{
}
interface B extends A
{
}
```

Here:

A → Parent Interface

B → Child Interface

So B inherits the members of A.

Syntax

```
interface ChildInterface extends ParentInterface
{
   // additional members
}
```

Example:

```
interface A
{
   void showA();
}

interface B extends A
{
   void showB();
}
```

Now interface B contains:

showA()
showB()

because B inherits showA() from A.

Implementing the Extended Interface

Consider:

```
interface A
{
   void showA();
}
interface B extends A
{
   void showB();
}
```

Now a class implements B:

```
class C implements B
{
   public void showA()
   {
      System.out.println("Method of A");
   }
   public void showB()
   {
      System.out.println("Method of B");
   }
}
```

```
interface A
{
    void showA();
}

interface B extends A
{
    void showB();
}
class C implements B
{
    public void showA()
    {
        System.out.println("Method of A");
    }
    public void showB()
    {
        System.out.println("Method of B");
    }
}
public class Main
{
    public static void main(String[] args)
    {
        C obj = new C();

        obj.showA();
        obj.showB();
    }
}
```

Interface Can Extend Multiple Interfaces . Unlike classes, an interface can extend multiple interfaces.

Example:

```
interface A
{
   void showA();
}
interface B
{
   void showB();
}
interface C extends A, B
{
   void showC();
}
```

Therefore, C inherits:

showA()

showB()

and additionally declares:

showC()

Multiple Interface Inheritance

```
interface A
{
   void showA();
}
interface B
{
   void showB();
}
interface C extends A, B
{
   void showC();
}
class Demo implements C
{
   public void showA()
   {
      System.out.println("A method");
   }
   public void showB()
   {
      System.out.println("B method");
   }
   public void showC()
   {
      System.out.println("C method");
   }
}

public class Main
{
   public static void main(String[] args)
   {
      Demo d = new Demo();
      d.showA();
      d.showB();
      d.showC();
   }
}
```

Interface Inheritance with Variables

Interface inheritance applies to interface constants as well.

Example:

```
interface A
{
   int x = 10;
}
interface B extends A
{
   int y = 20;
}
```

Now B has access to x and y

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      System.out.println(A.x);
      System.out.println(B.x);
      System.out.println(B.y);
   }
}
```

Example – Student and College

```
interface College
{
    String COLLEGE_NAME = "CV Raman Polytechnic";
    void collegeDetails();
}
interface Student extends College
{
    void studentDetails();
}
class Demo implements Student
{
    public void collegeDetails()
    {
        System.out.println("College: " + COLLEGE_NAME);
    }
    public void studentDetails()
    {
        System.out.println("Student Details");
    }
}

public class Main
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.collegeDetails();
        d.studentDetails();
    }
}
```

Combined Example: Interface Variables + Interface Extension

```
interface Company
{
   String COMPANY_NAME = "ABC Technologies";
   void companyInfo();
}
interface Employee extends Company
{
   int WORKING_HOURS = 8;
   void employeeInfo();
}
class Developer implements Employee
{
   public void companyInfo()
   {
      System.out.println("Company: " + COMPANY_NAME);
   }
   public void employeeInfo()
   {
      System.out.println("Working Hours: " + WORKING_HOURS);
   }
}
public class Main
{
   public static void main(String[] args)
   {
      Developer d = new Developer();
      d.companyInfo();
      d.employeeInfo();
     System.out.println(Company.COMPANY_NAME);

System.out.println(Employee.WORKING_HOURS);
   }
}
```

Arrays and Strings

What is an Array?

An array in Java is an object that stores a fixed number of elements of the same data type. Each element is accessed using an index.

For example, suppose we want to store the marks of five students.

Without an array, we may declare five separate variables:

int m1 = 75;

int m2 = 80;

int m3 = 68;

int m4 = 90;

int m5 = 85;

This approach becomes difficult when we need to store marks for 100 or 1,000 students.

Using an array:

int[] marks = {75, 80, 68, 90, 85};

Now, all five marks can be stored under a single variable name, marks.

Advantages of Arrays

- Store multiple values using a single variable name.
- Access elements using index numbers.
- Simplify repetitive operations using loops.
- Make it easier to perform searching and sorting.
- Organize related data efficiently.

Important Characteristics

Characteristic	**Description**
Data type	All elements have the same declared component type
Size	Fixed after the array is created
Indexing	Starts from 0
Access	Elements can be accessed using an index
Memory	Array elements are stored as part of an array object
Default values	Elements receive default values when an array is created

Important: Java arrays are objects, and their length is fixed after creation.

Declaring an Array

Syntax

dataType[] arrayName;

Example:

int[] numbers;

Here:

int specifies the type of elements.

[] indicates an array.

numbers is the array reference variable.

Java also permits this syntax:

int numbers[];

However, the preferred convention is:

int[] numbers;

Array Declaration vs. Array Creation

Declaring an array reference does not create the array object.

```
int[] numbers;
```

To create an array with space for five integers:

```
numbers = new int[5];
```

Both steps can be combined:

```
int[] numbers = new int[5];
```

The new keyword creates the array object.

Array Memory Representation

Consider:

```
int[] numbers = new int[5];
```

Conceptually, the array contains five integer elements:

Array: numbers

index	0	1	2	3	4
numbers	0	0	0	0	0

The default value of each int element is 0.

Default Values of Array Elements

When an array is created using new, its elements receive default values.

Data Type	Default Value
int	0
float	0.0f
double	0.0
char	'\u0000'
boolean	false
Reference types	null

Example:

```
public class Main {
    public static void main(String[] args) {
        int[] a = new int[3];
        boolean[] b = new boolean[3];

        System.out.println(a[0]);
        System.out.println(b[0]);
    }
}
```

Output:

0

false

Initialization at the Time of Declaration

An array can be initialized directly using values enclosed in curly braces.

```
int[] marks = {75, 80, 68, 90, 85};
```

The array contains five elements.

```
Index:      0    1    2    3    4
Value:    | 75 | 80 | 68 | 90 | 85 |
```

Accessing Array Elements

Array elements are accessed using their index.

Syntax:

```
arrayName[index]
```

Example:

```
int[] marks = {75, 80, 68, 90, 85};
System.out.println(marks[0]);
System.out.println(marks[3]);
```

Output:

```
75
90
```

Modifying Array Elements

Array elements can be modified by assigning a new value to a valid index.

```
int[] marks = {75, 80, 68, 90, 85};
marks[2] = 95;
System.out.println(marks[2]);
```

Output:

95

The original value 68 at index 2 is replaced with 95.

Accessing Array Length

Java provides the length property to determine the number of elements in an array.

```
int[] numbers = {10, 20, 30, 40};
System.out.println(numbers.length);
```

Output:

4

Traversing an Array Using Loops

Array traversal means accessing each element of an array one by one, usually to display or process its values.

Using the for Loop

The for loop is commonly used because array elements are accessed through index values.

Example: Display all elements of an array

```
public class Main {
   public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40, 50};
      for (int i = 0; i < numbers.length; i++) {
         System.out.println(numbers[i]);
      }
   }
}
```

Output:

```
10
20
30
40
50
```

Program 1: Calculate the Sum and Average of Array Elements

Question: Write a Java program to calculate the sum and average of five numbers stored in an array.

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int sum = 0;
        for (int i = 0; i < numbers.length; i++) {
            sum = sum + numbers[i];
        }
        double average = (double) sum / numbers.length;
        System.out.println("Sum = " + sum);
        System.out.println("Average = " + average);
    }
}
```

Output:

Sum = 150

Average = 30.0

Program 2: Find the Largest Element in an Array

Question: Write a Java program to find the largest element in an array.

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {25, 70, 15, 90, 45};
        int max = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }
        System.out.println("Largest element = " + max);
    }
}
```

Output:

Largest element = 90

2-D Array

In Java, a two-dimensional array is commonly used to represent data in rows and columns, such as a matrix, a timetable, or student marks in different subjects.

Need for a Two-Dimensional Array

Suppose marks of three students in two subjects need to be stored.

A 2D array can represent the data as follows:

	Subject 1	Subject 2
Student 1	80	75
Student 2	90	85
Student 3	70	88

This data can be stored using:

```
int[][] marks = {
    {80, 75},
    {90, 85},
    {70, 88}
};
```

Declaration Syntax

dataType[][] arrayName;

Example:

int[][] matrix;

Java also permits:

int matrix[][];

The preferred style is:

int[][] matrix;

Creating a Two-Dimensional Array

int[][] matrix = new int[3][2];

This creates an array with three rows and two columns.

All six elements are initialized to 0.

Declaration and Initialization Together

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

This creates a 3 × 3 array.

Accessing Two-Dimensional Array Elements

Syntax:

arrayName[rowIndex][columnIndex]

Example:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(matrix[0][0]);
System.out.println(matrix[1][2]);
System.out.println(matrix[2][1]);
```

Output:

1

6

8

Modifying a 2D Array Element

matrix[1][2] = 100;

This changes the element in row 1, column 2 to 100.

Traversing a Two-Dimensional Array

A 2D array is commonly traversed using nested loops.

The outer loop traverses the rows. The inner loop traverses the columns.

Program 3: Display a 2D Array

```
public class Main {
   public static void main(String[] args) {
      int[][] matrix = {
         {1, 2, 3},
         {4, 5, 6},
         {7, 8, 9}
      };
      for (int i = 0; i < matrix.length; i++) {
         for (int j = 0; j < matrix[i].length; j++) {
            System.out.print(matrix[i][j] + " ");
         }
         System.out.println();
      }
   }
}
```

Output:

```
1 2 3
4 5 6
7 8 9
```

Write a Java program to add two 2 × 2 matrices.

```
public class Main {
   public static void main(String[] args) {
      int[][] a = { {1, 2},   {3, 4} };
      int[][] b = {  {5, 6}, {7, 8}  };
      int[][] c = new int[2][2];
      for (int i = 0; i < 2; i++) {
         for (int j = 0; j < 2; j++) {
            c[i][j] = a[i][j] + b[i][j];
         } }
      System.out.println("Sum of matrices:");
      for (int i = 0; i < 2; i++) {
         for (int j = 0; j < 2; j++) {
            System.out.print(c[i][j] + " ");
         }
         System.out.println();
      } }}
```

Output:

Sum of matrices:

6 8

10 12

Jagged Arrays in Java

A jagged array is a two-dimensional array in which different rows can have different numbers of elements. Unlike a rectangular 2D array, each row of a jagged array can have a different length.

Example

```
public class Main {
   public static void main(String[] args) {
      int[][] arr = new int[3][];
      arr[0] = new int[2];  arr[1] = new int[3]; arr[2] = new int[4];
      arr[0][0] = 10;  arr[0][1] = 20;
      arr[1][0] = 30; arr[1][1] = 40;  arr[1][2] = 50;
      arr[2][0] = 60; arr[2][1] = 70;  arr[2][2] = 80; arr[2][3] = 90;
      for (int i = 0; i < arr.length; i++) {
         for (int j = 0; j < arr[i].length; j++) {
            System.out.print(arr[i][j] + " ");
         }
         System.out.println();
      } }}
```

Output:

```
10 20
30 40 50
60 70 80 90
```

Common Array Errors

Error	Example	Explanation
Array index out of bounds	arr[5] for an array of length 5	Valid indexes are 0 to 4
Using an uninitialized reference	int[] a; a[0] = 10;	The array has not been created
Invalid array size	new int[-3]	Negative array size causes an exception
Incorrect length syntax	arr.length()	Use arr.length
Accessing a null row	int[][] a = new int[3][]; a[0][0] = 10;	Row a[0] has not been created

Multiple-Choice Questions

Q1. What is the index of the first element of a Java array?

A. 0 B. 1 C. -1 D. 2

Q2. What is the output of the following code?

```
int[] a = {10, 20, 30};
System.out.println(a.length);
```

A. 2 B. 3 C. 10 D. Compilation error

Q3. Which keyword is used to create an array object?

A. Class B. array C. new D. create

Q4. What is the default value of an int array element?

A. Null B. 1 C. -1 D. 0

Q5. Which expression accesses the element in row 2, column 1 of a 2D array?

A. a[2,1] B. a[1][2] C. a[2][1] D. a(2)(1)

Assignment

1. Write a Java program to read 10 integers into an array and display them in reverse order.
2. Write a Java program to count the even and odd numbers in an array.
3. Write a Java program to find the second-largest element in an array.
4. Write a Java program to calculate the sum of diagonal elements of a square matrix.
5. Write a Java program to multiply two matrices.
6. Write a Java program to find the transpose of a matrix.

String in java

A **String** is a sequence of characters.

For example:

"Hello"

"Java Programming"

"CV Raman Polytechnic"

In Java, String is a class present in java. lang package.

Example:

String name = "Rahul";

Here:

String → class/data type

name → reference variable

"Rahul" → String object/value

Creating Strings in Java There are two ways to create a String.

Using a String Literal

```
String str = "Java";   //This is the most common way to create a String.
```

Example:

```
public class Main {
   public static void main(String[] args) {
      String language = "Java";
      System.out.println(language);
   }
}
```

Output:

Java

Using new Keyword A String object can also be created using the new keyword.

```
String str = new String("Java");
```

Example:

```
public class Main {
   public static void main(String[] args) {
      String str = new String("Java");
      System.out.println(str);
   }
}
```

String Indexing

Like arrays, characters in a String are indexed starting from 0.

Consider:

String str = "JAVA";

The characters can be represented as:

Character:	J	A	V	A
	↓	↓	↓	↓
Index:	0	1	2	3

Therefore:

str.charAt(0)

returns: J

and

str.charAt(2)

returns: V

Important String Methods

Java provides many methods for String manipulation.

1.length() Returns the number of characters in the String.

Syntax

string.length()

Example:

```
String str = "Java";
System.out.println(str.length());
```

Output: 4

2. charAt() Returns the character at a specified index.

Syntax

string.charAt(index)

Example:

```
String str = "Java";
System.out.println(str.charAt(0));
System.out.println(str.charAt(2));
```

Output:

J

v

3. toUpperCase() and toLowerCase()

These methods change the case of the characters in the resulting String.

Example

```
public class Main {
    public static void main(String[] args) {
        String str = "Java Programming";
        System.out.println(str.toUpperCase());
        System.out.println(str.toLowerCase());
    }
}
```

Output:

JAVA PROGRAMMING

java programming

4. equals() Method The equals() method compares the contents of two Strings.

Example

```
String s1 = "Java";
String s2 = "Java";
System.out.println(s1.equals(s2));
```

Output: true

If:

```
String s1 = "Java";
String s2 = "java";
```

then:

```
System.out.println(s1.equals(s2));
```

produces: false

because String comparison using equals() is case-sensitive.

5. equalsIgnoreCase()

This method compares two Strings while ignoring differences in uppercase and lowercase.

Example:

```
String s1 = "Java";

String s2 = "java";

System.out.println(s1.equalsIgnoreCase(s2));
```

Output:

true

Difference

equals() → case-sensitive

equalsIgnoreCase() → case-insensitive

6. Comparing Strings Using ==

```
String s1 = "Java";
String s2 = "Java";
System.out.println(s1 == s2);
```

For String literals, this may produce: true because of Java's String pool behavior.

However, consider:

```
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
```

Output:

false

true

Why?

== compares references, whereas equals() compares String contents.

NOTE

To compare the contents of two Strings, use equals() rather than ==.

7. concat() Method The concat() method joins two Strings.

Example:

```
String first = "Hello";
String second = "Java";
String result = first.concat(second);
System.out.println(result);
```

Output:

HelloJava

You can also use the + operator:

```
String result = first + second;
```

Output:

HelloJava

To insert a space:

```
String result = first + " " + second;
```

Output:

Hello Java

8. substring() Method The substring() method extracts part of a String.

Syntax 1 substring(beginIndex)

Syntax 2 substring(beginIndex, endIndex)

Example

```
String str = "Programming";
System.out.println(str.substring(3));
```

Output: gramming

example:

```
String str = "Programming";
System.out.println(str.substring(0, 4));
```

Output: Prog

Index representation:

P r o g r a m m i n g

0 1 2 3 4 5 6 7 8 9 10

Therefore:

substring(0, 4)

selects indexes: 0 1 2 3

Q. Write a Java program to display each character of a String on a separate line.

Program:

```
public class Main {
    public static void main(String[] args) {
        String str = "JAVA";
        for (int i = 0; i < str.length(); i++) {
            System.out.println(str.charAt(i));
        }
    }
}
```

Q. Write a Java program to reverse a String.

```
public class Main {
    public static void main(String[] args) {
        String str = "JAVA";
        String reverse = "";
        for (int i = str.length() - 1; i >= 0; i--)
{
            reverse = reverse + str.charAt(i);
        }
        System.out.println("Original String: " + str);
        System.out.println("Reverse String: " + reverse);
    }
}
```

Q. **Check Palindrome**

A String is called a **palindrome** if it reads the same forward and backward.

```
public class Main {
    public static void main(String[] args) {
        String str = "MADAM";
        String reverse = "";
        for (int i = str.length() - 1; i >= 0; i--) {
            reverse = reverse + str.charAt(i);
        }
        if (str.equals(reverse)) {
            System.out.println("Palindrome");
        } else {
            System.out.println("Not Palindrome");
        }
    }
}
```

String Immutability

One of the most important properties of Java Strings is that String objects are immutable.

Immutable means that once a String object is created, its contents cannot be changed.

Consider:

```
String str = "Java";
str.concat(" Programming");
System.out.println(str);
```

Output: Java

Why?

The concat() operation creates another String; it does not modify the original String.

To store the result:

```
String str = "Java";
str = str.concat(" Programming");
System.out.println(str);
```

Output: Java Programming

Method	Purpose	Example
length()	Returns String length	str.length()
charAt()	Returns character at index	str.charAt(2)
equals()	Compares contents	s1.equals(s2)
equalsIgnoreCase()	Compares ignoring case	s1.equalsIgnoreCase(s2)
toUpperCase()	Converts to uppercase	str.toUpperCase()
toLowerCase()	Converts to lowercase	str.toLowerCase()
concat()	Joins Strings	s1.concat(s2)
substring()	Extracts part of String	str.substring(2,5)

1. Which class is used to represent Strings in Java?

A. Character B. String

C. StringArray D. Text

2. Which method returns the number of characters in a String?

A. size() B. count()

C. length() D. getLength()

3. Which method returns a character at a specified position?

A. get() B. charAt()

C. character() D. index()

4. Which method compares String contents?

A. == B. compare()

C. equals() D. same()

5. Strings in Java are:

A. Mutable B. Immutable

C. Numeric D. Primitive

Wrapper Classes in Java

Learning Objectives

By the end of this lecture, students will be able to:

1. Explain the concept of wrapper classes in Java.
2. Identify wrapper classes corresponding to Java primitive data types.
3. Create wrapper class objects.
4. Understand autoboxing and unboxing.
5. Convert Strings into numeric values using wrapper classes.
6. Use commonly used methods of Integer, Float, Double, etc.
7. Write basic Java programs using wrapper classes.

In java there are 8 primitive data types.

Data Type	Size (bits)	Default Value	Range / Values	Example
byte	8 bits (1 byte)	0	-128 to 127	byte b = 100;
short	16 bits (2 bytes)	0	$-32,768$ to $32,767$	short s = 10000;
int	32 bits (4 bytes)	0	-2^{31} to $2^{31}-1$ ($\approx \pm 2.14 \text{ billion}$)	int i = 100000;
long	64 bits (8 bytes)	0L	-2^{63} to $2^{63}-1$	long l = 10000000000L;
float	32 bits (4 bytes)	0.0f	Single-precision floating point ($\approx 6\text{--}7$ decimal digits)	float f = 5.75f;
double	64 bits (8 bytes)	0.0d	Double-precision floating point ($\approx 15\text{--}16$ decimal digits)	double d = 19.99d;
boolean	1 bit (logical)	false	true or false	boolean isReady = true;
char	16 bits (2 bytes)	'\u0000'	0 to $65,535$ (Unicode characters)	char grade = 'A';

A **wrapper class** is a class that provides an object representation of a primitive data type.

In simple terms:

Wrapper classes "wrap" primitive values inside objects.

For example: int x = 10;

Here x is a primitive int.

The corresponding wrapper class is: Integer

Therefore, we can have: Integer x = 10;

Now x refers to an Integer object containing the value 10.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Why Do We Need Wrapper Classes?

If primitive types are simpler and faster, why do we need wrapper classes?

Wrapper classes are useful because Java has many APIs and collections that work with objects rather than primitives.

For example, a collection such as:

ArrayList<Integer>

stores Integer objects, not primitive int values.

We cannot write:

ArrayList<int> numbers; // Invalid

Instead:

ArrayList<Integer> numbers;

Wrapper classes also provide useful methods for:

- Conversion
- Parsing
- Comparing values
- Finding minimum/maximum values
- Obtaining constants and type information

The Integer Wrapper Class

Integer is the wrapper class for int.

Example

int a = 100; Integer b = 200;

Here: int → primitive Integer → wrapper class

Creating an Integer Object

Integer x = Integer.valueOf(100);

Example:

```
public class Main {
   public static void main(String[] args) {
      Integer x = Integer.valueOf(100);
      System.out.println(x);
   }
}
```

Important Integer Methods

1. **parseInt()**

Converts a String into a primitive int.

Syntax

Integer.parseInt(String)

Example:

```
public class Main {
    public static void main(String[] args) {
        String str = "100";
        int n = Integer.parseInt(str);
        System.out.println(n + 50);
    }
}
```

Output: 150

2. valueOf()

Integer x = Integer.valueOf("100");

This converts the String "100" into an Integer object.

Autoboxing

Autoboxing is the automatic conversion of a primitive value into its corresponding wrapper object.

Example:

```
int x = 10;
Integer obj = x;
```

Java automatically converts: int → Integer

Conceptually:

```
Integer obj = Integer.valueOf(x);
```

Example

```
public class Main {
   public static void main(String[] args) {
      int number = 50;
      Integer obj = number;
      System.out.println(obj);
   }
}
```

Output: 50

The programmer does not need to explicitly call Integer.valueOf().

Unboxing

Unboxing is the automatic conversion of a wrapper object into its corresponding primitive value.

Example:

Integer obj = 50;

int number = obj;

Java automatically performs the equivalent of:

int number = obj.intValue();

Autoboxing and Unboxing Program

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        // Autoboxing
        Integer obj = a;
        // Unboxing
        int b = obj;
        System.out.println("Primitive value: " + a);
        System.out.println("Wrapper object: " + obj);
        System.out.println("Unboxed value: " + b);
    }
}
```

Output:

Primitive value: 10

Wrapper object: 10

Unboxed value: 10

Float Wrapper Class

The wrapper class for float is Float.

Example:

```
float x = 10.5f;
Float obj = x;
```

This is autoboxing.

Parsing a Float

```
String str = "25.75";
float value = Float.parseFloat(str);
System.out.println(value);
```

Output:

25.75

Double Wrapper Class

The wrapper class for double is Double.

Example:

```
double price = 99.99;
Double obj = price;
```

Parsing a Double

```
String str = "99.99";
double value = Double.parseDouble(str);
System.out.println(value);
```

Output: 99.99

Character Wrapper class

Character

Example:

```
char ch = 'A';
Character obj = ch;
```

Useful methods:

```
Character.isLetter(ch)
Character.isDigit(ch)
Character.isUpperCase(ch)
Character.isLowerCase(ch)
```

Example:

```
public class Main {
   public static void main(String[] args) {
      char ch = '7';
      System.out.println(Character.isDigit(ch));
   }
}
```

Output: true

Boolean Wrapper class:

Example:

```
boolean result = true;
Boolean obj = result;
System.out.println(obj);
```

Output:

true

Wrapper	Useful Method	Purpose
Integer	parseInt()	String → int
Integer	valueOf()	Value → Integer
Integer	intValue()	Integer → int
Float	parseFloat()	String → float
Double	parseDouble()	String → double
Long	parseLong()	String → long
Short	parseShort()	String → short
Byte	parseByte()	String → byte
Boolean	parseBoolean()	String → boolean
Character	isDigit()	Checks whether character is digit
Character	isLetter()	Checks whether character is letter

Q. Write a Java program to read a number stored as a String and convert it into an integer.

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

        String str = "250";

        int number = Integer.parseInt(str);

        System.out.println("Number = " + number);
        System.out.println("Number + 50 = " + (number + 50));
    }
}
```

Q. Find Larger Integer

```
//Wrapper classes also provide useful comparison methods.

public class Main {
    public static void main(String[] args) {
        Integer a = 50;
        Integer b = 75;

        if (a.compareTo(b) > 0) {
            System.out.println("a is larger");
        } else if (a.compareTo(b) < 0) {
            System.out.println("b is larger");
        } else {
            System.out.println("Both are equal");
        }
    }
}
```

Main Difference

Primitive

↓

Stores a value directly

Wrapper

↓

Represents the primitive value as an object

Wrapper Objects Are Immutable

Wrapper objects such as Integer, Float, and Double are immutable.

For example:

Integer x = 10;

x = 20;

This does not modify the original Integer object containing 10. The variable x is made to refer to another value/object.

1. Which is the wrapper class of int?

A. Int B. Integer

C. Number D. Long

2. Which is the wrapper class of float?

A. Float B. Floating

C. Double D. Real

3. Which method converts a String to an integer?

A. Integer.convert() B. Integer.value()

C. Integer.parseInt() D. Integer.toInt()

4. What is the automatic conversion of int to Integer called?

A. Unboxing B. Autoboxing

C. Casting D. Parsing

5. What is the automatic conversion of Integer to int called?

A. Boxing B. Autoboxing

C. Unboxing D. Parsing