JAVA PROGRAMMING

Java and its usage

JAVA was developed by Sun Microsystems Inc in 1991, later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton.

The phases of program execution

- **Java Virtual Machine (JVM)** - It executes the bytecode generated by compiler.

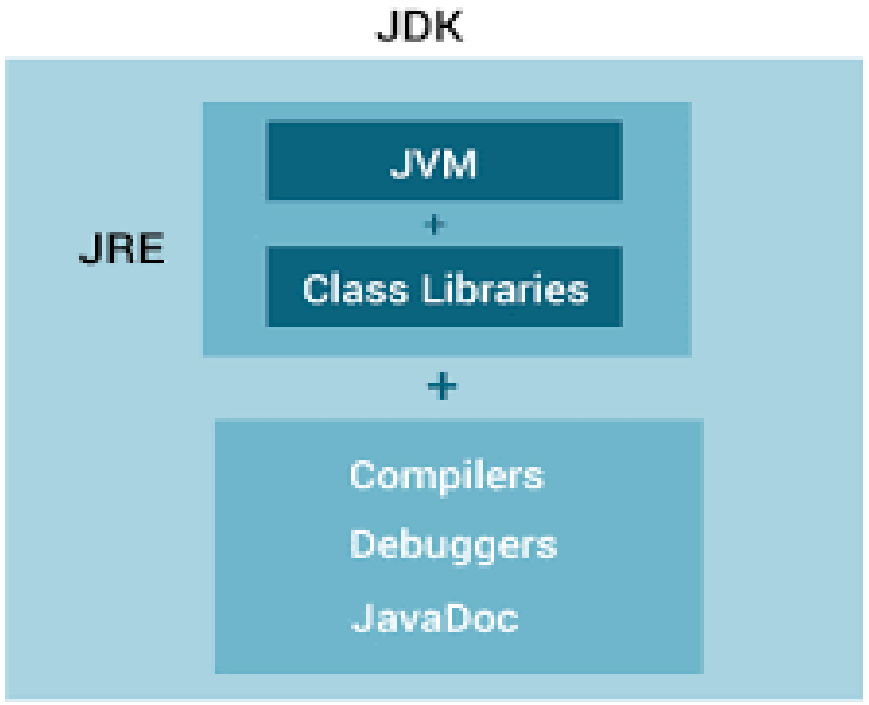
- **Java Development Kit(JDK)** – It is a complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc.

- **Java Runtime Environment(JRE)** - JRE installed on the system, one can run a java program - however one won't be able to compile it.

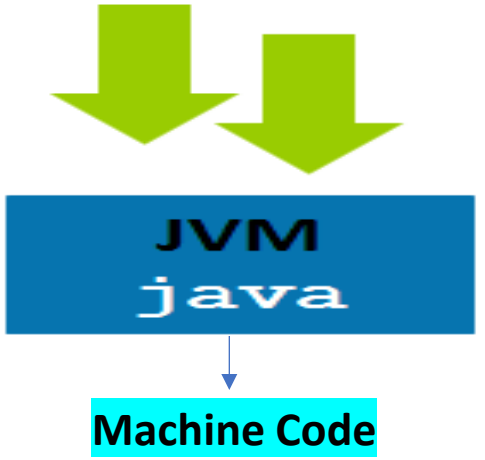
- **Java Compiler (javac)** - javac is the executable/application which compiles the .java source files into the byte code (.class files)

Java Environment

Java utilizes both a compiler and an interpreter in its execution process, making it a **"compiler-interpreter language."**



The Java Virtual Machine (JVM), which includes a software-based interpreter, then interprets this bytecode. The **JVM translates the bytecode into machine code specific to the underlying hardware and operating system at runtime**, allowing the Java application to run. Additionally, **the JVM often incorporates a Just-In-Time (JIT) compiler, which dynamically compiles frequently executed parts of the bytecode into native machine code for improved performance.**



- **Just In Time Compiler (JIT) –** It helps in compiling certain parts of byte code into the machine code for higher performance.

- Java is distributed in two packages - JDK and JRE.

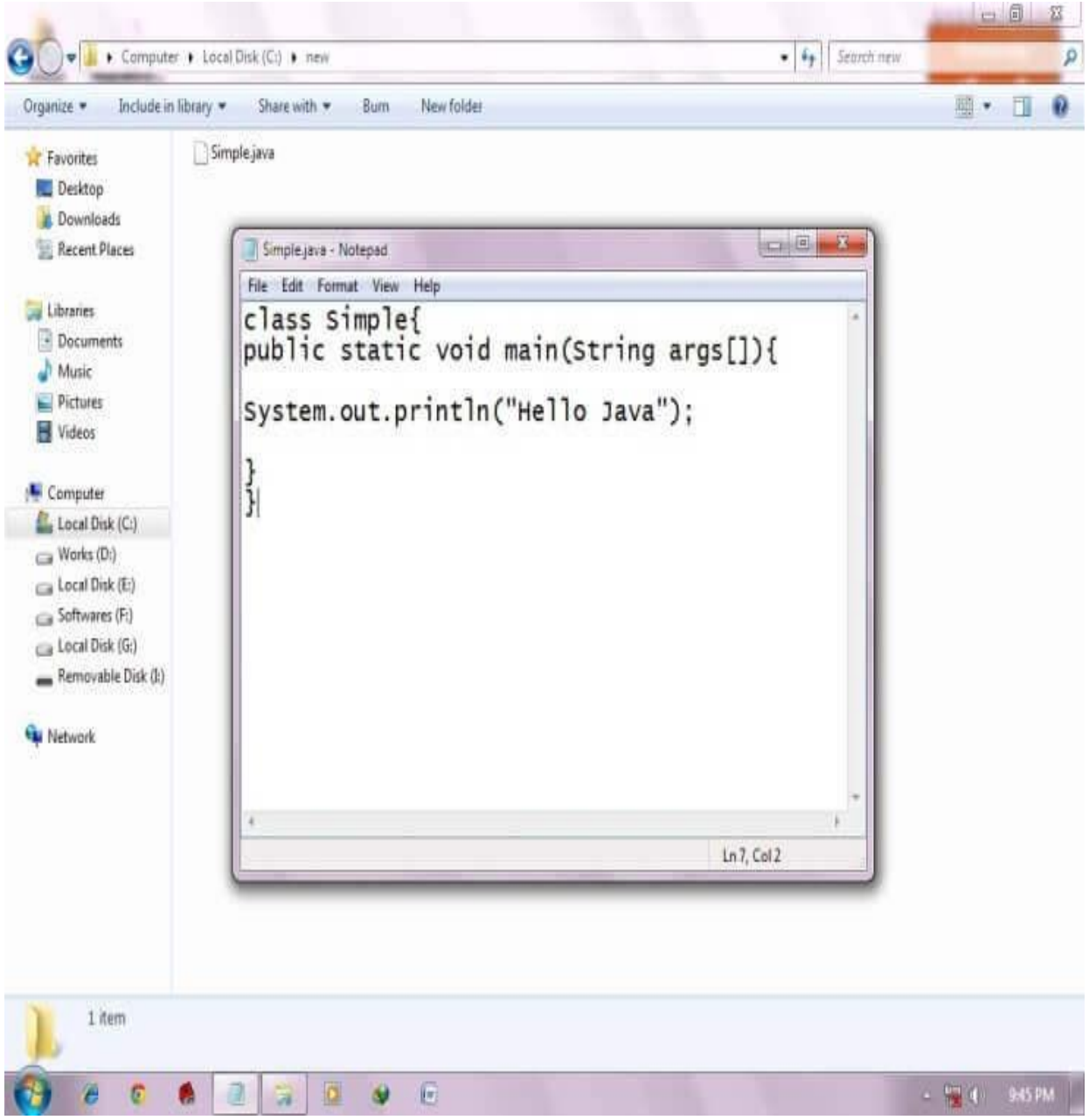
- When JDK is installed it also contains the JRE, JVM and JIT apart from the compiler, debugging tools.

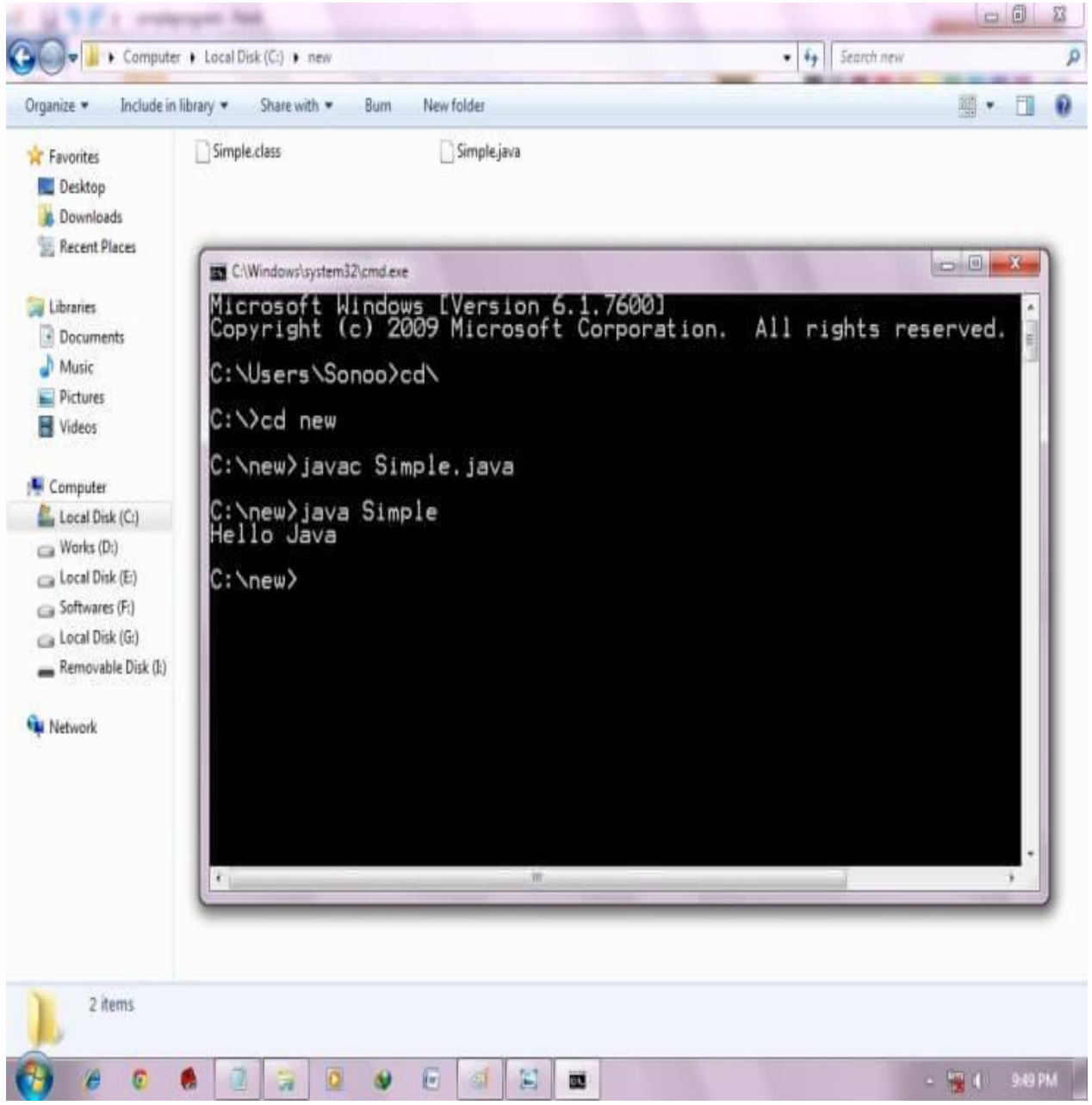
- When JRE is installed it contains the JVM and JIT and the class libraries.

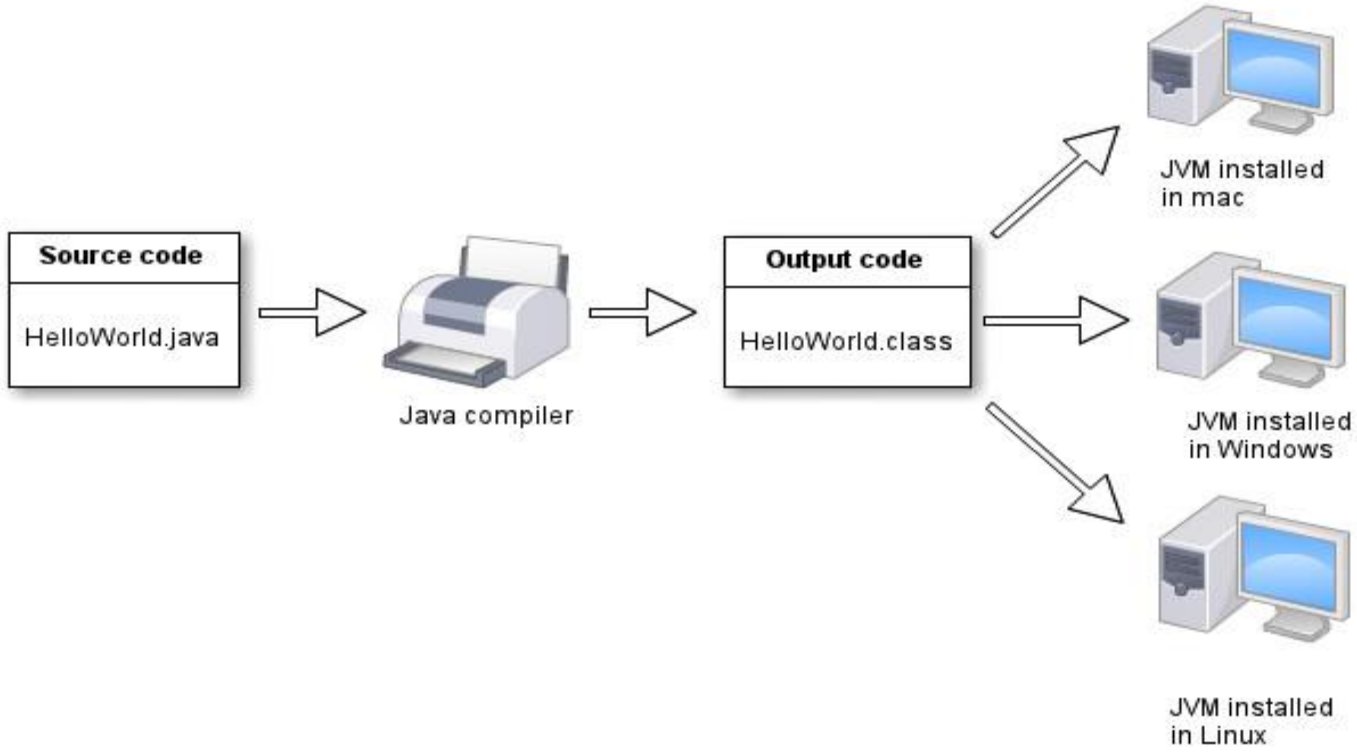
Sample java program and its explanation - Compilation Process

- Three things need to be done to make a java program work. They are,
    1. Create
    2. Compile
    3. Run

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


Computer ▸ Local Disk (C:) ▸ new
Search new
Organize
Include in library
Share with
Burn
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (C:)
Works (D:)
Local Disk (E:)
Softwares (F:)
Local Disk (G:)
Removable Disk (I:)
Network
Simple.java
Simple.java - Notepad
File Edit Format View Help
class Simple{
public static void main(String args[]){
System.out.println("Hello Java");
}
}
Ln 7, Col 2
1 item
9:45 PM


Computer ▸ Local Disk (C:) ▸ new
Search new
Organize
Include in library
Share with
Burn
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (C:)
Works (D:)
Local Disk (E:)
Softwares (F:)
Local Disk (G:)
Removable Disk (I:)
Network
Simple.class
Simple.java
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
C:\new>java Simple
Hello Java
C:\new>
2 items
9:49 PM



1. Package
2. Import Statement - **import java.io.*;**
3. Class - **class Simple**
4. Access Specifiers (also known as Visibility Specifiers) – public, private, protected, default
5. main method - **public static void main(String[] args)**
6. print() method - **System.out.print("Hello World");**
7. The braces **{…}**

OOPs – Concepts

1. Encapsulation

Definition: Encapsulation is the mechanism of wrapping data (variables) and code (methods) together as a single unit, typically **inside a class, and restricting direct access to some of the object's components.**

- Example 1:

```
public class Person {
    private String name; // encapsulated field

    public String getName() {
        return name;
    }
public void setName(String name) {
        this.name = name;
    }
}
```

Example 2

Accessing Private Members Directly

```
// Encapsulated class
public class Person {
    private String name; // private member

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Trying to Access Private Member from Another Class

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
            p.name = “ABC";   // ❌ Compile-time error: name has private access in Person
           p.setName(“ABC"); // ✅ Allowed: uses public method
         System.out.println(p.getName()); // ✅ Allowed: uses public method – ouput - ABC
    }
}
```

Purpose of this Operator in Java

In Java, the this keyword is a reference variable that refers to the current object (the object on which the method or constructor is being invoked).

Main Uses of this

Use	Description	Example
1. Refer to current class instance variable	Useful when local and instance variables have the same name.	this.name = name;
2. Call current class method	Used to call another method of the same class.	this.display();
3. Invoke current class constructor	To call one constructor from another constructor in the same class.	this();
4.Return current object	Return the current object from a method.	return this;

2. Inheritance

- Definition: Inheritance allows a class (child/subclass) to inherit properties and methods from another class (parent/superclass), promoting code reuse.
- **Through Interface – Java Supports Multiple Inheritance**

Example:

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

Diamond Problem & Resolution

- If two interfaces provide the same default method, and a class implements both, one must override the method to resolve the ambiguity.

```
interface A {
    default void greet() {
        System.out.println("Hello from A");
    }
}

interface B {
    default void greet() {
        System.out.println("Hello from B");
    }
}

class C implements A, B {
    // Must override to resolve ambiguity
    public void greet() {
        A.super.greet();  // or B.super.greet()
    } }
```

```
public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.greet();
    }
}
```

- **NOTE :** Use of super in Diamond Problem (with Interfaces) in Java
- **When a class implements multiple interfaces that contain default methods with the same name, Java throws a compile-time error due to ambiguity.**
- In such cases, you must override the method and use InterfaceName.super.methodName() to specify which interface's default method to call.
- This use of InterfaceName.super.method() is a special case of the super keyword to resolve the Diamond Problem in interfaces.
- **The super keyword** in Java is used to **refer to the immediate parent class of a subclass**. It is most commonly used in inheritance to access members (constructors, methods, variables) of the superclass that are overridden or hidden in the subclass.

Main Uses of super Keyword

Use Case	Description	Example
1. **Access parent class method**	Call a method from the parent class that is overridden in the child class	super.methodName();
2. **Access parent class variable**	Access a field (variable) that is shadowed in the subclass	super.variableName;
3. **Call parent class constructor**	Call a constructor of the superclass from the subclass constructor	super(); must be **first statement**

3. Polymorphism:

- Definition: Polymorphism means "many forms". It allows one interface to be used for a general class of actions. The most common use is when a parent class reference is used to refer to a child class object.

- Two types:
- **Compile-time Polymorphism (Method Overloading)** - Defining **multiple methods with the same name** in the same class but with **different parameters**.
- **Runtime Polymorphism (Method Overriding)** - Redefining a method in a **subclass** that already exists in the **parent class** - **same** (name, parameters, return type)

Example – Overloading (compile-time):

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

Example – Overriding (runtime):

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

4. Abstraction:

Definition: **Abstraction is the process of hiding internal implementation details and showing only essential features to the user.**

Two ways to achieve abstraction in Java:

- **Abstract Classes**
- **Interfaces**

Example – Abstract Class:

```
abstract class Shape {
    abstract void draw(); // abstract method
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Example – Interface: - through Interface – Java Supports Multiple Inheritance

```
interface Vehicle {
    void start(); // implicitly public and abstract
}
class Car implements Vehicle {
    public void start() {
        System.out.println("Car starts");
    }
}
```

Java Source File Structure

```
// 1. Package declaration (optional, but must be the first line if present)
package mypackage;

// 2. Import statements (optional, appear after package and before class definition)
import java.util.Scanner; //provides data structures, collection framework, date/time, and other helper classes, Random,Scanner (for user input),StringTokenizer
import java.io.DataInputStream; //package provides classes for reading and writing data (files, streams, etc.).

// 3. Class/interface/enum/record definitions –
public class MyClass {
    // 4. Fields (variables)
    private int number;
    private String name;
```

Enum

- In **Java**, an **enum** (short for *enumeration*) is **a special data type used to define a collection of constant values.**
- In Java, enum is not part of any package like java.util or java.io.
- **It is a language feature introduced in Java 5 and is built into the Java language itself**.
- Technically, **every enum type one creates implicitly extends the abstract class java.lang.Enum** - And since everything in java.lang is **automatically imported**, one don't need to import anything to use enum.

Example:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumExample {
    public static void main(String[] args) {
        Day today = Day.MONDAY;

        if (today == Day.MONDAY) {
            System.out.println("Start of the week!");
        }
    }
}
```

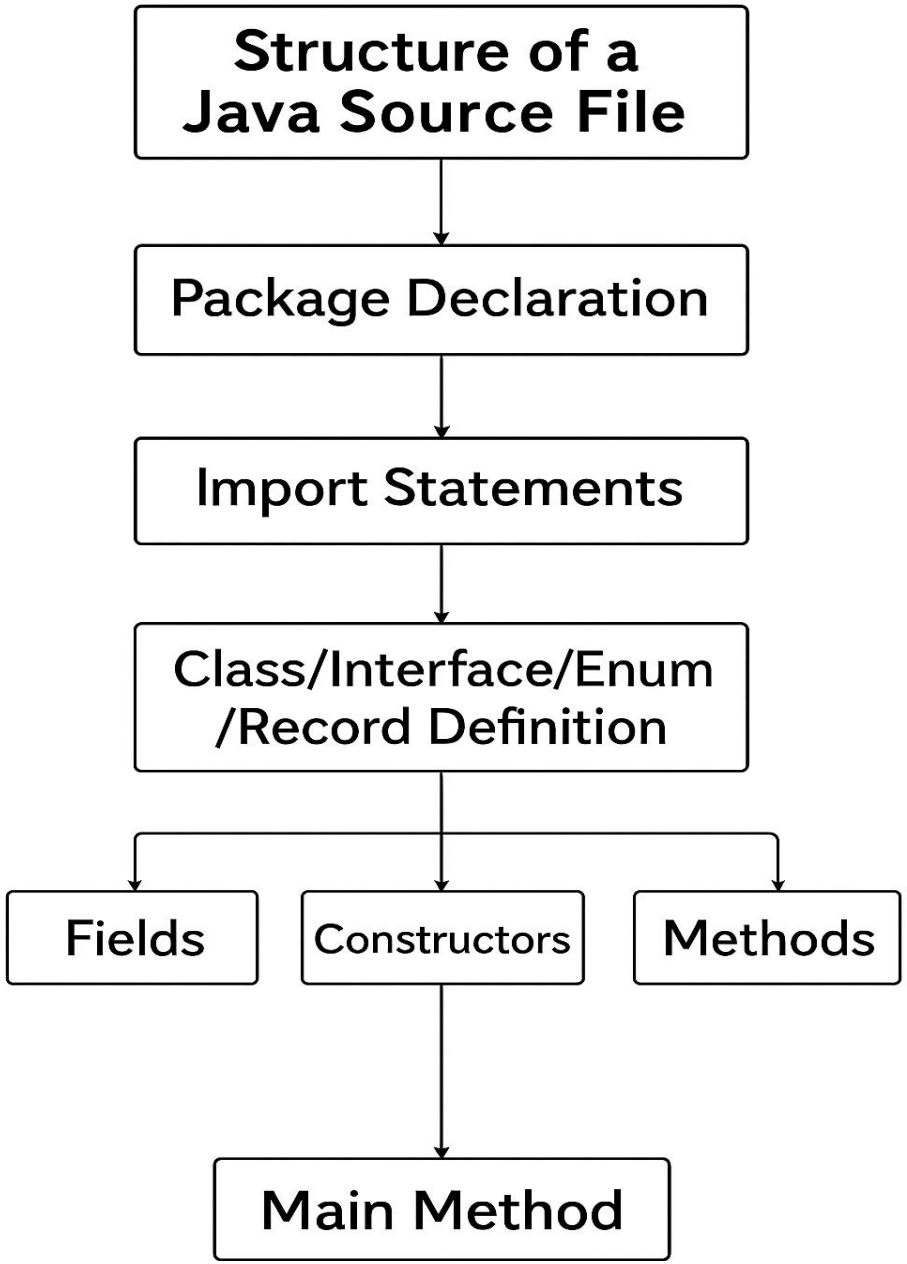
The java.lang package contains:

- Core classes (Object, Class, Enum)
- Wrapper classes (Integer, Double, Boolean, etc.)
- String handling (String, StringBuilder, StringBuffer)
- Math & System utilities (Math, System, Runtime)
- Multithreading support (Thread, Runnable)
- Exception hierarchy (Throwable, Exception, Error)

```
// 5. Constructors
    public MyClass(int number, String name) {
        this.number = number;
        this.name = name;
    }

    // 6. Methods
    public void displayInfo() {
        System.out.println("Number: " + number + ", Name: " + name);
    }

    // 7. Main method (entry point, optional)
    public static void main(String[] args) {
        MyClass obj = new MyClass(1, "Priya");
        obj.displayInfo();
    }
}
```



Keywords, Identifiers and comments

1. Java Keywords

- **Definition:** Reserved words that have a predefined meaning in Java and cannot be used as identifiers (variable names, class names, etc.).
- **Examples:**

  `class`, `public`, `static`, `void`, `if`, `else`, `for`, `while`, `return`, `int`, `double`, `new`, `try`, `catch`, `this`, `super`, `break`, `continue`, `true`, `false`, `null` (and many others — Java has ~50 keywords).
- **Example Code:**

java Copy Edit

```java
public class Example {   // 'public', 'class' are keywords
    public static void main(String[] args) {
        int number = 10;  // 'int' is a keyword
    }
}
```

2. Identifiers

- **Definition:** Names given to variables, methods, classes, or other user-defined items.
- **Rules:**
  - Can contain letters, digits, `_`, and `$`
  - Must **not** start with a digit
  - Cannot be a Java keyword
  - Java is **case-sensitive** (`Name` ≠ `name`)
- **Examples:** `myVariable`, `sum1`, `EmployeeDetails`, `_tempValue`
- **Example Code:**

java Copy Edit

```java
int age = 25;          // 'age' is an identifier
String userName = "Priya"; // 'userName' is an identifier
```

3. Comments

- **Definition:** Notes in the code ignored by the compiler, used for explanation or documentation.
- **Types:**
  1. **Single-line comment**

```java
// This is a single-line comment
int x = 5; // assign value 5 to x
```

  2. **Multi-line comment**

```java
/* This is a
   multi-line comment */
int y = 10;
```

  3. **Documentation comment** (for generating API docs using Javadoc)

```java
/**
 * This class demonstrates comments in Java.
 * @author Priya
 */
public class Demo { }
```

VARIABLES

Local Variables

- **Declared inside a method, constructor, or block**
- **Only accessible within that method or block**
- **Must be initialized before use**

```
public class Example {
    void show() {
        int a = 10; // local variable
        System.out.println(a);
    }
}
```

Instance Variables (Non-static Fields)

- **Declared inside a class but outside any method**
- **Each object gets its own copy**
- **Accessed using object references**
- **Default values are assigned if not initialized**

```
public class Example {
    int number = 5; // instance variable
    void display() {
        System.out.println(number);
    }
}
```

Static Variables (Class Variables)

- **Declared using static keyword**
- **Shared among all objects of the class**
- **Memory allocated only once at class loading time**

```
public class Example {
    static int count = 0; // static variable

    Example() {
        count++;
    }

    void display() {
        System.out.println("Count: " + count);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Example e1 = new Example();
        e1.display();           // Line A

        Example e2 = new Example();
        Example e3 = new Example();
        e2.display();           // Line B

        e3.display();           // Line C
    }
}
```

Output: - **CORRECT -?**

```
Count: 1
Count: 3
Count: 3
```

Constants using final

In Java, when one wants to declare a constant, one usually uses the final keyword.

Final means that once a variable is assigned a value, it cannot be changed (it becomes a constant).

Example 1: Declaring a constant

```
public class ConstantsExample
{
public static final double PI = 3.14159;  // constant value
public static final int MAX_USERS = 100;  // constant value
public static void main(String[] args)
{
System.out.println("PI = " + PI);
System.out.println("Max Users = " + MAX_USERS);
 // PI = 3.15;  ❌ Error: cannot assign a value to final variable
 }
}
```

NOTE:

final keyword → prevents reassignment of the variable.

static keyword → ensures the constant belongs to the class (not an object).

By convention, constants are written in UPPERCASE letters with words separated by underscores.

Access modifiers /Access specifiers

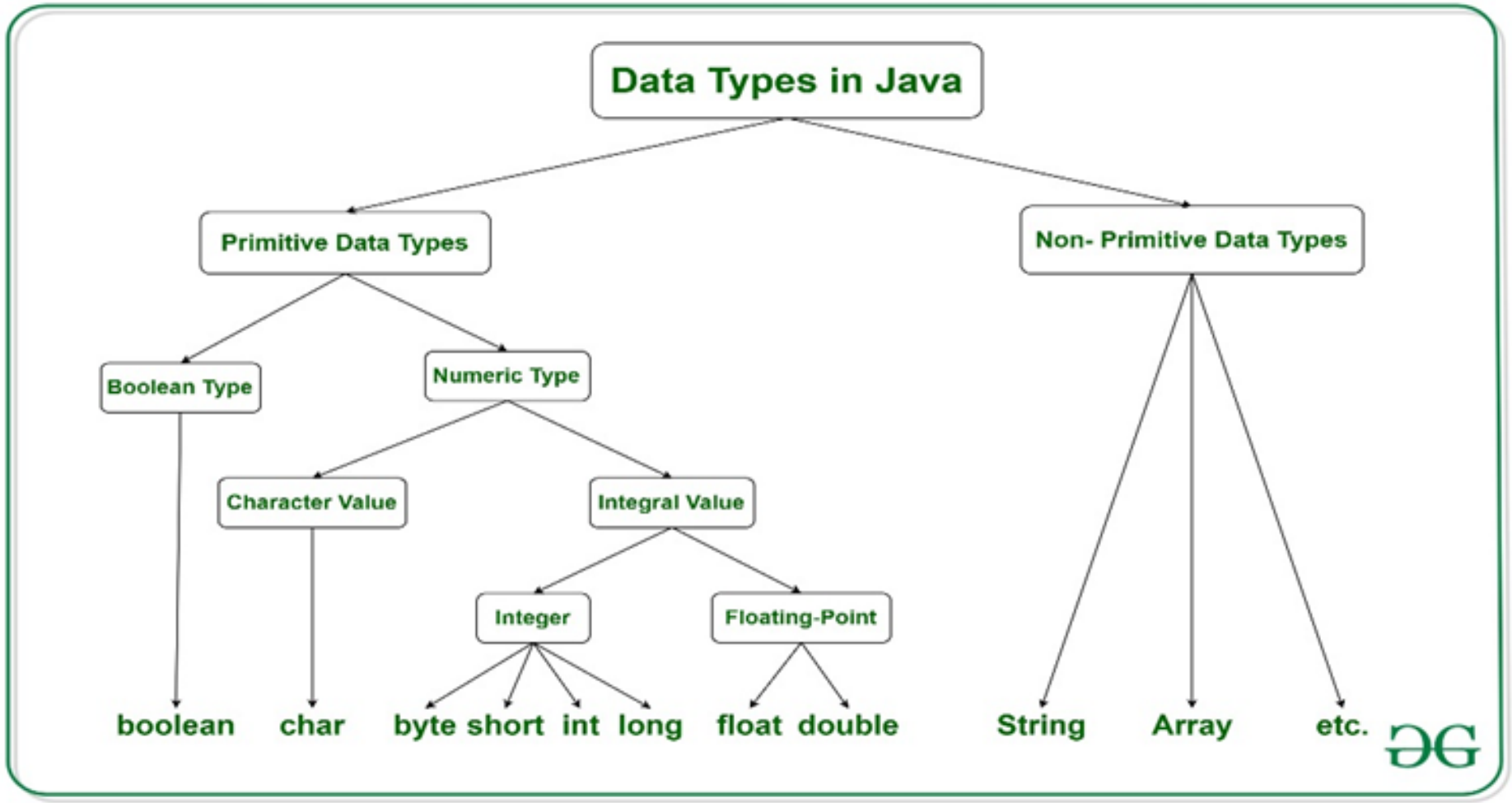
In **Java**, **access modifiers** (also called **access specifiers**) are keywords used to define the **visibility** and **accessibility** of classes, methods, variables, and constructors.

When **no keyword (public, private, protected) is used, Java assigns default access by default.** This allows the member (class, method, variable, or constructor) to be accessible only within the same package.

Modifier	Same Class	Same Package	Subclass (different package)	World (everywhere)
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
no modifier (default)	✅	✅	❌	❌
private	✅	❌	❌	❌

Data Types in Java

- In Java, **data types** specify the type of data a variable can hold. They are broadly categorized into two groups: **Primitive Data Types** and **Non-Primitive (Reference) Data Types**.



Primitive Data Types

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte (16 bits)
byte	0	1 byte (8 bits)
short	0	2 byte (16 bits)
int	0	4 byte (32 bits)
long	0L	8 byte (64 bits)
float	0.0f	4 byte (32 bits)
double	0.0d	8 byte (64 bits)

Non-Primitive (Reference) Data Types

These are derived types and are more flexible. They don’t store the actual data but a reference (memory address).

- **String** → A sequence of characters (String str = "Hello";)
- **Arrays** → Collection of elements of the same type (int arr[] = {1,2,3};)
- **Classes** → Blueprint for creating objects
- **Interfaces** → Abstractions for classes
- **Enums** → Special type for fixed set of constants

Keypoints:

- Primitive types are predefined by Java and stored directly in memory.
- Reference types are created by the programmer and store references (not actual values).
- String is a class, not a primitive, but used like one because of its importance.

Example :

```
Public class DataTypesExample {
    public static void main(String[] args) {
        // Primitive Data Types
        byte b = 100;
        short s = 2000;
        int i = 100000;
        long l = 15000000000L;
        float f = 3.14f;
        double d = 19.99;
        char c = 'A';
        boolean flag = true;
// Printing primitive types
        System.out.println("Byte Value: " + b);
        System.out.println("Short Value: " + s);
        System.out.println("Int Value: " + i);
        System.out.println("Long Value: " + l);
        System.out.println("Float Value: " + f);
        System.out.println("Double Value: " + d);
        System.out.println("Char Value: " + c);
        System.out.println("Boolean Value: " + flag);
```

```
// Non-Primitive Data Types
        String str = "Hello, Java!";
        int[] numbers = {10, 20, 30, 40};

        System.out.println("\nString Value: " + str);

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

Output:

```
Byte Value: 100
Short Value: 2000
Int Value: 100000
Long Value: 15000000000
Float Value: 3.14
Double Value: 19.99
Char Value: A
Boolean Value: true

String Value: Hello, Java!
Array Values: 10 20 30 40
```

Java Operators

- Operators are used to perform operations on variables and values.

- We can divide all the Java operators into the following groups −

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Conditional Operator
- new operator
- instanceof operator

Arithmetic Operators

Assume integer variable A holds 10 and variable B holds 20, then −

Operator	Description
+ (Addition)	Adds values on either side of the operator.
- (Subtraction)	Subtracts right-hand operand from left-hand operand.
* (Multiplication)	Multiplies values on either side of the operator.
/ (Division)	Divides left-hand operand by right-hand operand.
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.
++ (Increment)	Increases the value of operand by 1.
-- (Decrement)	Decreases the value of operand by 1.

Relational Operators

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Bitwise Operators

- Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation.

Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

Operator	Description
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Logical Operators

- The following table lists the logical operators −
- Assume Boolean variables A holds true and variable B holds false, then −

Operator	Description
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Java AND Operator Example: Logical && vs Bitwise &

```
class OperatorExample
{
public static void main(String args[])
{
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);
System.out.println(a<b&a++<c);
}
}
```

Java Left Shift Operator Example

```
class OperatorExample
{
public static void main(String args[])
{
System.out.println(10<<2);
System.out.println(10<<3);
System.out.println(20<<2);
System.out.println(15<<4);
}
}
```

Java Right Shift Operator Example

```
class OperatorExample
{
public static void main(String args[])
{
System.out.println(10>>2);
System.out.println(20>>2);
System.out.println(20>>3);
}
}
```

- a << b means shifting the **binary representation of a** to the left by b positions. – **Left Shift - Concepts**
- This is **equivalent to multiplying a by 2^b.**

Step-by-step Execution:

- **10 << 2**
  - 10 * 2^2 = 10 * 4 = 40
- **10 << 3**
  - 10 * 2^3 = 10 * 8 = 80
- **20 << 2**
  - 20 * 2^2 = 20 * 4 = 80
- **15 << 4**
  - 15 * 2^4 = 15 * 16 = 240

Output:

40

80

80

240

Concept of >> (Right Shift)

- a >> b means shifting the **binary representation of a** to the right by b positions.
- This is **equivalent to dividing a by 2^b** (ignoring remainder for integers).
- Important: >> preserves the sign bit (arithmetic shift).

Step-by-step Execution:

- **10 >> 2**
  - 10 / 2^2 = 10 / 4 = 2
- **20 >> 2**
  - 20 / 2^2 = 20 / 4 = 5
- **20 >> 3**
  - 20 / 2^3 = 20 / 8 = 2

Output:

2

5

2

Assignment Operators

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A

Conditional Operator (? :)

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as −

variable x = (expression) ? value if true : value if false

```
public class Test
{
public static void main(String args[])
{
    int a, b;
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );
     b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

Unary Operator

The Java unary operators require only one operand.

incrementing/decrementing a value by one

Java Unary Operator Example: ++ and --

```
class OperatorExample
{
        public static void main(String args[])
{
        int x=10;
        System.out.println(x++); //10 – then it is 11
        System.out.println(++x);  //12
        System.out.println(x--);  //12 –then it is 11
      System.out.println(--x);  //10
        }
}
```

Output:

```
10
12
12
10
```

Java program - pre-increment (++A) and post-increment (A++)

```
public class IncrementDemo {
    public static void main(String[] args) {
        int A = 5;

        // Post-increment (A++)
        int postResult = A++;  // A is used first, then incremented
        System.out.println("Post-increment:");
        System.out.println("Value assigned to postResult = " + postResult);
        System.out.println("Value of A after post-increment = " + A);

        System.out.println("----------------------");

        // Reset A
        A = 5;

        // Pre-increment (++A)
        int preResult = ++A;
        System.out.println("Pre-increment:");
        System.out.println("Value assigned to preResult = "
        + preResult);
        System.out.println("Value of A after pre-increment
        = " + A);
    }
}
```

```
Post-increment:
Value assigned to postResult = 5
Value of A after post-increment = 6
----------------------
Pre-increment:
Value assigned to preResult = 6
Value of A after pre-increment = 6
```

new operator

Example:

```
public class NewExample2
{
   NewExample2()
    {
       System.out.println("Invoking Constructor");
    }
     public static void main(String[] args)
{
       NewExample2 obj=new NewExample2();


    }

}
```

instanceof operator

Ex:1

```
 class Simple1
{
 public static void main(String args[])
{
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);
 }
}
```

o/p - True

Ex:2

```
class Main {
    public static void main (String[] args) {
        String name = "Programiz";
        Integer age = 22;

        System.out.println("Is name an instance of String: "+ (name
instanceof String));
        System.out.println("Is age an instance of Integer: "+ (age
instanceof Integer));
    }
}
```

Java Operator Precedence Table

Level	Operator	Description	Associativity
16	[] . ()	access array element access object member parentheses	left to right
15	++ --	unary post-increment unary post-decrement	not associative
14	++ -- + - ! ~	unary pre-increment unary pre-decrement unary plus unary minus unary logical NOT unary bitwise NOT	right to left
13	() new	cast object creation	right to left
12	* / %	multiplicative	left to right
11	+ - +	additive string concatenation	left to right
10	<< >> >>>	shift	left to right
9	< <= > >= instanceof	relational	not associative
8	== !=	equality	left to right
7	&	bitwise AND	left to right
6	^	bitwise XOR	left to right
5	|	bitwise OR	left to right
4	&&	logical AND	left to right
3	||	logical OR	left to right
2	?:	ternary	right to left
1	= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	right to left

Operator precedence

- example, x = 7 + 3 * 2 = ?

- Associativity

1. example:
x = y = z = 17 is treated as x = (y = (z = 17))
-- right-to-left associativity

2. example:
72 / 2 / 3 is treated as (72 / 2) / 3 – 36/3 -- 12
-- left-to-right associativity

Apply Operator precedence and Associativity, to find the output of the expressions

Expression	Value
6 + 2 * 3	?
8 / 2 - 3	?
8 + 10 * 2 - 4	?
4 + 11 % 2 - 1	?
6 - 3 * 3 + 7 - 1	?

Example :

Swap two numbers without using a third variable:

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

        System.out.println("Before swapping: a = " + a + ", b = " + b);

        // Swapping without third variable
        a = a + b;  // a = 30
        b = a - b;  // b = 10
        a = a - b;  // a = 20

        System.out.println("After swapping: a = " + a + ", b = " + b);
    }
}
```

Wrapper classes

In Java, Wrapper classes are classes that provide a way to use primitive data types (int, char, boolean, etc.) as objects.

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

Features of Wrapper Classes:

1. Convert primitives to objects (Autoboxing).
2. Convert objects to primitives (Unboxing).
3. Automatic Conversion.

Example 1:

```
public class WrapperExample {
    public static void main(String[] args) {
        int num = 10;

        // Autoboxing: converting primitive to object
        Integer obj = num;

        // Unboxing: converting object to primitive
        int value = obj;

        System.out.println("Primitive: " + num);
        System.out.println("Wrapper Object: " + obj);
        System.out.println("Unboxed Value: " + value);
    }
}
```

Output:

```
Primitive: 10
Wrapper Object: 10
Unboxed Value: 10
```

Java program to convert a string into primitive data types (int, double, and boolean) using Wrapper classes:

Example 2:

```
public class StringToPrimitive {
    public static void main(String[] args) {
        // String values
        String intStr = "100";
        String doubleStr = "45.67";
        String booleanStr = "true";

        // Converting using wrapper classes
        int intValue = Integer.parseInt(intStr);
        double doubleValue = Double.parseDouble(doubleStr);
        boolean booleanValue = Boolean.parseBoolean(booleanStr);

        // Display results
        System.out.println("String to int: " + intValue);
        System.out.println("String to double: " + doubleValue);
        System.out.println("String to boolean: " + booleanValue);
    }
}
```

Output:

```
String to int: 100
String to double: 45.67
String to boolean: true
```

Variable Shadowing

Variable shadowing happens when a local variable has the same name as an instance or class variable, and the local one hides the outer one.

Example :

```
public class VariableShadowing {
    // Instance variable
    int num = 50;

    public void display() {
        // Local variable with the same name
        int num = 100;

        System.out.println("Local variable num = " + num);
        System.out.println("Instance variable num = " + this.num);
    }

    public static void main(String[] args) {
        VariableShadowing obj = new VariableShadowing();
        obj.display();
    }
}
```

Output:

Local variable num = 100
Instance variable num = 50

Final keyword

- When a variable is declared as final, its value cannot be changed once initialized - It becomes a constant.
- In the program:final int MAX_VALUE = 100; → once set, MAX_VALUE can't be modified.
- If one tries to assign a new value (MAX_VALUE = 200;), the compiler - throws an error.

```
public class FinalVariableDemo {
    // final variable
    final int MAX_VALUE = 100;

    public void display() {
        System.out.println("The final variable MAX_VALUE = " + MAX_VALUE);

        // Uncommenting the below line will cause a compilation error
        // because final variables cannot be reassigned
        // MAX_VALUE = 200;
    }

    public static void main(String[] args) {
        FinalVariableDemo obj = new FinalVariableDemo();
        obj.display();
    }
}
```

Output:
The final variable
MAX_VALUE = 100

Questions on Data Types, Variables and Operators

1. Data Types

- Write a Java program to display the default values of all primitive data types.
- **Write a Java program to swap two numbers without using a third variable.**
- **Write a Java program to convert a string into a primitive data type (int, double, boolean) using wrapper classes.**

2. Variables

- Write a Java program to illustrate the difference between **local, instance, and static variables**.
- Write a Java program to calculate factorial using a static variable for counting function calls.
- Write a Java program to check the scope of a variable inside nested blocks
- **Write a Java program to demonstrate variable shadowing**.
- **Write a Java program to demonstrate the use of a final variable**.

3.Operators

- Explain the working of the ternary operator in Java.

new Keyword / Operator

- **Syntax:**

Class_Name class_Variable = new Class_Name();

- **Example:**

NewExample obj=**new** NewExample();

- **Points to remember about new keyword**

- A simple example to create an object using new keyword and invoking the method using the corresponding object reference:

```
public class NewExample1
{
   void display()
   {
      System.out.println("Invoking Method");
   }
  public static void main(String[] args)
{
      NewExample1 obj=new NewExample1();
      obj.display();
   }

}
```

- A simple example to create an object using new keyword and invoking the constructor using the corresponding object reference:

```
public class NewExample2
{
  NewExample2()
    {
       System.out.println("Invoking Constructor");
    }
  public static void main(String[] args)
{
       NewExample2 obj=new NewExample2();


    }


}
```

- An example to create an array object using the new keyword:

```
 public class NewExample4
{
  static int arr[]=new int[3];
  public static void main(String[] args)
{
      System.out.println("Array length: "+arr.length);
   }

}
```

Question:

Create a Java program to implement all the functionalities of “new” operator

Type Casting

Type casting in **Java** means converting a variable of one data type into another. Java supports two kinds of casting:

1. Widening Casting (Implicit Conversion)

- Automatically done by Java.
- Converts **a smaller type into a larger type.**
- **No data loss**.
- **Order of widening (small → large):**
  byte → short → int → long → float → double

```
public class WideningExample {
    public static void main(String[] args) {
        int num = 100;
        double d = num;  // int to double (automatic)
        System.out.println("Integer: " + num);
        System.out.println("Double: " + d);
    }
}
```

Output:

```
Integer: 100
Double: 100.0
```

2. Narrowing Casting (Explicit Conversion)

- Must be **done manually** using parentheses.
- Converts a **larger type into a smaller type.**
- **Might lose data or cause overflow**.

```
public class NarrowingExample {
    public static void main(String[] args) {
        double d = 9.78;
        int num = (int) d;  // double to int (manual)
        System.out.println("Double: " + d);
        System.out.println("Integer: " + num);
    }
}
```

Output:

```
Double: 9.78
Integer: 9
```

Type Casting with Objects

- Java also allows casting between **parent and child classes** (upcasting and downcasting).
- **Upcasting (implicit): Child → Parent**
- **Downcasting (explicit): Parent → Child**

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

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

```
public class CastingExample
{
    public static void main(String[] args)
{
        Animal a = new Dog();     // Upcasting
        a.sound();                // Dog barks

        Dog d = (Dog) a;          // Downcasting
        d.sound();                // Dog barks
    }
}
```

Example:

```
public class CastingHierarchy {
    public static void main(String[] args) {
        byte b = 10;
        int i = b;          // widening: byte → int
        double d = i;       // widening: int → double
        int x = (int) d;    // narrowing: double → int

        char c = 'A';       // 'A' = 65 in ASCII/Unicode
        int ci = c;         // widening: char → int

        System.out.println("byte: " + b);
        System.out.println("int (from byte): " + i);
        System.out.println("double (from int): " + d);
        System.out.println("char: " + c);
        System.out.println("char to int: " + ci);
        System.out.println("narrowed double to int: " + x);
    }
}
```

Output:

```
byte: 10
int (from byte): 10
double (from int): 10.0
char: A
char to int: 65
narrowed double to int: 10
```