Faculty Name: C. Meghana

Subject Name: Object Oriented Programming using Java

Unit Name: Introduction to Java Programming

Department Name: AI & ML

Unit - I

Introduction to Java Programming

⇒ Introduction to Java

Java is a high-level, object-oriented, and platform-independent programming language developed by James Gosling at Sun micro systems in 1995. It is widely used for building desktop applications, mobile apps (especially Android), web applications, and enterprise software.

High level language:

A high level language (HLL) is a programming language that is designed to be easy for humans to read, write and understand, using English-like syntax and words.

Object-oriented programming:

Object-oriented programming (OOP) is a programming paradigm based on the concept of objects, which contain data (attributes) and methods (functions) that operate on that data.

Platform Independent:

Platform Independent means a program can run on any operating system or hardware without modification.

* Java follows the WORA (Write Once, Run Anywhere) principle.
* It means "You write a Java program once, and you can run it on any computer or operating system - without changing the code."

⇒ History of Java

* In 1991, a team of engineers known as the Green Team led by "James Gosling" at "Sun Micro Systems" started a project to develop

Software for consumer electronic devices, like TVs, remote controls, etc.

* The goal was to create a platform independent language that could run on various devices regardless of their hardware.
* The first version of this language was named "Oak", after an oak tree that stood outside Gosling's office. The name "Oak" was later changed to "Java" because Oak was already a registered trademark.
* The name Java was choosen from a list of suggestions, inspired by Java coffee from Indonesia. The name reflects energy, vitality, and productivity - which matched the team's goals.
* In 1995, Java 1.0 was officially released. It was designed with the philosophy of "Write Once Run Anywhere" (WORA), meaning compiled Java code could run on any device with a Java Virtual Machine (JVM).
* In 2009, "Oracle Corporation" acquired Sun micro systems. Oracle became the official owner and maintainer of Java, continuing to release versions under a new, faster release cycle (every 6 months).
* The latest version of Java (Standard Edition) is Java SE 24, released on March 18, 2025, with the most recent update being Java 24.0.1, published on April 15, 2025.

⇒ Evaluation of Java

version	year	key features
Java 1.0	1995	Basic version with AWT, Applets
Java 1.1	1997	Inner classes, JDBC, RMI
Java 2 (J2SE 1.2)	1998	Collections Framework, Swing, JVM improvements
J2SE 1.3	2000	HotSpot JVM, better performance

Version	Year	Key Features
J2SE 1.4	2002	Assert keyword, NIO, exception chaining
Java 5 (1.5)	2004	Generics, Metadata (Annotations), Enums, Autoboxing
Java 6	2006	Improved web services, Scripting API
Java 7	2011	try-with-resources, diamond operator, NIO.2
Java 8	2014	Lambda expressions, Streams API, Functional interfaces
Java 9	2017	JPMS (Modules), JShell (REPL)
Java 10	2018	LTS version, Local variable type inference (var)
Java 11	2018	LTS version, new HTTP client, removal of Applets
Java 12-16	2019-2021	Incremental enhancements
Java 17	2021	LTS version, Sealed classes, pattern matching
Java 21	2023	Latest LTS version, more previews features

⇒ Types of Java Editions

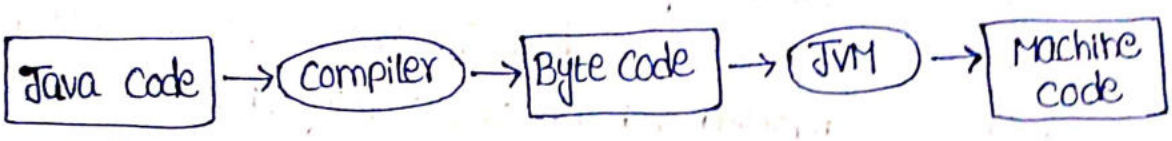
Editions	Full Name	Use case
Java SE	Java Standard Edition	General-purpose programming
Java EE	Java Enterprise Edition (Now Jakarta EE)	Large-scale, enterprise-level apps
Java ME	Java Micro Edition	Embedded devices, IoT, mobile

⇒ Byte Code

Byte code can be defined as an intermediate code that is generated by the Java compiler (javac) after the Java Source Code (.java file) is compiled.

* Java Source code is compiled into byte code, not into machine-specific code.
* This byte code is then interpreted or compiled by the JVM, which is available for different platforms (windows, Linux, macOS, etc).

* Therefore, the same .class file can run on any device that has a compatible JVM.



Why Java is popular:

* Platform- independent (runs on Windows, Mac, Linux)
* Strong community support
* Secure and reliable
* Used for Web development, Android Apps, Enterprise Applications, IOT, etc.

⇒ Java Buzzwords [or] Features of the Java

A list of the most important features of the Java language is given below.

1. Simple
2. Object-oriented
3. platform independent
4. Secured
5. Robust
6. Architecture neutral
7. portable
8. High performance
9. Distributed
10. Multi threaded
11. Dynamic
12. Interpreted

1. Simple:

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Micro System, Java language is a simple programming language because:

* Java Syntax is based on C and C++ (So easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage collection in Java.

2. object-oriented

OOP stands for Object-oriented programming. OOP is a programming paradigm in which every program is follows the concept of object. In other words, OOP is a way of writing programs based on the object concept.

Basic concepts of OOPS are:

1. Object
2. class
3. Inheritance
4. polymorphism
5. Abstraction
6. Encapsulation

3. platform independent

platform independent means that a Java program can run on any operating system (windows, Linux, macOS, etc) without modifying the source code. Java is platform independent because it doesn't produce machine-specific code. Instead, it produces byte code, which can run on any system that has the JVM.

4. Secured:

Java is known for its strong security. It helps create virus-free and secure applications. Here's why Java is considered secure:

* No pointers: Java doesn't use pointers, so it's harder for harmful code to access memory directly.
* Runs in a Sandbox: Java programs run inside a virtual machine (JVM), which keeps them separate from the system.
* Class Loader: It loads classes separately based on where they come from (local or network), adding a layer of security.
* Bytecode verifier: It checks the code before it runs to make sure nothing illegal or harmful is there.
* Security Manager: It controls what a Java program can do, like reading or writing files.

All these features help make Java secure by default.

5. Robust:

The English mining of Robust is strong. Java is robust because:

* It uses strong memory management.
* Java provides automatic garbage collection, which runs on the Java virtual machine to get rid of objects which are not being used by a Java application anymore.
* There are Exception handling and the type checking mechanism in Java. All these points make Java robust.

6. Architecture neutral:

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. Java occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

7. Portable:

Java is portable because it facilitates you to carry the Java byte code to any platform. It doesn't require any implementation.

8. High performance:

Java is faster than other traditional interpreted programming languages because Java byte code is "close" to native code. It is still a little bit slower than a compiled language (eg: C++). Java is an interpreted language that is why it is slower than compiled languages, eg: C, C++, etc.

9. Distributed:

Java is distributed because it facilitates users to create distributed applications is in Java. RMI (remote method invocation) and EJB (Enterprise java bean) are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

10. Multi-threaded:

A thread is like a separate program, executing concurrently, we can write java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, web applications, etc.

11. Dynamic:

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand.

* It also supports functions from its native languages, i.e., C and C++.
* Java supports dynamic compilation and automatic memory management (garbage collection).

12) Interpreted:

Java is a interpreter language. It is convert byte code to Executable code instructions in JVM. It is design to read the input source code and then translate the executable code instruction by instruction.

⇒ Java SE 8 (Standard Edition 8)

Released: March 2014

Java SE.8 is one of the most important versions of Java. It introduced many powerful features that made coding easier, cleaner, and more efficient.

Key Features of Java SE 8:

1. Lambda Expressions
   * Allows writing short and simple functions.
2. Functional Interfaces
   * An interface with only one abstract method
   * Used with lambda Expressions.
   * Example: Runnable, Comparator
3. Stream API:
   * Used to process collections (like list or Set) in a functional style.
   * Helps in filtering, mapping, and collecting data easily.
4. Default and Static Methods in Interfaces:
   * You can now write methods with code inside interfaces.
   * default - provides default implementation.
   * static - can be called using interface name.
5. Date and Time API (java.time):
   * A new and improved API to handle date and time easily.
6. Optional Class:
   * Used to avoid NullPointerException

* Wraps a value that might be null.

7) Nashorn JavaScript Engine:
   * Runs JavaScript code inside Java applications.

⇒ Object-oriented programming - Two paradigms

Programming paradigm:

A programming paradigm is a style or way of programming. It defines how a problem is approached and solved using programming languages.

Two paradigms in OOP

* procedure-oriented programming (POP)
* object-oriented programming (OOP)

1. procedure-oriented programming (POP):

Definition: A programming paradigm based on the concept of functions or procedures. The main focus is on writing a sequence of instructions to perform a task.

Key characteristics:

* Divides the program into functions.
* Emphasizes algorithms and steps.
* Data is global and shared among functions.
* Top-down approach.

Examples: C, Pascal, BASIC

Example (in C):

```
Void main() {
    int a=10, b=20;
    printf("%d", a+b);
}
```

2. Object-oriented programming (OOP):

Definition: A paradigm where a program is built using objects, which are instances of classes. The focus is on real-world entities and data encapsulation.

Key Characteristics:

* Divides the program into objects (data+behaviour).
* Emphasizes data hiding and reuse.
* Uses bottom-up approach.
* promotes modularity, and maintainability.

Example: Java, C++, Python

Core Concepts of OOP:

Concept	Description
class	Blue print for creating objects
object	Instance of a class
Encapsulation	Hiding internal data using access modifiers.
Abstraction	Showing essential features, hiding complexity
Inheritance	Reusing code from parent class
Polymorphism	One interface, many forms (method overloading/overriding)

Example (in Java):

```
class Calculator {
    int add(int a, int b) {
        return a+b;
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator c= new Calculator();
```

```
        System.out.println(c.add(5,10));
    }
}
```

⇒ Abstraction

Definition:

Abstraction is the process of hiding the internal implementation details and showing only the essential features of an object. It helps in reducing complexity by letting the user focus on what an object does, instead of how it does it.

Real-Life Example:

Think of a car -- when you drive, you use the steering, accelerator, and brakes. You don't need to know how the engine works internally. That's abstraction — showing only necessary parts and hiding the rest.

Abstraction in Java:

Java provides abstraction in two main ways:

Way	Description
1. Abstract Class	A class that cannot be instantiated and may contain abstract methods (no body).
2. Interface	A fully abstract type (before Java 8), now supports default and static methods too.

Abstract Class Example.

```
abstract class Animal {
    abstract void makeSound(); // abstract method
    void sleep() {
        System.out.println("Sleeping...");
    }
}
```

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

* makeSound() is abstract — each subclass will implement it differently.

Interface Example:

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

* Interface defines what needs to be done, not how.

Benefits of Abstraction:

* Simplifies complex systems
* Increases code reusability
* Enhances maintainability
* Improves security by hiding internal logic.

⇒ The three OOP principles

Java follows the core concepts of object-oriented programming. Out of the four main pillars (Encapsulation, Inheritance, Polymorphism, Abstraction), three principles commonly highlighted are:

1. Encapsulation:

Definition: Encapsulation is the process of wrapping data (variables) and methods (functions) together into a single unit (class). It restricts

direct access to some components, improving security and modularity.

How it's done in Java:

* Declare variables as private
* provide getter and setter methods

Example:

```
class Student {
    private int marks;
    public void setMarks(int m){
        marks = m;
    }
    public int getMarks(){
        return marks;
    }
}
```

* Here, marks is hidden and accessed only via methods = Encapsulation

2. Inheritance

Definition:

Inheritance is a mechanism where one class (child / subclass) can acquire the properties and behaviors (methods) of another class (parent/ superclass). It promotes code reusability.

How it's done in Java:

* Use the extends keyword

Example:

```
class Animal {
    void eat(){
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
```

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

* Dog class inherits eat() method from Animal.

3. Polymorphism

Definition: polymorphism means "many forms". In Java, a method can behave differently based on the object that is calling it.

Types:

* Compile-time polymorphism (Method overloading)
* Runtime polymorphism (Method overriding)

Example – Method overloading

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

Example – Method overriding

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

⇒ A First Simple Java program

Java programs go through three basic steps:

1. Entering the program (writing the Code)

You can write your Java program using any text editor like:

* Notepad (Windows)
* Notepad ++ / VS code
* IDEs like IntelliJ, Eclipse, NetBeans

Example program:

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

key points:

* the file name must match the class name (Hello.java)
* the entry point of the program is the main() method.

2. Compiling the program

You compile the program using the Java Compiler (javac command).

Command:

```
javac Hello.java
```

What happens:

* The Java compiler checks for syntax errors.
* If there are no errors, it creates a byte code file: Hello.class
* This byte code file is platform-independent and can be run on any system with a JVM.

3. Running the Program

You run the compiled byte code using the Java Virtual Machine (Java Command):

Command:

java Hello

Output: Hello, Java!

⇒ Variables.

A variable in java is a named memory location used to store data. Every variable has a type, which determines the size and layout of the variable's memory.

Types of variables in Java:

1. Local variable

A local variable is declared inside a method or a block, and it is accessible only within that method or block. These variables are created when the method is called and destroyed when the method ends.

Example:

```
void display() {
    int x=10; // local variable
    System.out.println(x);
}
```

2. Instance Variable

An instance variable is declared inside a class but outside any method. It is not static and belongs to each object of the class. Each object has its own copy of the instance variable.

Example:

```
public class Student {
    int age =20; //instance variable
}
```

3. Static Variable

A static variable is declared using the static keyword inside a class but outside methods. It is shared among all instances of the class and belongs to the class itself, not individual objects.

Example:

```
Public class Student {
    static int count = 0; // static variable
}
```

Declaration and Initialization:

```
int number = 5; // declaration and initialization
```

[or]

```
int number; // declaration
number = 5; // initialization
```

Syntax of Variable Declaration

```
<data_type> <variable_name> = <value>;
```

Example:

```
int age = 20;
float marks = 85.5f;
Char grade = 'A';
String name = "Meghana";
```

Rules for naming Variables:

* Must begin with a letter, $, or _
* Cannot start with a digit
* Cannot use Java reserved keywords (eg: int, class)
* Case-sensitive (name and Name are different)

Default Values:

Data type	Default value
int	0
float	0.0f

boolean	false
char	'\u0000'
object(eg. String)	null

* Local variables do not have 'default' values - they must be initialized before use.

Example: Different variables

```
Public class Student {
    int rollNumber; // instance variable
    Static String college = "ABC college"; // static variable
    void display() {
        String name = "Megha"; // local variable
        System.out.println(name + " " + rollNumber + " " + college);
    }
}
```

⇒ Data types:

In Java, data type specify the size and type of values that can be stored in a variable. Java is a statically-typed language, meaning all variables must be declared with a data type.

Types of Data Types:

Java supports different types of data to store various kinds of values. These data types are broadly classified into two main categories:

1. Primitive Data types

primitive Data types are the basic built-in data types provided by Java. They are predefined by the language and used to represent simple values such as numbers, characters, or boolean values. Java provides eight primitive data types:

* byte: used to store small integer values, mainly to save memory.
* Short: stores a slightly larger range of integers than byte.
* int: the default data type for integers; widely used.
* long: used to store large integer values, such as phone numbers.
* float: used to store decimal numbers with single precision.
* double: stores decimal values with double precision; more accurate than float.
* char: used to store a single Unicode character (eg: 'A', '1', '$').
* boolean: represents only two values — true or false, used for logical operations.

Each of these types has a fixed memory size and stores the actual value directly.

byte

* Size: 1 byte (8 bits)
* Range: −128 to 127
* use: Saves memory in large arrays when memory is limited
* Default value: 0
* Example: byte age=25;

Short

* Size: 2 bytes (16 bits)
* Range: −32,768 to 32,767
* use: Used instead of int when memory is limited.
* Default value: 0
* Example: short year = 2025;

int

* Size: 4 bytes (32 bits)
* Range: −2,147,483,648 to 2,147,483,647
* use: Default data type for integers
* Default value: 0
* Example: int Salary = 50000;

long:
* Size: 8 bytes (64 bits)
* Range: very large (±9 quintillion)
* Use: For large integer values (eg: phone numbers)
* Default value: 0L
* Example: long mobileNumber = 9876543210L;

float:
* Size: 4 bytes (32 bits)
* precision: 6-7 decimal digits
* Use: For decimal values with less precision
* Suffix: f or F
* Default value: 0.0f
* Example: float percentage = 87.5f;

double:
* Size: 8 bytes (64 bits)
* precision: 15 decimal digits
* Use: Default for decimal values (higher precision than float)
* Default value: 0.0d
* Example: double price = 9999.99;

Char
* Size: 2 byte (16 bits)
* Stores: A single Unicode character (like 'A', '1', '#')
* Range: 0 to 65,535
* Default value: '\u0000' (null character)
* Example: char grade = 'A';

boolean:
* Size: 1 bit (JVM-dependent)
* values: true or false
* Use: For logical operations and decision making
* Default value: false

Example: boolean isPassed = true;

2. Non-primitive Data types

Non-primitive data types, also known as reference types, refer to objects and can store more complex structures. Unlike primitive types, they do not store the actual value but store the reference (address) of the object in memory.

Examples of non-primitive data types include:

String: Represents a sequence of characters (text). Strings are objects in Java. Strings are Immutable (cannot be changed once created).

Example: String name = "Meghana";

Arrays: A collection of similar data elements stored in a continuous memory location. Fixed in size once declared.

Example: int[] marks = {90, 80, 70};

Classes: A blueprint for creating objects. It defines methods and variables.

Example:

```
class Student {
    int id;
    String name;
}
Student S = new Student();
```

Interfaces: A reference type that can contain abstract methods. It is used for abstraction and multiple inheritance.

Example:

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

Non-primitive types are user-defined or class-based and can be used to build complex programs and data structures.

⇒ Arrays in Java

An array in java is a container object that holds a fixed number of values of the same data type. It is used to store multiple values in a single variable, instead of declaring seperate variables for each value.

key features:

* Fixed Size: The size of an array is defined when it is created and Cannot be changed later.
* Indexed: Each element is accessed using its index, starting from 0.
* Homogeneous elements: All elements in an array must be of the same data type.
* Memory-efficient: Stored in contiguous memory locations.

Types of Arrays in java:

Java provides support for two main types of arrays:

1. Single Dimensional Array
2. Multi Dimensional Array

1. Single Dimensional Array

A Single Dimensional Array is a linear array where elements are stored in a single row. It is like a list of elements, and each element is accessed using a single index.

Characteristics:

* Stores multiple values of the same data type.
* Index starts from 0.
* All elements are stored in contiguous memory locations.
* It is the most commonly used type of array in Java.

Declaration & initialization:

```
int[] marks = new int[5]; // Declaration + Memory allocation
int[] ages = {18,19,20,21, 22}; // Declaration + Initialization
```

Accessing Elements:

```
System.out.println (ages[2]); // output : 20
```

Example program:

```
Public class OneDArray {
    Public static void main(String[] args){
        int[] numbers = {10,20,30,40};
        for (int i=0; i<numbers.length; i++){
            System.out.println("Element at index " +i+ ":" +numbers[i]);
        }
    }
}
```

2. Multi-Dimensional Array

A multi-dimensional is an array that contains more than one row or column. It is essentially an array of arrays. The most commonly used form is the two dimensional (2D) array, which can represent data in the form of tables or matrices.

Characteristics:

* useful for storing tabular data, such as marks of students in different subjects.
* Requires multiple indices to access each element – for 2D arrays: one for row and one for column.
* can be extended to 3D, 4D arrays etc., through less common.

Declaration & Initialization:

```
int[][] matrix = new int[2][3]; //2 rows, 3 columns
int[][] values = {
    {1,2,3},
    {4,5,6}
};
```

Accessing Elements

```
System.out.println(values[1][2]); // output: 6 (2nd row, 3rd column)
```

Example program:

```
Public class TwoDArray {
    Public Static void main (String[] args) {
        int[][] matrix = {
            {10,20},
            {30,40}
        };
        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();
        }
    }
}
```

Advantages of Arrays:

* Easy to manage and access multiple data items using a loop
* Improves code readability
* Supports random access using index
* Useful in Sorting, Searching, and matrix operations.

Limitations of Arrays:

* Fixed size (cannot increase or decrease after creation)
* Can hold only elements of the same data type
* Not dynamic like collections (eg: ArrayList)

⇒ Operators

In java, operators are special symbols or keywords used to perform operations on variables and values. These operations may include arithmetic, comparison, assignment, logical decisions and more.

Java operators are classified into several categories based on the type of operation they perform.

1. Arithmetic Operators

used to perform basic mathematical operations

operator	Meaning	Example	Result
+	Addition	10+5	15
-	Subtraction	10-5	5
*	Multiplication	10*5	50
/	Division	10/5	2
%	Modulus	10%3	1

Example:

```
int a=10, b=5;
System.out.println(a+b); // output: 15
```

2. Relational (Comparison) Operators

used to compare two values. The result is always true or false.

operator	Meaning	Example	Result
==	Equal to	5==5	true
!=	Not equal to	5!=3	true
>	Greater than	5>3	true
<	less than	3<5	true
>=	Greater than or equal	5>=5	true
<=	Less than or equal	4<=5	true

Example:

```
int x=10; y=20;
System.out.println(x<y); // output: true
```

3. Logical Operators

used to combine multiple conditions or boolean expressions.

operator	Description	Example
&&	Logical AND (both must be true)	(a>b) && (a>c)
||	Logical OR (at least one is true)	(a>b) || (a>c)
!	Logical NOT (reverses condition)	!(a>b)

Example:

```
int a=5, b=10;
System.out.println(a<b && b>0); //output: true
```

4. Assignment Operators

Used to assign values to variables.

operator	Meaning	Example	Equivalent To
=	Assign	a=5	a=5
+=	add and assign	a+=3	a=a+3
-=	Subtract and assign	a-=2	a=a-2
*=	Multiply and assign	a*=4	a=a*4
/=	Divide and assign	a/=2	a=a/2
%=	Modulus and assign	a%=2	a=a%2

Example:

```
int a=10;
a+=5;
System.out.println(a); // output:15
```

5. Unary operators

Used with only one operand.

operator	Meaning	Example	Result
+	Unary plus (positive value)	+a	+a
-	Unary minus (negative value)	-a	-a
++	Increment	a++ (or) ++a	a+1
--	decrement	a-- (or) --a	a-1

!	Logical NOT	!true	false

Example:

```
int a=5;
System.out.println(++a); //output: 6
```

6. Bitwise operators:

Used to perform operations on bits.

operator	Meaning	Example
&	Bitwise AND	a&b
|	Bitwise OR	a|b
^	Bitwise XOR	a^b
~	Bitwise Complement	~a
<<	Left Shift	a<<2
>>	Right Shift	a>>2

Example:

```
int a=5, b=3;
System.out.println(a&b); // output:1
```

7. Ternary operator:

This is a conditional operator that works like a shortcut for if-else.

Syntax:

```
condition ? expression1 : expression2;
```

Example:

```
int a=10 ; b=20;
int max = (a>b) ? a:b;
System.out.println(max); // output: 20
```

8. instanceof Operator:

Used to check whether an object is an instance of a specific class or subclass.

Example:

```
String str = "Hello";
System.out.println(str instanceof String); //output: true
```

Control Statements

Control statements in Java are used to control the flow of execution in a program. They help in making decisions, repeating tasks, or jumping to specific points within the code. These statements increase the efficiency and flexibility of the program.

Types of Control Statements

Java control statements can be classified into three main categories:

* Decision-Making Statements
* Looping Statements
* Jump Statements
* Jump Statements

1. Decision-Making Statements (Conditional)

These statements execute certain parts of code based on conditions. Java supports several types:

a) if Statement

Executes a block of code only if the specified condition is true.

Syntax:

```
if (condition) {
    // code block
}
```

Example:

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

b) if-else statement

Chooses between two blocks of code based on a condition.

Syntax:

```
if(condition) {
    // block 1
}
else {
    // block 2
}
```

Example:

```
if num=5;
int num=5;
if(num%2==0) {
    System.out.println("Even number");
}
else {
    System.out.println("odd number");
}
```

c) if-else-if Ladder

Checks multiple conditions in sequence.

Syntax:

```
if(condition1) {
    // block1
} else if (condition2) {
    // block2
} else {
    // default block
}
```

Example:

```
int marks =85;
if(marks >=90) {
    System.out.println("Grade A");
}
```

```
else if (marks >=75) {
    System.out.println ("Grade B");
}
else {
   System.out.println (" Grade C");
}
```

d) Nested if Statement

A nested if means putting one if statement inside another if. Used when the second condition depends on the first condition.

Syntax:

```
if (condition1) {
    if (condition2) {
        //code runs only if both condition1 and condition2 are true
    }
}
```

Example:

```
int age=20;
int marks =80;
if (age >=18) {
    if(marks >= 75) {
        System.out.println("Eligible");
    }
}
```

e) Nested if-else Statement

A nested if-else means using if-else statements inside another if or else block. Used to make more detailed decisions.

Syntax:

```
if (condition1) {
    if (condition2) {
        // Code if both Conditions are true
    } else {
        //Code if Condition1 is true but condition2 is false
    }
```

```
} else {
    //code if condition1 is false
}
```

Example:

```
int masks = 65;
if(masks >=50) {
    if (masks >=75) {
        system.out.println(" Distinction");
    } else {
        System.out.println (" Passed ");
    }
}
else {
    System.out.println (" Failed");
}
```

f) Switch Statement

Used when you have multiple options based on the value of a single variable. Works with int, char, String, enum, and wrapper classes like Integer, Character.

Syntax:

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

Example:

```
int day = 3;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    default: System.out.println("Invalid day");
}
```

2. Looping Statements (Iteration)

Looping statements are used to execute a block of code repeatedly.

a) for loop

used when number of iterations is known.

Syntax:

```
for(initialization; condition; inc/dec) {
    // loop body
}
```

Example:

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

b) while loop

used when condition must be checked before entering the loop.

Syntax:

```
while (condition) {
    // loop body
}
```

Example:

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

c) do-while loop

Executes the block at least once, then checks the condition.

Syntax:

```
do {
    // loop body
} while (condition);
```

Example:

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

3. Jump Statements

used to alter the normal flow of control in a program.

a) break Statement

Terminates the loop or switch block prematurely.

often used inside loops and switch-case

Example:

```
for(int i=1; i<=10; i++){
    if (i==5) {
        break;
    }
    System.out.println(i);
}
```

b) continue Statement

Skips the current iteration and proceeds with the next one.

Example:

```
for(int i=1; i<=5; i++){
    if(i==3) {
```

```
        continue;
    }
    System.out.println(i);
}
```

c) return Statement

Exits from the current method and optionally returns a value. Used in methods to return results.

Example:

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

⇒Class Fundamentals

A class in java is a blueprint or template for creating objects. It defines variables(fields) and methods that describe the behaviour and state of an object.

Key points:

* A class is a user-defined data type.
* It can contain fields (variables) and methods (functions).
* Classes help in organizing code using object-oriented programming (OOP).
* You can create multiple objects from a single class.

Syntax of a class

```
class ClassName {
    // Data Members (fields or variables)
    dataType variable1;
    dataType variable2;
    //Member Functions (methods)
    returnType methodName() {
        // Method body
    }
}
```

Explanation:

* class is the keyword to define a class.
* ClassName is the name you give to the class(must start with a capital letter by convention).
* Inside the class, you define variables and methods.

Example:

```
//Class definition
class Student {
    //Data Members (fields)
    int rollNo;
    String name;
    // Method to display data
    void displayInfo() {
        System.out.println("Roll Number:"+rollNo);
        System.out.println("Name:"+name);
    }
}
//Main class
public class StudentDemo {
    public static void main(String[] args) {
        // Creating object of class Student
        Student S1 = new Student();
        //Assigning values to object
        S1.rollNo = 101;
        S1.name = "Meghana";
        // Calling method using object
        S1.displayInfo();
    }
}
```

Example:

```
class Student {
    int rollNo;
    String name;
    void displayInfo() {
        System.out.println("Roll No: " + rollNo);
        System.out.println("Name: " + name);
    }
}
```

Benefits of Using classes

* Reusability: Code can be reused by creating multiple objects.
* Modularity: Code is easier to manage and understand.
* Encapsulation: Data and methods are bundled together.
* Scalability: Easy to expand and maintain.

⇒Declaration of objects

Object definition: An object is an instance of a class. It represents a real-world entity that has state (attributes) and behavior (methods).

Object declaration

To create an object in Java, we use the new keyword.

Syntax:

```
ClassName objectName = new ClassName();
```

Example:

```
Student s1 = new Student();
```

Here: * Student is the class
* s1 is the object reference
* new Student() creates a new object in memory.

How it works:

* new allocates memory.
* Student() is a constructor that initializes the object
* S1 holds the reference to the newly created object.

⇒ Assigning object Reference variables

Definition: In Java, object reference variables store the memory address (reference) of the object, not the actual object itself.

Assigning Object References:

You can assign one object reference to another, meaning both will point to the same object in memory.

Example:

```
Student S1 = new Student();
Student S2 = S1;
```

Now: * S2 also refers to the same object that S1 refers to
* Changing S2's data also affects S1.

```
S2.name = "John";
System.out.println(S1.name); // O/P: John
```

* Both references S1 and S2 point to the same memory location.
* No new object is created when you assign like this.
* Changes made using one reference will be reflected when accessed via the other reference.

⇒ Introducing Methods

Definition:

A Method is a block of code or a function that performs a specific task. Methods are used to define the behavior of an object.

Purpose of Methods:

* To organize code into reusable blocks.

* To avoid code duplication
* To enhance readability and maintainability

Syntax:

```
returnType methodName() {
    // method body
}
```

Ex:

```
void displayMessage() {
    System.out.println("Hello, Java!");
}
```

⇒ Adding a Method to the class

A method is added inside the class but outside other methods. It is accessed using objects.

Example:

```
class Student {
    String name;
    // method without return and parameters
    void showName() {
        System.out.println("Student Name: "+name);
    }
}
public class Demo {
    Public Static void main(String[] args) {
        Student s = new Student();
        s.name = "Meghana";
        s.showName(); // Method call
    }
}
```

⇒ Returning a value

A method can return a value using the return keyword. the return type must match the declared type.

Syntax:

```
returnType methodName() {
    //Logic
    return value;
}
```

Example:

```
class MathOperations {
    int add() {
        int a=5, b=3;
        return a+b;
    }
}
public class Main {
    public static void main (String[] args) {
        MathOperations m = new MathOperations();
        int result = m.add();
        System.out.println("Sum = " + result);
    }
}
```

⇒ Adding a Method that takes parameters

Methods can take input values (called parameters or arguments) to work dynamically with data.

Syntax:

```
returnType methodName (dataType param1, dataType param2) {
    // method body
}
```

Example:

```
class Calculator {
    int multiply(int a, int b) {
        return a*b;
    }
}
public class Demo {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        int product = c.multiply(4,5);
        System.out.println("Product = "+product);
    }
}
```

⇒ Constructors

Definition:

A constructor is a special method that is automatically called when an object is created. It is used to initialize objects.

Key features:

* Constructor name must match the class name.
* It has no return type, not even void.
* It is called automatically when an object is created.

Syntax:

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

Example:

```
class Student {
    Student() {
        System.out.println("Constructor called");
    }
    public static void main(String[] args) {
        Student S1 = new Student(); // constructor called
    }
}
```

⇒ Parameterized Constructor

A parameterized constructor accepts arguments to initialize Object values at the time of creation.

Syntax:

```
class ClassName {
    ClassName(dataType param1, dataType param2) {
        // use parameters to initialize
    }
}
```

Example:

```
class Student {
    String name;
    int age;
    //parameterized constructor
    Student (String n, int a) {
        name=n;
        age=a;
    }
```

```
void display() {
    System.out.println("Name: "+name+", Age: "+age);
}
Public Static void main(String[] args) {
    Student S1 = new Student("Ram", 20);
    S1.display();
}
}
```

⇒ The this keyword

this is a reference variable in java that refers to the current object.

Uses of this keyword:

1. To refer to current class instance variables.
2. To invoke current class methods / constructors.
3. To pass the current object as a parameter.

① Example (Using this to avoid variable name conflict)

```
class Student {
    String name;
    Student (String name) {
        this.name = name;
    }
    void display() {
        System.out.println("Name: "+name);
    }
    public Static void main (String[] args) {
```

```
        Student s1 = new Student("Meghana");
        s1.display();
    }
}
```

⇒Instance variable hiding

When the local variable (eg: method parameter) has the same name as the instance variable, the local one hides the instance variable. This is called variable hiding.

Using this helps resolve this ambiguity.

Example:

```
class Student {
    String name; // instance variable
    Student(String name) { // parameter name hides instance variable
        this.name = name;
    }
    void display() {
        System.out.println("Name: "+name);
    }
    public static void main(String[] args) {
        Student s = new Student("Meghana");
        s.display();
    }
}
```

⇒ Overloading constructors

Having multiple constructors in a class with different parameters.

```
class Student {
    Student() {
        System.out.println("Default constructor");
    }
    Student(String name) {
        System.out.println("parameterized constructor:" +name);
    }
    public static void main(String[] args) {
            Student obj1 = new Student();
            Student obj2 = new Student("Ram");
    }
}
```

(2)

```
class Add {
    int x,y,total;
    Add()
    {
      x=10;
       y=20;
    }
     Add(int a)
     {
        x=a;
         y=a;
     }
     Add( int a, int b)
     {
        x=a;
        y=b;
     }
```

```
    void Sum() {
        total = x+y;
        S.O.P("Sum=" +total);
    }
}
class Main {
    Public Static void main (String args[])
        Add obj1= new Add();
        Add obj2 = new Add(30);
        Add obj3 = new Add(40,50);
        obj1.Sum();
        obj2.sum();
        obj3.Sum();
    }
}
```

⇒ Garbage collection in java

Garbage collection in java is the process by which Java automatically deletes unused or unreached objects from memory to free up space and improve performance.

Why is it needed

* In Java, memory is allocated dynamically using the new keyword.
* when an object is no longer referenced, it becomes eligible for garbage collection.
* this avoids memory leaks and manual memory management (like in C/C++ with free() or delete).

How it works:

* The java virtual machine (JVM) has a garbage collector that runs in the background.
* It finds unreachable objects and deletes them automatically.
* You can suggest the JVM to run GC using:

```
System.gc();
```

* This is only a request, not a command. JVM decides whether to run GC or not.

finalize() Method in java

The finalize() method is a protected method of the object class. It is called by the garbage collector before destroying an object.

Syntax:

```
protected void finalize() throws Throwable {
    // cleanup code
}
```

purpose:

* To perform cleanup actions before the object is removed from memory.
* For example, closing file streams, releasing network connections, etc.

Example:

```
class Test {
```

```
    protected void finalize() {
        System.out.println(" object is destroyed.");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t = null;
        System.gc(); // request garbage collection
    }
}
```

O/P: object is destroyed

(Note: output may be vary depending on JVM behaviour.)

* Important points
* finalize() is called only once per object.
* It is part of the java.lang.Object
* From Java 9 onwards, finalize() is deprecated due to poor performance and unpredictability.
* Recommended to use try-with-resources or finally blocks for cleanup instead.

⇒ Overloading Methods

When two or more methods in the same class have the same name but different parameters.

Example:

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

```
    double add(double a, double b) {
        return a+b;
    }
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(2,3));
        System.out.println(c.add(2.5, 3.2));
    }
}
```

⇒ Recursion

A method calling itself to solve a smaller version of the same problem.

Example:

```
class RecursionExample {
    int factorial(int n) {
        if (n==1)
            return 1;
        else
            return n * factorial(n-1);
    }
    public static void main(String[] args) {
        RecursionExample r = new RecursionExample();
        System.out.println("Factorial: " + r.factorial(5));
    }
}
```

⇒ Methods Based on Parameters and Return Type

There are four different types of methods based on whether they take parameters or return values:

1. Methods without parameters and without Return values:

These methods neither take any input nor return a result they just perform a specific action.

```
void greet() {
    System.out.println("Hello, everyone!");
}
```

2. Methods with parameters but without Return values:

These methods take input parameters but don't return a result.

```
Void printSum(int a, int b) {
    System.out.println("Sum: "+(a+b));
}
```

3. Methods without parameters but with Return values:

These methods don't need any input, but they return a value.

```
int getNumber() {
    return 42;
}
```

4. Methods with parameters and Return values:

These methods take parameters (input) and also return a value.

```
int multiply(int a, int b) {
    return a*b;
}
```

⇒ this keyword in Java

In Java, the this keyword is a reference variable that refers to the current object of the class. It helps differentiate between instance variables and parameters, and is also used to invoke constructors and methods from within the class.

The this keyword is mainly used for:

① this to refer to instance variable

```
class Student {
    String name;
    Student(String name) {
        this.name = name; // refers to the instance variable
    }
}
```

② this to call current class method

```
class Demo {
    void display() {
        System.out.println("Hello from display");
    }
    void call() {
        this.display(); // same as just calling display()
    }
}
```

③ this() to call constructor

```
class Book {
    Book() {
        System.out.println("Default constructor");
    }
    Book(String name) {
        this(); // calls default constructor
        System.out.println("Book name: " + name);
    }
}
```

* this() must be the first statement in the constructor.

④ this to pass current object as argument

```
class Test {
    void show(Test obj) {
```

```
    System.out.println("Object received");
                        "object: "+obj
    }
    void display() {
        show(this); // passing current object to method
    }
}
t.display(); // Test@15db9742
```

* used in callbacks or frameworks like GUI, JDBC.

⑤ this to return current object

```
class Person {
    person getPerson() {
        return this;
    }
}
void display(){ System.out.println("this is a person object"); }
P1.getperson().display();
```

* Useful in method chaining or builder design patterns.

⇒ this cannot be used in static context (eg: static methods)

⇒ Always refers to current instance of the class.

⇒ Using objects as parameters in Java

In Java, we can pass objects to methods just like primitive data types. This helps in:

* Code reuse
* Accessing & modifying the object's data
* Encapsulation

Why use objects as parameters

* To send complete objects to methods for processing.
* To access or change instance variables or call methods.

Example:

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

```
Student (String n, int m) {
        name = n;
        marks = m;
}
void display() {
    System.out.println(name + " Scored " + marks);
}
void update(Student s) {
    s.marks += 10; // modifies passed object's data
}
Public static void main(String[] args) {
        Student S1 = new Student ("Ram", 75);
        S1.display();
        S1.update(S1); // passing object as parameter
        S1.display();
    }
}
```

Returning objects in java

In java, a method can return an object just like it returns other data types (like int, float, String, etc).

This is useful when:

* you want to create and return a new object
* You want to return an object based on some condition
* You want to perform operations and give the result as an object.

Example:

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

```
// Constructor
Student (String n, int m) {
    name = n;
    marks = m;
}
// Method that returns a Student object
Student getTopper (Student s2) {
    if (this.marks > s2.marks)
        return this;  // return current object
    else
        return s2;  //return passed object
}
void display() {
    System.out.println("Name: "+name+", Marks: "+marks);
}
public static void main (String[] args) {
    Student s1 = new Student("Raju", 85);
    Student s2 = new Student ("Ravi", 92);
    Student topper = s1.getTopper(s2); //Method returns object
    topper.display(); // output: Ravi is the topper
}
}
```

O/P: Name: Ravi, Marks: 92

⇒ A closer look at Argument passing in Java

Argument passing:

When you call a method in Java, you can pass values or objects to it. These are called arguments.

Java supports pass-by-value only – but what that means depends on what you're passing (primitive or object).

1. Primitive data Types

When you pass a primitive value (like int, float, char), a copy of the value is passed to the method.

Ex:

```
void change(int x) {
    x = x+10;
}
public static void main(String[] args) {
    int a=5;
    change(a);
    System.out.println(a);
}
```

* the value of "a" doesn't change because a copy of a is passed to the method.

2. Objects (Reference Types)

When you pass an object, the reference (address) to that object is passed by value. So the method can modify the object's contents, but not change its reference.

Ex:

```
class Student {
    int marks = 50;
}
void update(Student s) {
    s.marks = 90;
}
public static void main(String[] args) {
    Student obj = new Student();
    update(obj);
    System.out.println(obj.marks); // output: 90
}
```

* the method modifies the marks inside the object, and it reflects in main() because both point to the same object.

⇒ printing Statements

1. System.out.println()

* It is used to print a message to the console followed by a new line

Syntax:

```
System.out.println("Hello, Java!");
System.out.println("welcome");
```

O/P: Hello, Java!
welcome

2. System.out.print()

prints the message without adding a new line at the end.

Syntax:

```
System.out.print("Hello");
System.out.print("Java!");
```

O/P:

Hello Java!

3. System.out.printf()

used for formatted output (like in C language).

Syntax:

```
int age = 20;
System.out.printf("Age is: %d", age);
```

O/P: Age is: 20

Comments in Java

comments are non-executable statements used to explain the code. they help in code readability and documentation.

1. Single-line comment (//)

* Begins with //
* used for short explanations.

Example:

```
// this is a single-line comment
System.out.println("Hello");
```

2. Multiline comment (/*...*/)

Used for commenting multiple lines

Example:

```
/* this is a multi-line comment
   explaining the following code */
System.out.println("Hello");
```

3. Documentation Comment (/** ... */)

* used for writing API documentation
* passed by JavaDoc tool

Example:

```
/**
 * this class demonstrates printing in Java.
 */
class Demo {
    // code
}
```

Faculty Name: C. Meghana

Subject Name: Object Oriented Programming using Java

Unit Name: Access Controls and Inheritance

Department Name: AI & ML

Unit-2
Access Controls and Inheritance

⇒ Static

The Static keyword in Java is used for memory management. It means the member belongs to the class rather than instances (objects) of the class. It can be used for variables, methods, blocks, and nested classes.

1. Static variable (class variable)

A Static variable is shared among all objects of the class. It is initialized only once at the time of class loading, not everytime an object is created.

Characteristics:

* Declared using Static keyword.
* Belong to the class, not the instance.
* Saves memory as it is not duplicated per object.
* Can be accessed via Object or class name.

Syntax:

```
class ClassName {
    static dataType variableName;
}
```

Example:

```
class Student {
    int rollNo;
    String name;
    static String college = "JNTU"; //Static variable
    Student (int r, String n) {
        rollNO = r;
        name = n;
    }
```

```
    void display() {
        System.out.println(rollNo + " " + name + " " + college);
    }
    public static void main(String[] args) {
        Student s1 = new Student(101, "Meghana");
        Student s2 = new Student(102, "Anu");
        s1.display();
        s2.display();
    }
}
```

2. Static Method

A static method belongs to the class and can be called without creating an object. It can only access static variables and other static methods directly.

Characteristics:

* Cannot access non-static members directly.
* Cannot use this keyword.
* Can be called using

```
ClassName.methodName();
```

Syntax:

```
class MyClass {
    static void myMethod() {
        // logic
    }
}
```

Example:

```
class Utility {
    static int Square(int x) {
```

```
    return x * x;
  }
  public static void main(String[] args) {
      int result = Utility.square(5);
      System.out.println("Square: " + result);
  }
}
```

Important Rules

* Declaration and later initialization is allowed.
* Initialization can be done inside static blocks, methods, or directly.
* You cannot initialize a static variable after declaration inside a non-static block (like a normal instance initializer)
* you can reassign a static variable as many times as needed.

3. Static Block

A static block is used to initialize static variables. It executes only once, when the class is loaded into memory.

Characteristics:

* Runs before the main() method.
* Ideal for static initialization logic.

Syntax:

```
class MyClass {
    static {
        // initialization code
    }
}
```

Example:

```
class Test {
    static int a;
    static {
        a = 100;
        System.out.println("Static block executed.");
    }
```

```
    public static void main(String[] args) {
        System.out.println("value of a: "+a);
    }
}
```

4. Static Nested Class

A static nested class is a class defined inside another class using the static keyword. It behaves like a regular top-level class, but it is logically grouped within the outer class.

Characteristics:

* Can access only static members of the outer class directly.
* Does not need an instance of the outer class to be created.

```
class Outer {
    static class Inner {
        void show() {
            //logic
        }
    }
}
```

Example:

```
class Outer {
    static int data = 50;
    static class Inner {
        void show() {
            System.out.println("Data= "+data);
        }
    }
}
class Test {
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.show();
    }
}
```

Introducing final

The final keyword in Java is used to create constants or to restrict inheritance and modification. It can be applied to

1. variables
2. methods
3. classes

1) Final variables

A final variable is a constant. Once a value is assigned to it, it cannot be changed.

Syntax:

```
final int MAX = 100;
```

Rules:

* Must be initialized when declared or inside a constructor
* Cannot be reassigned a new value.

Example:

```
class Example {
    final int value = 10;
    void show() {
        // value = 20; // Error : can't assign a value to final variable
        System.out.println("value = " + value);
    }
}
```

2) Final Methods

A final method cannot be overridden by subclasses.

Example:

```
class A {
    final void display() {
        System.out.println("Final method in class A");
    }
}
class B extends A {
    // void display() { } //Error: cannot override final method
}
```

3. Final Classes

A final class cannot be inherited. No subclass can be created from it.

Example:

```
final class Vehicle {
    void run() {
        System.out.println("Running");
    }
}
// class Car extends Vehicle {} // Error: cannot inherit from final class
```

Why use final

* For constants (eg: PI = 3.14)
* To prevent accidental changes
* To secure important logic from being modified
* Improve readability and security.

⇒ Introducing Nested and Inner Classes

A nested class is a class defined within another class.

Types of Nested classes:

1. Static nested class
2. Non-Static Nested Class (Inner class)
   a) Member Inner class
   b) Local Inner class
   c) Anonymous Inner class

1) Static Nested Class

* Defined using static keyword inside an outer class.
* Does not need an instance of the outer class to be accessed.
* Can only access static members of the outer class.

Example:

```
class Outer {
    static int data = 50;
```

```
    static class Nested {
        void display() {
            System.out.println("Data: "+data);
        }
    }
}
class Test {
    public static void main(String[] args) {
        Outer.Nested obj = new Outer.Nested();
        obj.display();
    }
}
```

output:

Data : 50

2. Non-Static Nested classes (Inner Class)

a) Member Inner class

* Defined inside a class but outside methods.
* Can access all members (even private) of outer class.

Example:

```
class Outer {
    int x=10;
    class Inner {
        void Show() {
            System.out.println("x="+x);
        }
    }
}
class Test {
    public static void main(String[] args) {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();
        i.Show();
    }
}
```

output:

x=10

b) Local Inner class

* Defined inside a method of the Outer class.
* Has access to final or effectively final local variables.

Example:

```
class Outer {
    void outerMethod() {
        int num=10; // local variable
        // local inner class
        class LocalInner {
            void display() {
                System.out.println("Number = "+num);
            }
        }
        // Create an object of LocalInner and call display
        LocalInner li=new LocalInner();
        li.display();
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
}
```

output:

Number=10

c) Anonymous Inner class

* No name.
* Used when a class needs to be created for one-time use, usually for one-time use, usually for overriding methods or using interfaces.

Example:

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

```
class Test {
    public static void main(String[] args) {
        Animal a = new Animal() {
            void sound() {
                System.out.println("Roar");
            }
        };
        a.sound();
    }
}
```

output:

Roar

Uses of Nested classes

* Improves encapsulation
* Logical grouping of classes.
* Better readability & structure.

Exploring the String class

* In Java, String is a class in the java.lang package used to store and manipulate text data.
* Strings in Java are objects, not primitive data types.
* Java provides special support for Strings through:
    * String literals ("Hello")
    * String operations (concatenation, comparison, etc)
* Strings are immutable, meaning once created, their content cannot be changed.

Creating Strings:

There are two main ways to create a String:

a) Using String Literals

```
String s1 = "Hello";
```

* Stored in String Constant Pool (SCP)
* If another string with the same content exists, it reuses the same object.

b) Using new keyword

String s2 = new String("Hello");

* Always creates a new object in the heap, even if the same string exists in the SCP.

Immutability of Strings:

When you modify a string, a new object is created instead of changing the original.

Example:

```
String s = "Java";
s.concat("Programming");
System.out.println(s); // output: Java
```

* "Java Programming" is created as a new object, but s still refers to "Java".

Commonly used String Methods:

Method	Description	Example	Output
length()	Returns length of string	"Hello".length()	5
charAt(int index)	Returns character at given index	"Java".charAt(2)	v
substring(int beginIndex)	Returns substring from index	"Hello".substring(2)	llo
substring(int begin, int end)	Returns substring between indices	"Hello".substring(1,4)	ell

equals(Object obj)	Compares content	"Java".equals("java")	false
equalsIgnoreCase(String)	Compares ignoring case	"Java".equalsIgnoreCase("java")	true
compareTo(String)	Compares lexicographically	"A".compareTo("B")	-1
toUpperCase()	converts to ~~lowercase~~ uppercase	"java".toUpperCase()	JAVA
toLowerCase()	Converts to lower case	"JAVA".toLowerCase()	java
trim()	"_hello_".trim() removes leading/ trailing spaces	"hello".trim()	hello
replace(char,char)	Replaces characters	"java".replace('a','o')	jova
Split(String regex)	Splits string	"a,b,c".split(",")	[a,b,c]

String Comparison:

== → Compares references (memory location).

.equals() → Compares actual content.

Example:

```
String S1 = "Java";
String S2 = "Java";
String S3 = new String("Java");
System.out.println(S1==S2); // true (Same object in SCP)
System.out.println(S1==S3); // false (different object in heap)
System.out.println(S1.equals(S3)); // true (content is same)
```

String Concatenation:

Using + operator:

```
String s1 = "Hello";
String s2 = "World";
String s3 = s1 + " " + s2;
System.out.println(s3); // Hello World
```

Using concat() method:

```
String s4 = s1.concat(" ").concat(s2);
```

StringBuffer and StringBuilder:

Since Strings are immutable, for frequent modifications, Java Provides:

* StringBuffer → Mutable and thread-safe
* StringBuilder → Mutable and faster, but not thread-safe.

Example program:

```
public class StringExample {
    public static void main(String[] args) {
        String str = "Java Programming";
        System.out.println("Length: " + str.length());
        System.out.println("Uppercase: " + str.toUpperCase());
        System.out.println("SubString(5): " + str.subString(5));
        System.out.println("Replace: " + str.replace("Java", "Python"));
        System.out.println("Contains 'Pro': " + str.contains("Pro"));
    }
}
```

O/P:

```
Length: 16
Uppercase: JAVA PROGRAMMING
Substring(5): Programming
Replace: Python Programming
Contains 'Pro': true
```

⇒ Object class

* In Java, Object is the parent class of all classes.
* It is part of the java.lang package.
* If a class does not explicitly extend another class, it automatically inherits from Object.
* This means every class you create has access to object class methods.

Importance of object class

* provides common methods that can be used or overridden by all Java classes.
* Ensures uniform behaviour across different types of objects.
* Acts as the root of the root of the class hierarchy in java.

Commonly used Methods of Object class

Method	purpose
toString()	Returns a string representation of the object
equals(Object obj)	Compares two objects for equality
hashCode()	Returns an integer hash code for the object.
getClass()	Returns the runtime class of the object.
clone()	Creates and returns a copy of the object (requires Cloneable interface).
finalize()	Called before the object is destroyed by garbage collector
wait(), notify(), notifyAll()	Used for thread synchronization.

Example:

```
class Student {
    String name;
    Student (String name) {
        this.name = name;
    }
    public String toString() {
        return name;
    }
    public boolean equals (Object obj) {
        return (obj instanceof Student) && this.name.equals(((Student)obj)
                .name);
    }
}
public class ObjectClassDemo {
    public static void main(String[] args) {
        Student S1 = new Student ("Meghana");
        Student S2 = new Student ("Meghana");
        System.out.println(S1); // Meghana
        System.out.println(S1.equals(S2)); // true
        System.out.println(S1.getClass()); // class Student
    }
}
```

O/P:

```
Meghana
true
class Student
```

Key points

* Object class methods can be overridden to customize behavior.
* All Java objects can be stored in variables of type Object.
* Essential for polymorphism, collections, and object comparison.

⇒ Method overriding in Java

* Method overriding occurs when a subclass provides its own implementation of a method that is already defined in its superclass.
* The overriding method must have the same name, return type, and parameters as the method in the superclass.

key points

1) Same Method Signature – name, parameters, and return type must be the same.
2) Inheritance required – overriding happens only through IS-A relationship (extends).
3) Access Modifier Rule – overriding method cannot have a lower access level than the overridden method.
4) Return Type Rule – must be the same or covariant type (subclass of original return type).
5) Cannot Override final/ static/ private Methods
   * final → cannot be overridden.
   * static → method hiding, not overriding
   * private → not inherited, so not overridden.
6) @Override Annotation (optional but recommended) helps detect mistakes.
7) Runtime polymorphism – which method executes depends on the object's actual type, not the reference type.

Example:

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

```
class Dog extends Animal {
    @Override
    void Sound(){
        System.out.println(" Dog barks ");
    }
}
Public class TestOverride {
    public static void main(String[] args) {
        Animal a = new Dog(); //upcasting
        a.sound(); // Calls Dog's version
    }
}
```

O/P: Dog barks

⇒ Dynamic Method Dispatch

* Dynamic Method Dispatch (also called Runtime polymorphism) is the process by which a call to an overridden method is resolved at runtime, not at compile time.

* It occurs when a Superclass reference variable points to a Subclass reference variable points to a Subclass object, and the overridden method is called.

Key points

1) uses Method overriding – Required for runtime polymorphism.

2) Reference Type vs. object Type
   * Reference Type decides which methods are accessible at compile time
   * Object Type decides which method actually executes at runtime.

3) upcasting is common:
   Superclass ref = new Subclass();

4) Helps in extensibility & loose coupling of code.

How It works

* At compile time → Java checks method availability based on the reference type.
* At runtime → JVM determines which version (superclass or subclass) to execute based on the object's type.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}
public class DynamicDispatchExample {
    public static void main(String[] args) {
        Animal ref; // Reference of Superclass
        ref = new Dog(); // Dog object
        ref.sound(); //calls Dog's version
        ref=new Cat(); //Cat object
        ref.sound(); // calls Cat's version
    }
}
```

o/p: Dog barks
Cat meows

Advantages

* Implements runtime polymorphism.
* Supports code reusability and flexibility.
* Makes programs easily extensible.

Important Rules

* works only with overridden instance methods (non-static).
* Static, private, and final methods are not involved in dynamic dispatch (they are resolved at compile time).
* Variables are not polymorphic - reference type decides which variable is accessed.

⇒ Abstract classes

* An abstract class is a class declared with the abstract keyword.
* It can have abstract methods (without body) and non-abstract methods (with body).
* Cannot be instantiated directly, must be inherited by a subclass.

key points

* Abstract Method: declared without implementation

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

* A subclass must implement all abstract methods unless the subclass itself is abstract.
* Can have constructors, fields, and method implementations.
* Supports partial abstraction (can mix abstract and concrete methods).

Example:

```
abstract class Shape{
    abstract void draw(); // abstract method
    void info(){ // concrete method
        System.out.println("This is a Shape.");
    }
}
class Circle extends Shape{
    void draw(){
        System.out.println("Drawing a circle");
    }
}
Public class AbstractExample{
    public static void main(String[] args){
        Shape s= new Circle(); // upcasting
        s.draw();
        s.info();
    }
}
```

O/P: Drawing a circle
This is a Shape.

⇒ Inheritance Basics

Inheritance is the process by which one class (child/subclass) acquires the properties and behaviours (fields and methods) of another class (parent/superclass).

Purpose:
Reusability of code, method overriding, and establishing relationships between classes.

Superclass → the class whose members are inherited.
Subclass → the class that inherits members from super class.
extends → keyword to implement inheritance.

Syntax:

```
class Parent {
    //fields & methods
}
class Child extends Parent {
    // additional field & methods
}
```

⇒ Member Access & inheritance

* Inherited Members:
  ⇒ All public and protected members are inherited by the subclass.
  ⇒ Default (package-private) members are inherited only if subclass is in the same package.
  ⇒ private members are not inherited directly (can be accessed through public / protected methods.

* Overriding & Hiding:
  ⇒ Methods can be overridden in the subclass (same name, same parameters).
  ⇒ Fields can be hidden, but it's not recommended.

Example:

```
class A {
    public int x = 10;
    private int y = 20; // Not inherited
    public void display() {
        System.out.println("x = " + x);
    }
}
class B extends A {
    public void show() {
        System.out.println("From Subclass, x = " + x); // Accessible
        // System.out.println(y); // ERROR: y is private
    }
}
```

⇒ A practical Example

```
class Vehicle {
    int speed = 50;
    void display() {
        System.out.println("Vehicle Speed: " + speed);
    }
}
class Car extends Vehicle
    int speed = 100; // Hides parent speed
    void displaySpeed() {
        System.out.println("Car speed: " + speed);
    }
}
Public class TestInheritance {
    public static void main(String[] args) {
        Car c = new Car();
        c.display(); // From Vehicle
        c.displaySpeed(); // From car
    }
}
```

o/p: Vehicle Speed: 50
Car Speed: 100

⇒ Accessing Superclass Members

Use the super keyword to access:

1. Superclass variables when they are hidden in subclass.
2. Superclass methods when they are overridden in Subclass.
3. Super class Constructors from Subclass constructors.

⇒ Usage of super keyword

1) Access Superclass variable

```
class Parent {
    int num = 100;
}
```

```
class Child extends Parent {
    int num = 200;
    void printNumbers() {
        System.out.println("Child num: " + num);
        System.out.println("Parent num: " + super.num);
    }
}
```

2. Call Superclass Method

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    void sound() {
        super.sound(); // call parent method
        System.out.println("Dog barks");
    }
}
```

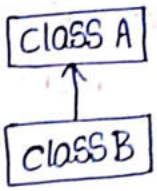
3. Call Superclass Constructor

```
class Person {
    Person(String name) {
        System.out.println("Name: "+name);
    }
}
class Student extends Person {
    Student(String name) {
        super(name); // call person's constructor
        System.out.println("Student object created");
    }
}
```

⇒ Types of Inheritance in Java

Java supports the following types of inheritance:

1. Single Inheritance
   * A class inherits from only one Superclass.
   * Example: Class B extends A



Example:

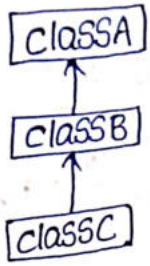
```
class A {
    void display() {
        System.out.println("Class A method");
    }
}
class B extends A {
    void show() {
        System.out.println("Class B method");
    }
}
public class SingleInheritanceDemo {
    public static void main(String[] args) {
        B obj = new B();
        obj.display(); // from A
        obj.show(); // from B
    }
}
```

O/P: Class A method
Class B method

2. Multilevel Inheritance
   * A class is derived from another derived class (chain of inheritance)
   * Example: class C extends B and class B extends A

Example:

```
class A {
    void methodA() {
```

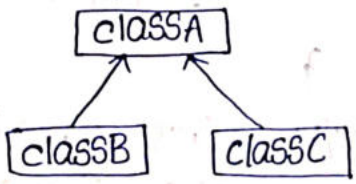


```
    System.out.println("From Class A");
  }
}
class B extends A {
    void methodB() {
        System.out.println("From Class B");
    }
}
class C extends B {
    void methodC() {
        System.out.println("From class C");
    }
}
Public class MultilevelInheritanceDemo {
    public static void main(String[] args) {
        C obj = new C();
        obj.methodA();
        obj.methodB();
        obj.methodC();
    }
}
```

O/P: From Class A
From Class B
From Class C

3. Hierarchical Inheritance

* Multiple classes inherit from the same Superclass.
* Example: class B extends A, class C extends A



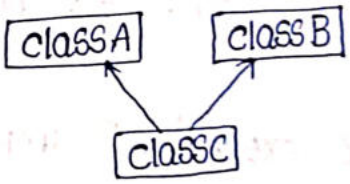
Example:

```
class A {
  void greet() {
      System.out.println("Hello from Class A");
  }
}
class B extends A {
    void displayB() {
        System.out.println("Hello from class B");
    }
}
```

```
class C extends A {
    void displayC() {
        System.out.println("Hello from Class C");
    }
}
public class HierarchicalInheritanceDemo {
    public static void main(String[] args) {
        B objB = new B();
        objB.greet();
        objB.displayB();
        C objC = new C();
        objC.greet();
        objC.displayC();
    }
}
```

O/P:
```
Hello from Class A
Hello from Class B
Hello from Class A
Hello from Class C
```

4. Multiple Inheritance (Using Interfaces)



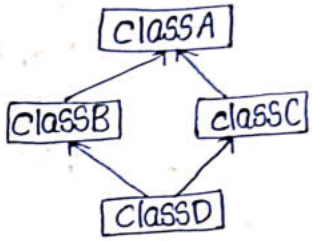
* A class implements multiple interfaces.
* Example: class D implements X, Y

Example:
```
interface X {
    void methodX();
}
interface Y {
    void methodY();
}
class Z implements X, Y {
    public void methodX() {
        System.out.println("From Interface X");
    }
    public void methodY() {
        System.out.println("From Interface Y");
    }
}
```

```
public class MultipleInheritanceDemo {
    public static void main(String[] args) {
        Z obj = new ZC();
        obj.methodXC();
        obj.methodYC();
    }
}
```

O/P: From Interface X
From Interface Y

5. Hybrid Inheritance (class + Interface)



* Combination of two or more types of inheritance.
* In Java, it is achieved using classes + interfaces.

Example:

```
interface A {
    void methodAC();
}
class B {
    void methodBC() {
        System.out.println("From class B");
    }
}
class C extends B implements A {
    public void methodAC() {
        System.out.println("From Interface A");
    }
}
public class HybridInheritanceDemo {
    public static void main(String[] args) {
        C obj = new CC();
        obj.methodAC();
        obj.methodBC();
    }
}
```

O/P: From Interface A
From Class B

⇒ Accessing Constructors in Inheritance

* When you create an Object of a subclass, constructors of both superclass and subclass are executed.
* Java always calls the superclass constructor first, then the subclass constructor.
* If the superclass does not have a default (no-arg) constructor, you must explicitly call a superclass constructor using super(...) in the subclass.
* super() must be the first statement inside the subclass constructor.

How it works

* Implicit Call: If you don't write super(), java adds it automatically to call the no-argument constructor of the superclass.
* Explicit Call: You can use super(arguments) to call a specific constructor from the superclass.

Example – Implicit super()

```
class Parent {
    parent() {
        System.out.println("parent constructor called");
    }
}
class Child extends Parent {
    Child() {
        System.out.println("Child constructor called");
    }
}
public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
    }
}
```

O/P: parent constructor called
Child constructor called

Example - Explicit Super()

```
class Parent {
    Parent(String msg) {
        System.out.println("Parent says: "+msg);
    }
}
class Child extends Parent {
    Child(String msg) {
        super(msg); // calling parameterized constructor of parent
        System.out.println("Child constructor called");
    }
}
Public class Main {
    public static void main(String[] args) {
        Child obj = new Child("Hello from Parent");
    }
}
```

O/P:

```
Parent says: Hello from Parent
Child constructor called
```

Important Rules

1) super() must be the first statement in the Subclass constructor.
2) If you don't call super() explicitly, Java inserts a call to the no-arg constructor of the Superclass automatically.
3) If the Superclass has only parameterized constructors, you must explicitly call one of them from the Subclass.

⇒ Using final with Inheritance

The final keyword in java can be used to restrict the usage of classes, methods, and variables.

When it comes to inheritance, final plays an important role in preventing overriding or extending.

① final class

* A final class cannot be inherited.
* This is used when you want to prevent other classes from extending it.

Example:

```
final class Parent {
    void display() {
        System.out.println("This is a final class");
    }
}
// This will cause a compile-time error
// class Child extends Parent { }
public class Main {
    public static void main(String[] args) {
        parent P = new parent();
        P.display();
    }
}
```

O/P: This is a final class

② final Method

* A final method cannot be overridden in a subclass
* This is useful when you want a method's implementation to remain unchanged for all subclasses.

Example:

```
class Parent {
    final void show() {
        System.out.println("Final method in Parent");
    }
}
class Child extends parent {
    // This would cause an error if we try to override:
    // void show() { }
    void display() {
        System.out.println("Child class method");
    }
}
```

```
Public class Main {
    Public Static void main(String[] args) {
        Child c = new Child();
        c.show();   // from Parent
        c.display(); // from Child
    }
}
```

O/P: Final method in Parent
Child class method

3. final variable

* Once assigned, the value of a final variable cannot be changed.
* It must be initialized at the time of declaration or inside the constructor.

Example:

```
class Parent {
    final int value=100;
    void display() {
        System.out.println("value = " +value);
    }
}
public class Main {
    public Static void main(String[] args) {
        Parent p = new Parent();
        p.display();
    }
}
```

O/P: value = 100

⇒ Access Control

Access control in Java determines while which classes, methods, and variables can be accessed from where in your program. It is implemented using access modifiers and non-access modifiers.

Importance of Access Control

* protects data from unauthorized access.
* Helps maintain encapsulation.
* Avoids accidental modifications.
* Controls visibility of variables/methods across packages.

Types of Access Modifiers

Java provides four levels of access control:

1. private

* Accessible only within the same class.
* Not visible outside the class (even in subclasses).
* Commonly used for data hiding.

Example:

```
class Test {
    private int data=10;
    private void display() {
        System.out.println("private method");
    }
}
```

2. default (package-private)

* If no modifier is specified, it's default.
* Accessible within the same package only.
* Not visible in other packages.

Example:

```
class Test{
    int data=10; //default
    void display() { // default
        System.out.println("Default method");
    }
}
```

3. protected

* Accessible within the same package.
* Accessible in subclasses (even in different packages) via inheritance.

* Not accessible from non-subclass classes in other packages.

Example:

```
class Test {
    protected int data = 10;
    protected void display() {
        System.out.println("protected method");
    }
}
```

4. public

* Accessible from anywhere in the program.
* No restrictions.

Example:

```
class Test {
    public int data = 10;
    public void display() {
        System.out.println("public method");
    }
}
```

Rules of Access Control:

* Access control applies to classes, constructors, methods, and variables.
* A class can only have public or default access (not private or protected).
* Members of a class can have any access level.

⇒ Creating a Multilevel Hierarchy

In multilevel inheritance, a class is derived from another derived class, forming a chain of inheritance.

Example: class C → extends Class B → extends Class A.

purpose:

* To build classes step-by-step.
* To reuse and extend features multiple times.

Syntax:

```
class A {
   //base class
}
class B extends A {
    //derived from A
}
class C extends B {
    //derived from B
}
```

Example:

```
class Animal {
    void eat() {
        System.out.println("eating");
    }
}
class Mammal extends Animal {
    void walk() {
        System.out.println("walking...");
    }
}
class Dog extends Mammal {
    void bark() {
        System.out.println("Barking...");
    }
}
public class MultilevelExample {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat(); // from Animal
        d.walk(); // from Mammal
        d.bark(); // from Dog
    }
}
```

O/P: Eating
walking
Barking