# PROGRAMMING IN JAVA

Dr.K.Kasturi
Dr.J.Jebathangam
Dr.K.Rohini
Dr.R.Bhuvana

AN Publications

# Authors

**Dr.K.Kasturi M.Sc.,MCA.,M.Phil.,Ph.D., is working as an Associate Professor in the Department of Applied Computing & Emerging Technologies, School of Computing Sciences,Vels Institute of Science Technology and Advanced Studies(VISTAS), Chennai, India. Her experience includes, as a teaching faculty for more than nineteen years. As part of the research work, she has published more than 40 articles in national and international journals.**

**Dr.J.Jebathangam has completed her MCA degree from University of Madras, Tamil Nadu, India and M.P hil., degree from Vinayaka Missions University, Salem,Tamil Nadu, India. She has completed her Ph.D., degree from Mother Teresa Women's University, Tamil Nadu, India. She is currently working as Associate Professor in the department of Information Technology, School of Computing Sciences, Vels Institute of Science and Technology,Chennai.**

**Dr.K.Rohini received her MCA from University of Madras and M.Phil., degree from Annamalai University, Tamil Nadu, India . She received Ph.D., degree in computer science in Vels Institute of Science, Technology & Advanced Studies, Tamil Nadu, India. She is currently working as Professor, Department of Information Technology, School of Computing Sciences, Vels Institute of Science, Technology & Advanced Studies (VISTAS), Chennai.**

**Dr R.BHUVANA has completed her MSC,M.Phil,MCA., from Bharathidasan University. She has completed her Ph.D. in computer science from VISTAS, Chennai. She has Qualified NTA NET in Dec 2021 and currently working as Associate Professor in the Department of Computer Science, AM Jain College Chennai.**

978-81-989017-6-7

**AN Publications**

**No 29, Moorthy Street**
**Balavinayagar nagar, Tiruvallur-602001**
**Tamilnadu, India**
**Contact 9087236988**

**WWW.anpublication.com**

# Programming in Java

(For all UG, PG in BCA,BSc,MCA, M.Sc Students)

**Dr.K.Kasturi**
Associate Professor,
Dept. of Applied Computing and Emerging Technologies,
VISTAS,Chennai.Tamilnadu, India

**Dr.J.Jebathangam**
Professor,
Dept. of Computer Applications(UG),
VISTAS, Chennai, Tamil Nadu, India

**Dr.K.Rohini**
Professor,
Dept. of Computer Applications(UG),
VISTAS, Tamilnadu, India

**Dr.R.Bhuvana**
Associate Professor,
Department of Computer Science,
AM Jain College, Chennai, Tamilnadu, India

**AN PUBLICATIONS**
**ISO Certified International Publisher,Regd in MSME Govt.of India**
**No: 29,Moorthy Street,Balavinayagar nagar,**
**Tiruvallur-602001,**
**Tamilnadu,India**

**Title: Programming in Java**

**Authors: Dr.K.Kasturi, Dr.J.Jebathangam, Dr.K.Rohini, Dr.R.Bhuvana**

**ISBN: 978-81-989017-6-7**

**Published 2025 by AN PUBLICATIONS**



**Printing: VST press, Chennai.**



**AN Publications**

No:29, Moorthy street, Balavinayagar Nagar,

Tiruvallur-602001,Tamilnadu, India.

# Preface

Java has stood the test of time as one of the most reliable, versatile, and widely used programming languages in the world of software development. From desktop and web applications to enterprise systems, mobile apps, and cloud-based solutions, Java continues to play a pivotal role in shaping modern computing. This book is written with the objective of providing a clear, structured, and practical understanding of Java programming for students, educators, and aspiring software professionals.

The primary aim of this book is to introduce the core concepts of Java in a simple and progressive manner, beginning with fundamental programming constructs and gradually advancing to object-oriented principles, exception handling, multithreading, file handling, and modern Java features. Emphasis is placed not only on syntax but also on logical thinking, problem-solving skills, and best programming practices that are essential for real-world software development.

Each chapter is designed to be self-contained and learner-friendly, enriched with clear explanations, illustrative examples, and well-structured programs. Wherever appropriate, real-time scenarios and use cases are included to help readers relate theoretical concepts to practical applications. Review questions and exercises at the end of each chapter encourage self-assessment and deeper understanding.

This book is especially intended to support undergraduate and postgraduate students of computer applications and computer science, including MCA and related programs, as well as beginners who wish to build a strong foundation in Java. Faculty members may also find this book useful as a teaching aid aligned with academic curricula.

We hope that this book will serve as a reliable companion in your journey to mastering Java programming and inspire you to explore advanced technologies built upon the Java platform.

***Authors***

# Authors

**Dr.K.Kasturi** M.Sc.,MCA.,M.Phil.,Ph.D., is working as an Associate Professor in the Department of Applied Computing & Emerging Technologies, School of Computing Sciences,Vels Institute of Science Technology and Advanced Studies(VISTAS), Chennai, India. Her experience includes, as a teaching faculty for more than nineteen years. As part of the research work, she has published more than 40 articles in national and international journals, presented more than 15 papers in national and international conferences and her Google scholar citation is more than 50.She has published 6 patents including 1 granted. She has published 2 books and 8 book chapters in computer science and Application Streams. She has produced 2 research scholars. At present, she is guiding 6 research scholars. She has received the Young Researcher award, excellence in Teaching award and Best faculty award. She is an active member in Research Gate Society and her reads crosses more than 7,800.She is a life time member in various professional bodies including I2OR.Her research area includes Artificial Intelligence, Machine Learning,Deep Learning, Data Mining,Computer Networks, Network Security, Block chain, Internet of Things and Image Processing. My hearty thanks to my family and VISTAS family for supporting me to complete the work.

**Dr.J.Jebathangam** has completed her MCA degree from University of Madras, Tamil Nadu, India and M.P hil., degree from Vinayaka Missions University, Salem,Tamil Nadu, India. She has completed her Ph.D., degree from Mother Teresa Women's University, Tamil Nadu, India. She is currently working as Associate Professor in the department of Information Technology, School of Computing Sciences, Vels Institute of Science and Technology,Chennai. She has 16+ years of teaching experience in both UG and PG level. Her areas of specialization are Image Processing and Neural Networks. She has produced 5 M.phil., scholars and 3 PhD scholars . As an active participant in research she has published more than 50

papers in various international journals out of which 17 are scopus indexed journals and have also published 3 books.

**Dr.K.Rohini** received her MCA from University of Madras and M.Phil., degree from Annamalai University, Tamil Nadu, India . She received Ph.D., degree in computer science in Vels Institute of Science, Technology & Advanced Studies, Tamil Nadu, India. She is currently working as Professor, Department of Information Technology, School of Computing Sciences, Vels Institute of Science, Technology & Advanced Studies (VISTAS), Chennai, Tamil Nadu. She has 18 years of teaching experience in both UG and PG Level. Her research interests include Data Mining , IOT, , Cloud Computing, Image Processing, Bigdata analytics , Machine learning and Cybersecurity. She has produced 8 M.Phil., Scholars and 10 Ph.D., Research Scholars awarded under her Guidance and Supervision.

She has published more than 42 research papers including in various International Journal such in Science Citation Index, IEEE Access, Springer Book Chapter, Scopus, and UGC referred journals. She has presented many papers in various international Conferences and received 6 awards.

**Dr R.BHUVANA** has completed her MSC,M.Phil,MCA., from Bharathidasan University. She has completed her Ph.D. in computer science from VISTAS, Chennai. She has Qualified NTA NET in Dec 2021 and currently working as Associate Professor in the Department of Computer Science, AM Jain College Chennai. She has 17 years of experience in handling both UG & PG level Courses. Her area of specialization is Artificial Neural Network, Machine Learning and Data Mining. She has published more than 35 journals in which 6 are Scopus indexed. She has presented more than 35 articles in National & International Conferences and also published 8 articles in book chapters.

**CONTENTS** **Pg. No**

# Chapter -1 FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

## 1.1 Paradigms of Programming Languages

A **programming paradigm** is a style or way of programming. It defines how programs are structured and how problems are approached. Common paradigms include:

- **Procedural Programming:** Based on the concept of procedures or routines. Example: C, Pascal.
- **Object-Oriented Programming (OOP):** Based on the concept of objects and classes. Example: C++, Java.
- **Functional Programming:** Emphasizes functions without changing states. Example: Haskell, Lisp.
- **Logic Programming:** Based on formal logic. Example: Prolog.
- **Event-Driven Programming:** Responds to events or user actions. Example: JavaScript (for UI).

## 1.2 Procedural Oriented Programming vs. Object-Oriented Programming

| **Feature** | **Procedural Programming** | **Object-Oriented Programming** |
|---|---|---|
| **Approach** | Top-down | Bottom-up |
| **Focus** | Functions/Procedures | Objects and Classes |
| **Data Access** | Global access, less secure | Data encapsulation, more secure |
| **Reusability** | Low | High due to classes and inheritance |
| **Examples** | C, Pascal | Java, C++, Python |
| **Code Management** | Harder in large programs | Easier with modular structure |

## 1.3 Basic Concepts of Object-Oriented Programming (OOP)

- **Class**

A **class** is a blueprint or template for creating objects. It defines attributes (data) and behaviors (methods) that the objects created from it will have.

**Real-World Example:**
A **"Car"** class can define properties like color, brand, speed, and methods like drive(), brake(), accelerate().

- **Object**

An **object** is an instance of a class. It represents a real-world entity that has state and behavior.

**Real-World Example:**
From the "Car" class, we can create multiple **objects** like car1 and car2, each with different colors and brands.

- **Encapsulation**

**Encapsulation** is the concept of hiding internal details of an object and exposing only necessary parts. It binds data and methods that operate on the data within one unit.

**Protects data by:** public, protected, default and private access levels of data

**Real-World Example:**
Think of a **TV remote**. We press buttons without knowing the internal circuitry. The complexity is hidden.

- **Abstraction**

**Abstraction** means showing only essential features and hiding unnecessary details.

**Hides complexity by :** Abstract classes, interfaces

**Real-World Example:**
When we **send a message** on a phone, we don't know the inner workings (signal transmission, server handling). We only see the "Send" button.

- **Inheritance**

**Inheritance** allows one class (child or derived) to inherit properties and methods from another class (parent or base).

**Real-World Example:**
A **"Dog"** is an **"Animal"**. The "Dog" inherits common traits like eat(), sleep() from the "Animal" class but can have specific methods like bark().

- **Polymorphism**

**Polymorphism** means "many forms". It allows the same method name to behave differently based on context.

a) Compile-time Polymorphism (Method Overloading)

**Example:** Calculator with multiple add() methods.

b) Runtime Polymorphism (Method Overriding)

**Example:** Shape class with overridden draw() method in subclasses.

- **Message Passing**

Objects communicate by **sending messages** to each other (i.e., calling methods).

**Real-World Example:**
We (object) send a message to a **coffee machine** (another object) by pressing a button to make coffee (method call).

### 1.4 Benefits of Object Oriented Programming

Object-Oriented Programming is a programming paradigm centered around the concept of **objects**, which are instances of **classes**. Each object can contain data (fields) and code (methods). Here are the key benefits:

**Modularity**

- Programs are divided into independent units called classes.
- Each class is self-contained, making it easier to develop, test, and maintain.

**Reusability**

- Classes and objects can be reused across different programs.
- **Inheritance** allows new classes to be built from existing ones, reducing code duplication.

**Encapsulation**

- Data and methods are bundled together within objects.

- Internal details of objects are hidden using access modifiers, promoting **data security** and integrity.

**Scalability and Manageability**

- OOP systems are easier to scale and manage as they grow larger.
- New features can be added without affecting existing code significantly.

**Abstraction**

- Focuses on relevant data and hides complex implementation details.
- Helps in modeling real-world systems more naturally and intuitively.

**Polymorphism**

- Allows the same interface to be used for different underlying data types.
- Makes code more flexible and easier to extend..

**Ease of Maintenance and Upgrades**

- Modular code and encapsulation make it easier to identify bugs and update specific parts without affecting the entire program.

**Real-World Modeling**

- OOP concepts (objects, classes, inheritance) mirror real-world entities, making it easier to design and understand systems.

## 1.5 Applications of Object Oriented Programming

Object-Oriented Programming is widely used in modern software development due to its modularity, flexibility, and real-world modelling capabilities. Below are common application areas:

### Software Development

- Most modern applications are built using OOP languages like Java, C++, Python, and C#.
- **Example:** Enterprise applications (e.g., banking, HR systems) developed using Java.

### Graphical User Interface (GUI) Applications

- OOP simplifies the creation of components like buttons, windows, and forms as objects.
- **Example:** Java Swing, Python Tkinter, and Windows Forms in .NET.

### Web Development

- Backend web development frameworks are built using OOP principles.
- **Example:** Django (Python), Spring Boot (Java), Ruby on Rails (Ruby).

**Game Development**

- Characters, scenes, weapons, and other elements are represented as objects.
- **Example:** Unity (C#), Unreal Engine (C++).

**Mobile App Development**

- OOP is essential for Android (Java/Kotlin) and iOS (Swift/Objective-C) app development.
- Components like activities, views, and fragments are treated as objects.

**Simulation and Modeling**

- Real-world entities and processes are modeled as objects.
- **Example:** Traffic simulation systems, flight simulators.

**Artificial Intelligence and Machine Learning**

- ML models, data sets, and processing units are often represented as objects.
- OOP helps organize code in scalable AI frameworks like TensorFlow and PyTorch.

**Operating Systems**

- Many OS components are implemented using OOP.
- **Example:** File systems, process schedulers, and memory managers in C++.

**Embedded Systems**

- OOP helps manage complexity in microcontroller-based applications.
- **Example:** Smart appliances, automotive systems using C++ or Java.

# Chapter – 2 INTRODUCTION TO JAVA PROGRAMMING

## 2.1 Java Evolution:

- Java is a programming language created by James Gosling from Sun Microsystems (Sun) in 1991
- The first publicly available version of Java (Java 1.0) was released in
- 1995.
- Sun Microsystems was acquired by the Oracle Corporation in 2010.
- Over time new enhanced versions of Java have been released.
- The latest version of Java is Java Development Kit (JDK) is JDK 24, which was released on March 2025

### 2.1.1 Java Features

There are many features of java. They are also known as java buzzwords.

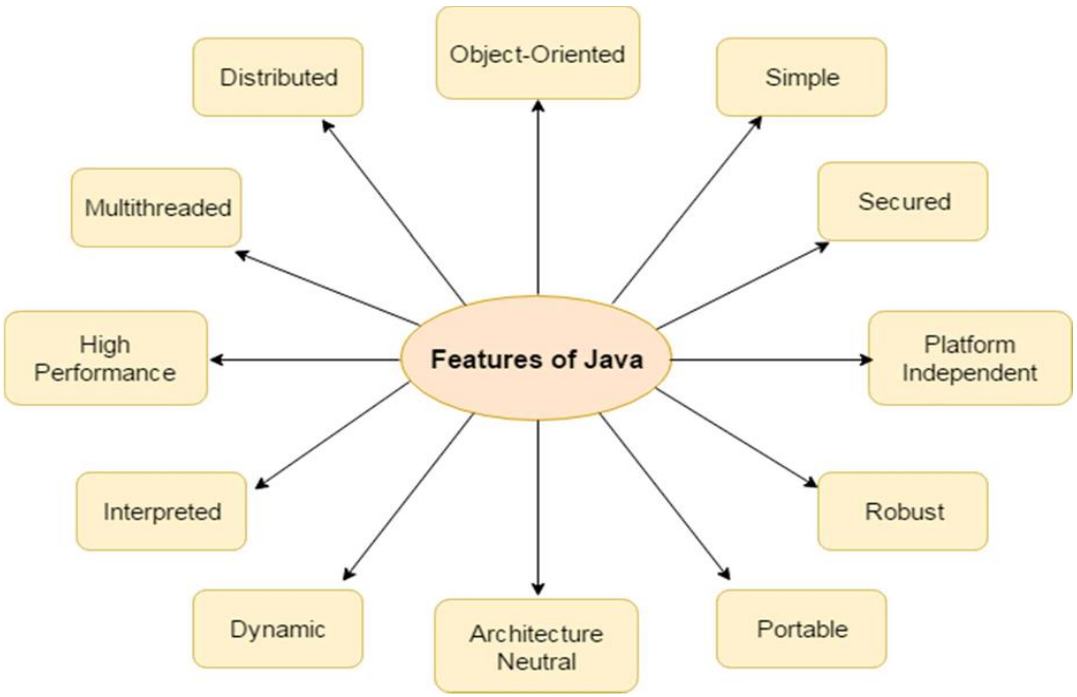



Fig: Java Features

1. **Simple**
    - Easy to learn, clean syntax, and avoids complex features like pointers.

2. **Object-Oriented**
    - Everything is based on classes and objects; supports OOP principles like abstraction inheritance, encapsulation polymorphism.
3. **Portable**
    - Java bytecode can run on any machine with a JVM – "Write Once, Run Anywhere".
4. **Platform Independent**
    - Java programs are compiled to bytecode, which is platform-independent (runs on any OS with JVM).
5. **Secured**
    - Provides runtime security through no pointer access, bytecode verification, and Security Manager.
6. **Robust**
    - Handles memory management automatically and provides strong exception handling.
7. **Architecture Neutral**
    - Bytecode is the same across all hardware and operating systems.
8. **Dynamic**
    - Classes are loaded at runtime as needed, using tools like ClassLoader and reflection.
9. **Interpreted**
    - Bytecode is interpreted/executed by the JVM at runtime (with JIT compilation).
10. **High Performance**

- Faster than traditional interpreted languages due to the Just-In-Time (JIT) compiler.

11. **Multithreaded**

- Supports multiple threads of execution simultaneously for better performance.

12. **Distributed**

- Supports development of distributed applications via RMI, sockets, and web services.

### 2.1.2 Difference between Java, C and C++

| Feature | C | C++ | Java |
|---|---|---|---|
| **Paradigm** | Procedural | Object-Oriented + Procedural | Pure Object-Oriented (mostly) |
| **Platform Dependency** | Platform dependent (compiled) | Platform dependent (compiled) | Platform independent (JVM-based) |
| **Memory Management** | Manual | Manual | Automatic (Garbage Collection) |
| **Pointers** | Supported | Supported | Not supported |
| **Multiple Inheritance** | Not supported | Supported (via classes) | Supported via interfaces |
| **Security** | Low | Moderate | High |
| **Speed** | Fast | Fast | Slower (due to JVM) |
| **Use Case** | System programming | Game engines, system software | Web, mobile, enterprise apps |

### 2.1.3 Java and Internet

**Why Java is good for the Internet**

1. **Runs Anywhere**
   Java programs can work on any computer or mobile — Windows, Mac, Android — because of a special tool called the JVM (Java Virtual Machine).

2. **Safe and Secure**
   Java is built to be safe. It can stop harmful programs from doing damage, which is important when sharing code over the internet.

3. **Easy to Connect**
   Java has tools to connect computers over the Internet. It can send and receive data using networking protocols like HTTP and FTP.

4. **Supports Web Applications**
   Java can create programs that run on websites (like Servlets and JSPs) or help build big web services (like Spring Boot apps).

**Java Tools for the Internet**

| **Tool** | **What It Does** |
|---|---|
| java.net | Helps Java programs connect to the Internet. |
| Servlets | Java programs that run on a web server. |
| JSP (Java Server Pages) | Lets us add Java to HTML web pages. |
| Spring Boot | Helps make websites and online services easily. |

**Examples of Java on the Internet**

- Online banking apps
- Shopping websites
- Mobile apps using the Internet
- Web servers like Tomcat

### 2.1.4 Java Environment

**J**ava Development Environment Includes:

**Java Development Kit (JDK)**

- Provides command line tools to compile, run, debug Java applications.
- Includes: javac, java, javadoc, jar.

javac Compiles Java source code javac MyProgram.java

java Runs compiled Java programs java MyProgram

javadoc Creates HTML documentation javadoc MyProgram.java

jar Packages Java files into a (**J**ava **AR**chive) jar cf MyApp.jar *.class

**Java Runtime Environment (JRE)**

- Includes JVM + standard libraries.
- Required to **run** Java applications.

**Java Virtual Machine (JVM)**

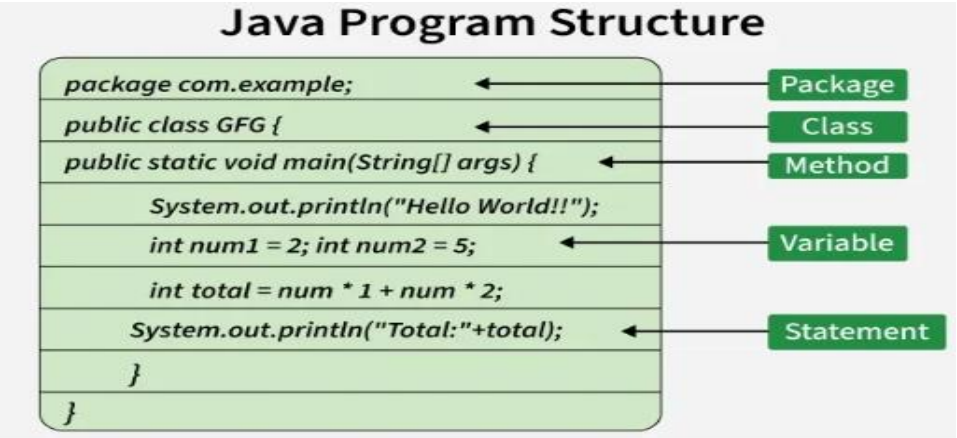
- Executes bytecode (.class files).
- Platform-specific implementation of Java's virtual execution environment.
- The **JIT compiler** is a part of the **Java Virtual Machine (JVM)** that improves the **performance** of Java programs by **compiling bytecode into native machine code at runtime**.

**Java Source Code (.java)**
**↓ javac**
**Bytecode (.class)**
**↓ JVM (with JIT)**
**Native Machine Code**
**↓**
**Fast Execution**

## 2.2 Overview of Java Language:

### 2.2.1 Java Program Structure

A Java program follows a specific structure that makes it **organized**, **modular**, and **easy to read**. Every valid Java program includes a class and a main() method, which serves as the **entry point** of execution.



**Explanation:**

| Part | Description |
|---|---|
| **Package Declaration** | Organizes the class into a namespace (optional). |
| **Import Statements** | Brings in other Java classes/libraries. |
| **Class Definition** | Every Java program must be inside a class. |
| **Main Method** | Entry point of the program. JVM starts here. |
| **Statements** | Actual program logic, variable declarations, method calls, etc. |

**Example:**

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

**Output:**

Welcome to Java Programming!

### 2.2.2 Simple Java Programs

**1.Addition of Two Numbers**

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

**Output:**

Sum = 30

**2. Find the Largest of Two Numbers**

```
public class LargestOfTwo
{
   public static void main(String[] args)
   {
      int a = 25, b = 40;
      if(a > b)
      {
         System.out.println("Largest = " + a);
      }
      else
      {
         System.out.println("Largest = " + b);
      }
   }
}
```

**Output:**

Largest = 40

**3. Check Even or Odd**

```
public class EvenOdd
{
   public static void main(String[] args)
   {
      int num = 7;
      if(num % 2 == 0)
      {
         System.out.println(num + " is Even");
      }
      else
      {
         System.out.println(num + " is Odd");
```

```
        }
    }
}
```

**Output:**
7 is Odd

**4. Print Numbers from 1 to 5 (Using Loop)**

```
public class PrintNumbers
{
    public static void main(String[] args)
    {
        for(int i = 1; i <= 5; i++)
        {
            System.out.println(i);
        }
    }
}
```

**Output:**
1
2
3
4
5

**5. Simple Program using Scanner (User Input)**

```
import java.util.Scanner;

public class UserInputExample
{
    public static void main(String[] args)
  {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = sc.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

**Output:**

Enter your name: Kasturi
Hello, Kasturi!

### 2.2.3 Usage of Comments

**Comments** in Java are **non-executable lines** in the code used to:

- Explain or document the code.
- Improve readability.
- Temporarily disable parts of code.
- Generate API documentation (using special comments).

**Types of Comments in Java**

| **Type** | **Syntax** | **Purpose** |
|---|---|---|
| **Single-line** | // comment | Quick inline notes |
| **Multi-line** | /* comment block */ | Longer explanations or block-level comments |
| **Documentation** | /** doc comment */ | Used by javadoc tool for API documentation |

**Single-Line Comment Example:**

```
public class SingleLineComment
{
   public static void main(String[] args)
{
      // This is a Single line comment
      System.out.println("Hello Java");
   }
}
```

**Output:**

Hello Java

**Multi-Line Comment Example:**

```
public class MultiLineComment
{
   public static void main(String[] args)
  {
       /*
        This program prints a welcome message.
        It is written using a multi-line comment.
       */
       System.out.println("Welcome to Java!");
   }
}
```

**Output:**

Welcome to Java!

**Documentation Comment** (**javadoc**) **Example:**

```
/**
 * This class demonstrates the use of documentation comments.
 * It prints a simple message.
 */
public class DocCommentExample {
    /**
     * This is the main method that starts the program.
     * @param args Command-line arguments
     */
    public static void main(String[] args) {
        System.out.println("JavaDoc Example");
    }
}
```

**Output:**

JavaDoc Example

**We can generate documentation using:**

javadoc DocCommentExample.java

**Best Practices for Comments**

- Use **single-line comments** for brief notes.
- Use **multi-line comments** for code blocks
- Use **JavaDoc comments** to document classes, methods, and parameters.

**2.2.4 Java Tokens and statements.**

**Java Tokens**

**Tokens** are the **smallest units** in a Java program. They are like **building blocks** that make up the code.
**There are** 6 types of tokens **in Java.**

| Token Type | Description | Example |
|---|---|---|
| **1. Keywords** | Reserved words used by Java | class, public, if, new |
| **2. Identifiers** | Names given to variables, classes, methods | sum, Student, main |
| **3. Literals** | Constant values assigned to variables | 10, 'A', "Hello", true |
| **4. Operators** | Symbols that perform operations | +, -, *, ==, && |
| **5. Separators** | Symbols that separate code components | (), {}, ;, , |
| **6. Comments** | Notes ignored by the compiler | //, /* */, /** */ |

**Example**

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

```
 {
      int num = 10; // variable declaration
      System.out.println("Number: " + num); // Prints output
   }
}
```

**Tokens Breakdown:**

- public, class, static, void – **keywords**
- Demo, main, args, num – **identifiers**
- 10, "Number: " – **literals**
- =, + – **operators**
- {}, (), ; – **separators**
- // – **comment**

**Java Statements**

A **statement** is a complete **instruction** in Java that performs some action.Each statement usually ends with a **semicolon (;)**.

**Types of Java Statements**

| **Type** | **Description** | **Example** |
|---|---|---|
| **Declaration** | Declares a variable | int a; |
| **Assignment** | Assigns a value | a = 10; |
| **Expression** | Performs a calculation | b = a + 5; |
| **Control Statements** | Controls flow: if, for, while | if(a > 5) { ... } |
| **Method Call** | Calls a method | System.out.println("Hello"); |
| **Block** | Group of statements inside {} | { int a = 10; int b = 20; } |

**Example Java Program with Statements**

public class StatementExample

```
{
   public static void main(String[] args)
{
      int x = 5;              // declaration & assignment
      int y = x + 10;         // expression
      if (y > 10)             // control statement
      {
         System.out.println(y); // method call
      }
   }
}
```

**Output:**

15

### 2.2.5 Command Line Arguments

In Java, **command line arguments** are the inputs passed to the main() method when a program is run from the **command prompt** or **terminal**.

**Syntax of main() with arguments**

public static void main(String[] args)

- args is an **array of Strings** that stores command-line inputs.
- Each argument is stored as a **String** in the args[] array.

**Example Program**

**public class CommandLineExample**

```
{
   public static void main(String[] args)
{
      if(args.length == 2)
      {
         System.out.println("First Argument: " + args[0]);
         System.out.println("Second Argument: " + args[1]);
      }
```

```
else {
      System.out.println("Please provide exactly 2 arguments.");
       }
   }
}
```

**How to Run this Program?**

**Steps in Command Prompt:**

1. **Compile:** javac CommandLineExample.java

2. **Run:** java CommandLineExample Java Programming

**Output:**
First Argument: Java
Second Argument: Programming

- Arguments are **always Strings**, even if we input numbers. We can convert them using:
    - Integer.parseInt(args[0]) for int
    - Double.parseDouble(args[0]) for double
- args.length gives the number of arguments passed.
- If no arguments are passed, args.length == 0.

**Example with Integer Conversion**

```
public class SumArgs {
   public static void main(String[] args) {
      if(args.length == 2) {
         int a = Integer.parseInt(args[0]);// parseInt() function Converts String to Integer
         int b = Integer.parseInt(args[1]);
         int sum = a + b;
         System.out.println("Sum = " + sum);
      } else {
         System.out.println("Enter exactly 2 integer values.");
      }
   }
}
```

**Run:**

java SumArgs 10 20

**Output:**

Sum = 30

# Chapter – 3 CONSTANTS, VARIABLES AND DATA TYPES

## 3.1 Constants

In Java, constants are variables whose value cannot be changed once assigned. They are declared using the final keyword.

### Why Use Constants?

To store fixed values (like Pi = 3.14)

To make the code more readable

To prevent accidental changes in important values

### Syntax

final dataType CONSTANT_NAME = value;

Note: By convention, constant names are written in uppercase letters with words separated by underscores.

### Example

```
public class ConstantExample
{
   public static void main(String[] args)
 {
       final int MAX_MARKS = 100;
       final double PI = 3.14159;
       final String COMPANY_NAME = "ABC Company";
       System.out.println("Max Marks: " + MAX_MARKS);
       System.out.println("Value of Pi: " + PI);
       System.out.println("Company: " + COMPANY_NAME);
   }
}
```

**Output**

Max Marks: 100

Value of Pi: 3.14159

Company: ABC Company

**Key Points**

- Constants must be initialized at the time of declaration.
- Once declared, their value cannot be changed.
- Trying to assign a new value to a constant will cause a compile-time error.

**Invalid Usage**

```
final int MAX_AGE = 60;
MAX_AGE = 70;  //  Error: cannot re-assign a value to final variable
```

### 3.2 Variables

A **variable** in Java is a **named memory location** used to store data. The value stored in a variable can be **changed during program execution**.

**Types of Variables:**

1. **Local Variables**
   - Declared inside methods, constructors, or blocks
   - Accessible only within that block
   - Must be initialized before use
2. **Instance Variables**
   - Declared inside a class but **outside methods**
   - Each object gets its own copy
3. **Static Variables (Class Variables)**
   - Declared using the static keyword
   - Shared by all objects of the class

**Syntax:**

```
dataType variableName;          // Declaration
dataType variableName = value; // Declaration + Initialization
```

**Rules for Naming Variables**

- Must start with a letter, $, or _
- Cannot start with a digit
- No spaces allowed
- Case-sensitive (e.g., age and Age are different)
- Avoid using Java keywords as variable names

**Multiple Variable Declarations:**

int a, b = 10, c = 20;

**Note:**

 Static Variable (Class-level)  Instance Variable (Object-level)  Local Variable (Method-level)

**1. Static Variable Example**

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

   StaticExample() {
      count++; // Shared among all objects
      System.out.println("count is: " + count);
   }

   public static void main(String[] args) {
      StaticExample obj1 = new StaticExample();  // count is: 1
      StaticExample obj2 = new StaticExample();  // count is: 2
      StaticExample obj3 = new StaticExample();  // count is: 3
   }
}
```

**Output:**
count is: 1
count is: 2
count is: 3

count is shared by **all objects**. It's declared using static

**2. Instance Variable Example**

```
public class InstanceExample
 {
    int count = 0;  // instance variable

   InstanceExample() // constructor method
   {
      count++; //  Not common for all objects
      System.out.println("count is: " + count);
   }

   public static void main(String[] args)
  {
      InstanceExample obj1= new InstanceExample();  // count is: 1
      InstanceExample obj2= new InstanceExample();  // count is: 1
      InstanceExample obj3= new InstanceExample();  // count is: 1
   }
}
```

**Output:**

```
count is: 1
count is: 1
count is: 1
```

**3. Local Variable Example**

```
public class LocalExample
{
   public void display()
     {
      int number = 10;  // Local variable
      System.out.println("Number: " + number);
     }

   public static void main(String[] args)
   {
      LocalExample obj = new LocalExample();
```

```
        obj.display();  // Number: 10
    }
}
```

**Output:**
Number: 10

- number is a **local variable** — it exists **only inside the method** display().

### 3.3 Data Types

**Data types** in Java specify the **type of data** a variable can hold. Java is a **statically-typed language**, meaning every variable must be declared with a data type.

### Categories of Data Types

Java data types are broadly classified into:

1. Primitive Data Types (8 types)
2. Non-Primitive (Reference) Data Types

### 1. Primitive Data Types

There are **8 primitive types**, divided into 4 groups:

**Integer Types**

| Type | Size | Range | Example |
|---|---|---|---|
| byte | 1 byte | -128 to 127 | byte a = 10; |
| short | 2 bytes | -32,768 to 32,767 | short s = 1000; |
| int | 4 bytes | -2^31 to 2^31–1 | int x = 5000; |
| long | 8 bytes | -2^63 to 2^63–1 | long l = 100000L; |

**Floating-Point Types**

| Type | Size | Precision | Example |
|---|---|---|---|
| float | 4 bytes | ~6-7 digits | float f = 3.14f; |
| double | 8 bytes | ~15 digits | double d = 3.14159; |

**Character Type**

| Type | Size | Description | Example |
|---|---|---|---|
| char | 2 bytes | Single character (Unicode) | char c = 'A'; |

Note: **Unicode** is a **universal character encoding standard**. It supports **all characters** from **all languages**: English, Hindi, Tamil, Chinese, Arabic, Emoji, Symbols, etc.

**Boolean Type**

| Type | Size | Values | Example |
|---|---|---|---|
| boolean | 1 bit | true/false | boolean b = true; |

**2. Non-Primitive (Reference) Data Types**

- Not predefined — created by the programmer.
- Refer to **objects**.
- Can store **multiple values**.

**Examples:**

- **String**: String name = "Kasturi";
- **Arrays**: int[] arr = {1, 2, 3};
- **Classes, Interfaces**

**Example Program**

```
public class DataTypeExample
{
```

```
   public static void main(String[] args)
  {
      byte b = 10;
      int i = 100;
      long l = 10000L;
      float f = 3.14f;
      double d = 3.141592;
      char c = 'A';
      boolean flag = true;
      String str = "Hello, Java!";
      System.out.println("Byte: " + b);
      System.out.println("Int: " + i);
      System.out.println("Long: " + l);
      System.out.println("Float: " + f);
      System.out.println("Double: " + d);
      System.out.println("Char: " + c);
      System.out.println("Boolean: " + flag);
      System.out.println("String: " + str);
   }
}
```

**Output:**

Byte: 10
Int: 100
Long: 10000
Float: 3.14
Double: 3.141592
Char: A
Boolean: true
String: Hello, Java!

### 3.4 Variables and Assigning values

**Variables:**

A **variable** in Java (or programming) is a **named storage location in memory** used to store data that can change during program execution.

A variable is a **container** that holds a value.

**Declaration and Assignment Separately**

We can declare a variable first and assign a value later:

```
int age;         // Declaration
age = 25;        // Assignment
```

**Declaration and Assignment Together**

This is the most common way:

```
int age = 25;    // Declaration + Assignment
```

**Assigning Multiple Variables**

We can assign values to multiple variables of the same type in one line:

```
int x = 10, y = 20, z = 30;
```

Or:

```
int a, b, c;
a = 5;
b = 10;
c = 15;
```

◆ **Re-assigning Values (for non-final variables)**

We can change the value of a variable later in the program:

```
int score = 50;
score = 75;  // New value assigned
```

◆ **Assigning Values from Expressions**

We can assign results of calculations:

```
int a = 10, b = 20;
int sum = a + b;        // sum = 30
int product = a * b;    // product = 200
```

**Example Program**

```
public class AssignExample
{
   public static void main(String[] args)
   {
      int age = 20;            // Direct assignment
      float marks = 85.5f;     // Float with 'f'
      char grade = 'A';        // Character
      boolean passed = true;   // Boolean
      String name = "ABC"; // String

      System.out.println("Name: " + name);
      System.out.println("Age: " + age);
      System.out.println("Marks: " + marks);
      System.out.println("Grade: " + grade);
      System.out.println("Passed: " + passed);
   }
}
```

**Output:**

Name: ABC

Age: 20

Marks: 85.5

Grade: A

Passed: true

### 3.5 Symbolic Constants

A **symbolic constant** is a name that represents a **fixed value** which **does not change** during the execution of a program. In Java, symbolic constants are created using the **final** keyword.

**Why Use Symbolic Constants?**

- Improves **readability**
- Makes code easier to **maintain**
- Prevents **accidental changes** of critical values
- Replaces **"magic numbers"** with meaningful names

**Syntax:**

final dataType CONSTANT_NAME = value;
Constant names are usually written in **ALL UPPERCASE** with underscores.

**Example:**

```
public class SymbolicConstantExample
{
   public static void main(String[] args)
  {
       final double PI = 3.14159;
       final int MAX_STUDENTS = 100;
       final String INSTITUTE = "AI Academy";

       System.out.println("Value of PI: " + PI);
       System.out.println("Max Students Allowed: " + MAX_STUDENTS);
       System.out.println("Institute Name: " + INSTITUTE);
   }
}
```

**Output**

```
Value of PI: 3.14159
Max Students Allowed: 100
Institute Name: AI Academy
```

### 3.6 Typecasting

**Typecasting** in Java refers to converting a value from one data type to another.
Java provides two types of typecasting:

#### 1. Widening Typecasting (Implicit)

- Also called **automatic type conversion**
- Converts a **smaller type to a larger type**
- No data loss, done **automatically** by the compiler

Example:

```
int num = 10;
double d = num;  // int to double
System.out.println("Double: " + d);  // Output: 10.0
```

Common Widening Conversions:
byte → short → int → long → float → double

#### 2. Narrowing Typecasting (Explicit)

- Also called **manual type conversion**
- Converts a **larger type to a smaller type**
- May lead to **data loss**
- Must be done **explicitly** using casting syntax

Example:

```
double d = 9.78;
int num = (int) d;  // double to int
System.out.println("Int: " + num);  // Output: 9 (decimal part lost)
```

#### Syntax of Typecasting

```
// Narrowing (manual)
dataType variable = (targetType) value;

// Widening (automatic)
dataType variable = value;
```

#### Example Program

```
public class TypeCastingExample {
   public static void main(String[] args) {
      // Widening
      int a = 100;
      long l = a;          // int to long
      float f = l;         // long to float

      // Narrowing
      double d = 75.89;
      int b = (int) d;     // double to int (decimal lost)

      System.out.println("Widened float: " + f);  // 100.0
      System.out.println("Narrowed int: " + b);   // 75
   }
}
```

**Output:**
Widened float: 100.0
Narrowed int: 75

**Summary Table:**

| **Type** | **Conversion Direction** | **Automatic?** | **Data Loss?** | **Example** |
|---|---|---|---|---|
| Widening | Smaller → Larger type | ✅ Yes | ❌ No | int → double |
| Narrowing | Larger → Smaller type | ❌ No | ✅ Possible | double → int |

# Chapter – 4 OPERATORS AND EXPRESSIONS

In Java, operators and expressions are fundamental components used to perform computations and manipulate data. Operators are special symbols that perform specific operations on one or more operands (variables, literals, or other expressions). They tell the Java compiler to perform a particular action. Expressions are fundamental components used to perform computations and manipulate data.

## 4.1 Arithmetic and Relational operators

### Arithmetic operators:

**Definition:**

Arithmetic operators are used to perform mathematical operations on numeric values.

**Operators**

+ (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus)

**Example**

```
public class ArithmeticDemo {
   public static void main(String[] args) {
      int a = 15, b = 4;

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

**Output**

```
a + b = 19
a - b = 11
a * b = 60
a / b = 3
a % b = 3
```

### Relational operators:

**Definition:**

Relational operators are used to compare two values.
They always return boolean values (true or false).

**Operators**

== (Equal to), != (Not equal to), > (Greater than), < (Less than), >= (Greater than or equal to), <= (Less than or equal to)

**Example**

```
public class RelationalDemo {
   public static void main(String[] args) {
      int x = 10, y = 20;

      System.out.println("x == y : " + (x == y)); // Equal
      System.out.println("x != y : " + (x != y)); // Not equal
      System.out.println("x > y  : " + (x > y));  // Greater
      System.out.println("x < y  : " + (x < y));  // Less
      System.out.println("x >= y : " + (x >= y)); // Greater or equal
      System.out.println("x <= y : " + (x <= y)); // Less or equal
   }
}
```

**Output**

```
x == y : false
x != y : true
x > y  : false
x < y  : true
x >= y : false
x <= y : true
```

**4.2 Logical operators and Assignment operators**
**Logical operators :**

Logical operators are used to perform logical operations on boolean values.
They return either true or false.
**Types of Logical Operators**

&& → Logical AND

|| → Logical OR

! → Logical NOT

**Example**

```
public class LogicalDemo {
   public static void main(String[] args) {
      int x = 10, y = 20;

      // Logical AND
      System.out.println((x > 5) && (y > 15)); // true

      // Logical OR
      System.out.println((x < 5) || (y > 15)); // true

      // Logical NOT
      System.out.println(!(x == y)); // true
   }
}
```

**Output**

```
true
true
true
```

**Assignment operators:**

Assignment operators are used to assign values to variables.
They can also be combined with arithmetic operators.

**Types**

= → Assigns value

+= → Add and assign

-= → Subtract and assign

*= → Multiply and assign

/= → Divide and assign

%= → Modulus and assign

**Example**

```
public class AssignmentDemo {
   public static void main(String[] args) {
      int a = 10;
      System.out.println("Initial a = " + a);

      a += 5; // a = a + 5
      System.out.println("a += 5 → " + a);

      a -= 3; // a = a - 3
      System.out.println("a -= 3 → " + a);

      a *= 2; // a = a * 2
      System.out.println("a *= 2 → " + a);

      a /= 4; // a = a / 4
      System.out.println("a /= 4 → " + a);

      a %= 3; // a = a % 3
      System.out.println("a %= 3 → " + a);
   }
}
```

**Output**

```
Initial a = 10
a += 5 → 15
a -= 3 → 12
a *= 2 → 24
a /= 4 → 6
a %= 3 → 0
```

**4.3 Increment & Decrement operators:**

Increment (++) and Decrement (--) operators are unary operators (work on a single operand).

They increase or decrease the value of a variable by 1.

**1. Increment Operator (++)**

Pre-increment (++x) → Increases the value first, then uses it.

Post-increment (x++) → Uses the value first, then increases it.

**2. Decrement Operator (--)**

Pre-decrement (--x) → Decreases the value first, then uses it.

Post-decrement (x--) → Uses the value first, then decreases it.

**Example**

```
public class IncDecDemo {
   public static void main(String[] args) {
      int a = 5, b = 5;

      // Pre-increment
      System.out.println("Pre-increment: " + (++a)); // a = 6, prints 6

      // Post-increment
      System.out.println("Post-increment: " + (b++)); // prints 5, then b = 6
      System.out.println("Value of b after post-increment: " + b);

      int x = 10, y = 10;

      // Pre-decrement
      System.out.println("Pre-decrement: " + (--x)); // x = 9, prints 9

      // Post-decrement
      System.out.println("Post-decrement: " + (y--)); // prints 10, then y = 9
```

```
        System.out.println("Value of y after post-decrement: " + y);
    }
}
```

**Output**

Pre-increment: 6
Post-increment: 5
Value of b after post-increment: 6
Pre-decrement: 9
Post-decrement: 10
Value of y after post-decrement: 9

### 4.4 Conditional operators and Bitwise operators

**Conditional operators:**

The conditional operator (?:) is the only ternary operator in Java (works on three operands).

It acts as a shorthand for an if-else statement.

**Syntax**

variable = (condition) ? value_if_true : value_if_false;

**Example**

```
public class ConditionalDemo {
    public static void main(String[] args) {
        int age = 20;

        String result = (age >= 18) ? "Eligible to Vote" : "Not Eligible";
        System.out.println("Result: " + result);

        int x = 15, y = 25;
        int max = (x > y) ? x : y; // find maximum
        System.out.println("Maximum is: " + max);
    }
}
```

**Output**

Result: Eligible to Vote

Maximum is: 25

**Bitwise operators**

Bitwise operators work at the bit level (0s and 1s).

They are mainly used for low-level programming, masks, encryption, and optimization.

**Operators**

& → Bitwise AND

| → Bitwise OR

^ → Bitwise XOR

~ → Bitwise NOT (One's complement)

<< → Left shift

>> → Right shift (signed)

>>> → Unsigned right shift

**Example**

```
public class BitwiseDemo {
   public static void main(String[] args) {
      int a = 5;   // 0101 in binary
      int b = 3;   // 0011 in binary
      System.out.println("a & b = " + (a & b));  // AND → 0001 = 1
      System.out.println("a | b = " + (a | b));  // OR  → 0111 = 7
      System.out.println("a ^ b = " + (a ^ b));  // XOR → 0110 = 6
      System.out.println("~a = " + (~a));        // NOT → -(a+1) = -6

      System.out.println("a << 1 = " + (a << 1)); // Left shift → 1010 = 10
      System.out.println("a >> 1 = " + (a >> 1)); // Right shift → 0010 = 2
```

```
        System.out.println("a >>> 1 = " + (a >>> 1)); // Unsigned right shift
    }
}
```

**Output**

```
a & b = 1
a | b = 7
a ^ b = 6
~a = -6
a << 1 = 10
a >> 1 = 2
a >>> 1 = 2
```

**4.5 Arithmetic Expressions**

It is a combination of operands (constants, variables) and operators (+, -, *, /, %) that Java evaluates to produce a value.

**Example**

```
class ArithmeticExample
{
    public static void main(String[] args)
    {
        int a = 10, b = 3;
        int sum = a + b;
        int diff = a - b;
        int prod = a * b;
        int div = a / b;   // integer division
        int mod = a % b;   // remainder

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + diff);
        System.out.println("Product: " + prod);
        System.out.println("Division: " + div);
        System.out.println("Remainder: " + mod);
    }
}
```

**Output:**

Sum: 13
Difference: 7
Product: 30
Division: 3
Remainder: 1

**4.6 Evaluation of Expressions**

Java evaluates expressions based on precedence and associativity rules.
The Parentheses () can be used to change the natural order.

**Example**

```
class EvaluationExample
{
   public static void main(String[] args)
   {
      int result = 10 + 5 * 2;//Multiplication happens first
      int result2 = (10 + 5) * 2;//Parentheses change the order

      System.out.println("10 + 5 * 2 = " + result);
      System.out.println("(10 + 5) * 2 = " + result2);
   }
}
```

**Output:**

10 + 5 * 2 = 20
(10 + 5) * 2 = 30

**4.7 Type Conversions in Expressions**

Java automatically performs type conversion when operands are of different data types.

- Implicit conversion (type promotion): smaller type → larger type
- Explicit conversion (casting): programmer forces conversion

**Example**

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

```
        int a = 5;
        double b = 2.5;

        double result = a + b;   // int promoted to double
        int result2 = (int)(a + b);  // explicit casting to int

        System.out.println("Result (double): " + result);
        System.out.println("Result (int): " + result2);
    }
}
```

**Output:**

Result (double): 7.5
Result (int): 7

### 4.8 Operator Precedence and Associativity

When multiple operators appear in an expression:

- Precedence decides which is evaluated first.
- Associativity decides the order when operators have the same precedence.

Precedence Order (highest to lowest):

1. () : Parentheses
2. *, /, % : Multiplication, Division, Modulus
3. +, - : Addition, Subtraction
4. Assignment operators (=, +=, etc.)

Associativity:

- Most operators (+, -, *, /) → Left to Right
- Assignment operators (=, +=, etc.) → Right to Left

**Example**

```
class PrecedenceExample {
    public static void main(String[] args) {
        int x = 10, y = 20, z = 30;

        int result = x + y * z;   // * first, then +
        int result2 = (x + y) * z; // parentheses change order
```

```
        int result3 = x + y - z;   // left to right associativity

        System.out.println("x + y * z = " + result);
        System.out.println("(x + y) * z = " + result2);
        System.out.println("x + y - z = " + result3);
    }
}
```

**Output:**

```
x + y * z = 610
(x + y) * z = 900
x + y - z = 0
```

# Chapter – 5 CONTROL STATEMENTS

- A program executes from top to bottom except when we use control statements.
- we can control the order of execution of the program, based on logic and values.

In Java, **control statements** can be divided into **three categories**:

- Decision Making Statements
- Branching Statements
- Looping Statements

## 5.1 Decision Making

Decision making statements are used to control the flow of execution based on conditions.
They allow the program to take different paths depending on whether a condition is true or false.

### 1. if Statement

Executes a block only if the condition is true.

**Syntax**:

```
if (condition) {
   // code to execute if condition is true
}
```

**Example:**

```
public class IfDemo {
   public static void main(String[] args) {
      int age = 20;
      if (age >= 18) {
         System.out.println("Eligible to vote");
      }
   }
}
```

**Output:**

Eligible to vote

**2. if-else Statement**

Executes one block if condition is true, otherwise another block.

**Example:**

```
public class IfElseDemo {
    public static void main(String[] args) {
        int num = 10;
        if (num % 2 == 0) {
            System.out.println("Even number");
        } else {
            System.out.println("Odd number");
        }
    }
}
```

**Output:**

Even number

**3. if-else-if Ladder**

Used when multiple conditions need to be tested.

**Example:**

```
public class IfElseIfDemo {
    public static void main(String[] args) {
        int marks = 82;
        if (marks >= 90) {
            System.out.println("Grade A");
        } else if (marks >= 75) {
            System.out.println("Grade B");
        } else if (marks >= 50) {
            System.out.println("Grade C");
        } else {
            System.out.println("Fail");
```

```
        }
    }
}
```

**Output:**

Grade B

**4. Nested if Statement**

An if inside another if.

**Example:**

```
public class NestedIfDemo {
    public static void main(String[] args) {
        int age = 25;
        boolean hasID = true;

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

**Output:**

Eligible to vote

**5. Switch Statement**

Used when we have multiple possible values for a single variable.
Better than long if-else-if ladders.

**Example:**

```
public class SwitchDemo {
    public static void main(String[] args) {
        int day = 3;
        switch (day) {
```

```
            case 1: System.out.println("Monday"); break;
            case 2: System.out.println("Tuesday"); break;
            case 3: System.out.println("Wednesday"); break;
            case 4: System.out.println("Thursday"); break;
            case 5: System.out.println("Friday"); break;
            case 6: System.out.println("Saturday"); break;
            case 7: System.out.println("Sunday"); break;
            default: System.out.println("Invalid day");
        }
    }
}
```

**Output:**

Wednesday

**5.2 Looping statements**

Looping statements allow a set of instructions to be executed repeatedly until a condition is satisfied.They are also called iteration statements.

**Types of Loops:**

**1. for Loop**

Best when number of iterations is known.

**Syntax:**

```
for(initialization; condition; update)
{
   // code block
}
```

**Example:**

```
public class ForLoopDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Count: " + i);
        }
    }
}
```

**Output:**

Count: 1
Count: 2
Count: 3
Count: 4
Count: 5

**2. While Loop**

Condition checked before execution.

May run zero times if condition is false initially.

**Example:**

```
public class WhileLoopDemo {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 5) {
            System.out.println("i = " + i);
            i++;
        }
    }
}
```

**Output:**

i = 1
i = 2
i = 3
i = 4
i = 5

**3. do-while Loop**

Condition checked after execution and runs at least once.

**Example:**

```
public class DoWhileDemo
{
   public static void main(String[] args) {
      int i = 1;
      do {
         System.out.println("Value: " + i);
         i++;
      } while (i <= 5);
   }
}
```

**Output:**

```
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
```

**4. for-each Loop (Enhanced for)**

Special loop for arrays/collections. Automatically iterates through elements.

**Example:**

```
public class ForEachDemo {
   public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40};
      for (int n : numbers) {
         System.out.println(n);
      }
   }
}
```

**Output:**

```
10
20
30
40
```

### 5.3 Jump in loops

Jump statements are used to change the normal flow of execution inside loops. They allow us to exit, skip, or continue loops without finishing all iterations.

#### 1. break Statement

- Terminates the current loop immediately.
- Control jumps outside the loop.
- Can also be used in switch statements.

**Example:**

```
public class BreakDemo {
   public static void main(String[] args) {
      for (int i = 1; i <= 5; i++) {
         if (i == 3) {
            break; // exit loop when i is 3
         }
         System.out.println("i = " + i);
      }
   }
}
```

**Output:**

```
i = 1
i = 2
```

**2. Continue Statement**

Skips the current iteration and moves to the next iteration of the loop.

**Example:**

```
public class ContinueDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                continue; // skip printing 3
            }
            System.out.println("i = " + i);
        }
    }
}
```

**Output:**

i = 1
i = 2
i = 4
i = 5

**3. return Statement**

Used to exit a method immediately, optionally returning a value.
Can also be used to exit from a loop inside a method.

**Example**:

```
public class ReturnDemo
{
    public static void checkNumber(int n)
    {
        for (int i = 1; i <= 5; i++)
         {
            if (i == n) {
                System.out.println("Number found: " + i);
```

```
            return; // exit method immediately
                    }
          }
        System.out.println("Number not found");
      }
    public static void main(String[] args)
    {
       checkNumber(3);
    }
}
```

**Output:**

Number found: 3

**5.4 Labelled loops**

- A labelled loop is a loop that is given a name (label).
- It is used with break or continue to control nested loops.
- Labels allow us to break out of or continue a specific outer loop.

**Syntax:**

```
labelName:
for(initialization; condition; update)
{
   // loop body
}
```

**Using break with Labelled Loop**

**Example:**

```
public class LabelledBreakDemo {
   public static void main(String[] args) {
      outerLoop:  // label
```

```
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (i == 2 && j == 2)
                 {
                    break outerLoop; // breaks outer loop
                 }
                System.out.println("i = " + i + ", j = " + j);
            }
        }
    }
}
```

**Output:**

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 2, j = 1
```

**Explanation:**

When i=2 and j=2, the break outerLoop terminates the outer loop, not just the inner loop.

**Using continue with Labelled Loop**

**Example:**

```
public class LabelledContinueDemo {
    public static void main(String[] args) {
        outerLoop:
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (i == 2 && j == 2) {
                    continue outerLoop; // skip remaining inner loop & continue outer
```

```
            }
            System.out.println("i = " + i + ", j = " + j);
        }
    }
  }
}
```

**Output:**

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 2, j = 1
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```

**Explanation:**

When i=2 and j=2, continue outerLoop skips the rest of inner loop for i=2 and continues with i=3.

# Chapter – 6 CLASSES, OBJECTS AND METHODS

- A class in Java is a blueprint or template that defines the properties (fields/variables) and behaviors (methods/functions) of objects.
- It is a user-defined data type.
- A class itself does not occupy memory; memory is allocated only when an object is created from it.

## 6.1 Defining a Class

**Structure of a Class**

```
class ClassName
 {
   // Fields (variables)
   dataType variableName;

   // Methods (functions)
   returnType methodName(parameters)
    { // code block
      return value; // optional
   }
}
```

**Example of a Class**

```
class Person
{
   // Fields (attributes)
   String name;
   int age;

   void display()   // Method (behavior)
  {
      System.out.println("Name: " + name + ", Age: " + age);
   }
}
```

**Note:**

- name and age → fields/attributes storing object data.

- display() → method/behavior of the class.

**Creating  Objects from a Class**

```
public class Main
{
   public static void main(String[] args)
 {
       // Creating two objects of class Person
       Person p1 = new Person();
       Person p2 = new Person();

       // Assign values to fields
       p1.name = "Alice";
       p1.age = 25;

       p2.name = "Bob";
       p2.age = 30;

       // Call method
       p1.display();
       p2.display();
   }
}
```

**Output:**

Name: Alice, Age: 25
Name: Bob, Age: 30

Below is a **simple, clear, exam-friendly explanation** for **6.3 Methods** and **6.3.1 Built-In Numerical Methods** (commonly part of Java basics / Java Math API units).

## 6.2 Methods

A **method** in Java is a block of code that performs a specific task.
Methods help in **code reusability, modularity, readability**, and **easy debugging**.

**Syntax of a Method**

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

**Types of Methods**

1. **Built-in methods**
   Already defined in Java libraries (e.g., Math.sqrt(), System.out.println())
2. **User-defined methods**
   Created by the programmer.

### 6.2.1 Built-In Numerical Methods

Java provides a number of **predefined mathematical functions** inside the **Math** class (java.lang.Math).These methods help perform **numerical operations** easily.

Common Built-In Numerical Methods in Java

**1. Math.sqrt(x)**

Returns the **square root** of x.

```
System.out.println(Math.sqrt(25));   // 5.0
```

**2. Math.pow(a, b)**

Returns **a raised to the power b**.

```
System.out.println(Math.pow(2, 3));  // 8.0
```

**3. Math.abs(x)**

Returns the **absolute value** (positive value).

System.out.println(Math.abs(-10)); // 10

**4. Math.max(a, b)**

Returns the **greater** of two numbers.

System.out.println(Math.max(10, 20)); // 20

**5. Math.min(a, b)**

Returns the **smaller** of two numbers.

System.out.println(Math.min(10, 20)); // 10

**6. Math.round(x)**

Rounds the number to the **nearest integer**.

System.out.println(Math.round(4.6f)); // 5

**7. Math.ceil(x)**

Returns the **smallest integer greater than or equal to x** (rounds UP).

System.out.println(Math.ceil(4.1)); // 5.0

**8. Math.floor(x)**

Returns the **largest integer less than or equal to x** (rounds DOWN).

System.out.println(Math.floor(4.9)); // 4.0

**9. Math.random()**

Returns a **random number** between **0.0 and 1.0**.

System.out.println(Math.random());

### 6.3 Constructors

- A constructor is a special method in a class that is called automatically when an object is created.
- It is used to initialize objects.
- A constructor has the same name as the class and no return type (not even void).

**Key Points**

- Constructor name = class name.
- No return type (not even void).
- Called automatically when an object is created.
- Can have parameters (parameterized constructor) or no parameters (default constructor).

**Types of Constructors**

**1. Default Constructor**

No parameters.
Initializes objects with default values.

**Example:**

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

   // Default constructor
   Person()
  {
      name = "Unknown";
      age = 0;
   }

   void display()
   {
      System.out.println("Name: " + name + ", Age: " + age);
```

```
    }
}

public class Main
{
    public static void main(String[] args)
   {
        Person p1 = new Person(); // calls default constructor
        p1.display();
    }
}
```

**Output:**
Name: Unknown, Age: 0

**2. Parameterized Constructor**

Takes arguments to initialize object with specific values.

**Example**:

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

    // Parameterized constructor
    Person(String n, int a)
   {
        name = n;
        age = a;
    }
    void display()
    {
        System.out.println("Name: " + name + ", Age: " + age);
    }
```

```
}

public class Main
{
   public static void main(String[] args)
{
       Person p1 = new Person("Alice", 25); // calls parameterized constructor
       Person p2 = new Person("Bob", 30);

       p1.display();
       p2.display();
   }
}
```

**Output:**

```
Name: Alice, Age: 25
Name: Bob, Age: 30
```

### 6.4 Method Overloading

- Method Overloading is a feature in Java that allows a class to have more than one method with the same name but different parameters.
- It is a way to achieve compile-time (static) polymorphism.

**Key Points**

- Methods must have same name.
- Must have different parameter list (number or type of parameters).
- Can have different return types, but return type alone cannot be used to distinguish methods.
- Overloading improves code readability.

**Example 1: Overloading by Number of Parameters**

```
class Calculator
```

```
{
   // Method to add two numbers
   int add(int a, int b)
   {
      return a + b;
   }

   // Method to add three numbers
   int add(int a, int b, int c)
   {
      return a + b + c;
   }
}

public class Main
{
   public static void main(String[] args)
  {
      Calculator calc = new Calculator();
      System.out.println("Sum of 2 numbers: " + calc.add(10, 20));
      System.out.println("Sum of 3 numbers: " + calc.add(10, 20, 30));
   }
}
```

**Output:**

```
Sum of 2 numbers: 30
Sum of 3 numbers: 60
```

**Example 2: Overloading by Type of Parameters**

```
class Calculator
{
   // Method to add integers
   int add(int a, int b)
  {
```

```
        return a + b;
    }

    // Method to add doubles
    double add(double a, double b)
    {
        return a + b;
    }
}

public class Main
{
    public static void main(String[] args)
  {
        Calculator calc = new Calculator();
        System.out.println("Sum of integers: " + calc.add(10, 20));
        System.out.println("Sum of doubles: " + calc.add(10.5, 20.5));
    }
}
```

**Output:**
Sum of integers: 30
Sum of doubles: 31.0

**Example 3: Overloading by Number & Type of Parameters**

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

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

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

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));
        System.out.println(calc.add(5.5, 10.5));
        System.out.println(calc.add(1, 2, 3));
    }
}
```

**Output:**

```
15
16.0
6
```

### 6.5 Constructor Overloading

A class can have more than one constructor with different parameters.

**Example:**

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

    Person()
  {
        name = "Unknown";
```

```
        age = 0;
    }

    Person(String n)
    {
        name = n;
        age = 0;
    }

    Person(String n, int a)
    {
        name = n;
        age = a;
    }

    void display()
    {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main
{
    public static void main(String[] args)
{
        Person p1 = new Person();
        Person p2 = new Person("Alice");
        Person p3 = new Person("Bob", 30);

        p1.display();
        p2.display();
        p3.display();
    }
}
```

**Output:**

Name: Unknown, Age: 0
Name: Alice, Age: 0
Name: Bob, Age: 30

### 6.6 Inheritance.

Inheritance is a mechanism in Java by which a new class (subclass/child class) acquires the properties and behaviors (fields and methods) of an existing class (superclass/parent class).

- It allows code reusability and hierarchical classification.
- The subclass can use, extend, or modify the features of the superclass.

**Keyword used:** extends
**Example in Words**

- Superclass: Animal → has methods like eat()
- Subclass: Dog → inherits eat() from Animal and can have its own method bark()

This way, the subclass **does not need to rewrite** the methods of the superclass.

**Types of Inheritance in Java**

.

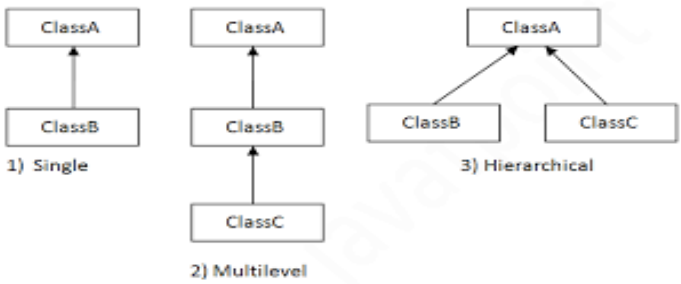



Fig: Types of Inheritances

- Single Inheritance → One subclass inherits from one superclass.
- Multilevel Inheritance → Class A → Class B → Class C.
- Hierarchical Inheritance → multiple subclasses inherit from one superclass.

- Multiple Inheritance (through interfaces) → Java does not support multiple inheritance with classes directly to avoid ambiguity, but can be achieved using interfaces

## 6.7 Overriding Methods

- Method Overriding occurs when a subclass (child class) provides its own implementation of a method that is already defined in its superclass (parent class).
- It is a way to achieve runtime (dynamic) polymorphism.

**The overridden method must have:**

- Same method name
- Same parameter list
- Same return type (or covariant return type)
- Access modifier in the subclass cannot be more restrictive than in the superclass.

**Key Points**

- Occurs between superclass and subclass.
- Used to provide specific implementation in subclass.
- Static methods cannot be overridden (can be hidden instead).

**Example 1: Simple Method Overriding**

```
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 Main
{
   public static void main(String[] args)
  {
      Animal a = new Animal();
      a.sound(); // calls superclass method

      Dog d = new Dog();
      d.sound(); // calls subclass method
   }
}
```

**Output:**

Animal makes a sound
Dog barks

**Example 2: Overriding with Polymorphism**

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

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

public class Main
{
   public static void main(String[] args) {
      Animal a;

      a = new Animal();
      a.sound(); // Animal method
```

```
        a = new Cat();
         a.sound(); // Cat method (runtime polymorphism)
    }
}
```

**Output:**
Animal makes a sound
Cat meows

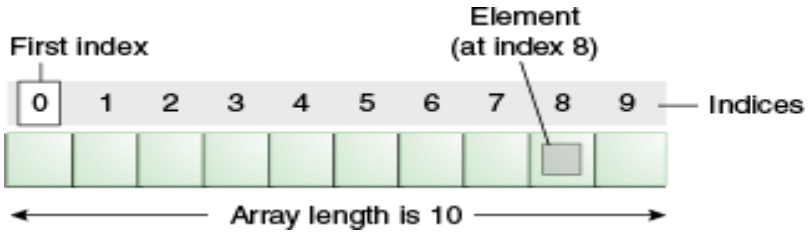
**Note:**

The reference type is Animal, but the actual object determines which method is called at runtime.

# Chapter – 7 ARRAYS, STRINGS AND VECTORS

## 7.1 Creating 1D arrays and 2D arrays

- Array is a collection of similar type of elements that have contiguous memory location.
- Array is an object that contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.
- Array in java is index based, first element of the array is stored at 0 index.




There are two types of array.

- Single Dimensional Array

  A **Single Dimensional Array** in Java is a **collection of elements of the same data type** stored in **contiguous memory locations**.

  It is like a **list of items** (e.g., marks of students, names, numbers) that can be accessed using an **index**.

- Multidimensional Array

  A **Multidimensional Array** in Java is an **array of arrays** — that means each element of the main array can itself hold another array.

The most common type is the **Two-Dimensional (2D) Array**, which can be visualized as **rows and columns** (like a table or matrix).

**Single Dimensional Array (1D array)**

**Syntax to Declare**

dataType[] arrvariable; (or) dataType arrvariable[];

**Syntax to Instantiate array**

arrayRef Var=**new** datatype[size];

**Example:**

In the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

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

        int a[]=new int[5];//declaration and instantiation

        a[0]=10;//initialization
        a[1]=20;
        a[2]=70;
        a[3]=40;
        a[4]=50;

        //printing array
        for(int i=0;i<a.length;i++)//length is
        the property of array
        System.out.println(a[i]);
    }
}
```

**Output:**

```
10
20
 70
 40
 50
```

**Declare, Instantiate and Initialize the java array together by:**

```
 int a[]={33,3,4,5};  //declaration, instantiation and initialization.
class Testarray1
{
          public static void main(String args[])
        {
          int a[]={33,3,4,5};//declaration, instantiation and initialization
          //printing array
        for(int i=0;i<a.length;i++)//length is
        the property of array
        System.out.println(a[i]);
        }
}
```

**Output:**

```
33
3
4
5
```

**Two Dimensional Array (2D array)**

Declaration:

```
int[][] arr=new int[3][3]; //3 rows and 3 columns

 arr[0][0]=1;

 arr[0][1]=2;

 arr[0][2]=3;

 arr[1][0]=4;

 arr[1][1]=5;
```

```
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

**Example:**

```
//To declare, instantiate, initialize and print the 2Dimensional array.
class Testarray3
{
public static void main(String args[])
 {
        //declaring and initializing 2D array
         int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
         //printing 2D array
        for(int i=0;i<3;i++)
        {
          for(int j=0;j<3;j++)
                {
         System.out.print(arr[i]
         [j]+" "); }
         System.out.println();
        }
  }
 }
```

**Output:**

```
1 2 3
2 4 5
4 4 5
```

**7.2 Strings**

Definition:

A String in Java is a sequence of characters enclosed in double quotes (" ").

Strings are objects of the String class in java.lang package and are immutable, meaning once a string object is created, its content cannot be changed.

**Example Code**:

```
public class StringExample {
   public static void main(String[] args)
{
       // Creating a String
       String message = "Java Programming";
      // Performing various operations on String
       System.out.println("Original String: " + message);
       System.out.println("Length of String: " + message.length());
       System.out.println("Uppercase: " + message.toUpperCase());
       System.out.println("Lowercase: " + message.toLowerCase());
       System.out.println("Character at index 5: " + message.charAt(5));
       System.out.println("Substring (from 5 to 16): " + message.substring(5, 16));
       System.out.println("Contains 'Java'? " + message.contains("Java"));
       System.out.println("Replaced 'Java' with 'Python': " + message.replace("Java",
"Python"));
   }
}
```

**Output:**

```
Original String: Java Programming
```

Length of String: 16

Uppercase: JAVA PROGRAMMING

Lowercase: java programming

Character at index 5: P

Substring (from 5 to 16): Programming

Contains 'Java'? true

Replaced 'Java' with 'Python': Python Programming

**Explanation:**

| Method | Description | Example Output |
|---|---|---|
| `length()` | Returns the number of characters in the string | 16 |
| `toUpperCase()` | Converts all characters to uppercase | JAVA PROGRAMMING |
| `toLowerCase()` | Converts all characters to lowercase | java programming |
| `charAt(index)` | Returns the character at a specific position | P |
| `substring(start,end)` | Extracts part of the string from start to end-1 | Programming |
| `contains()` | Checks if a particular substring exists | true |
| `replace()` | Replaces one substring with another | Python Programming |

**Key Points:**

Strings are immutable — operations like replace() or toUpperCase() create a new String object, not modify the original.

**Strings can be created using:**

String s1 = "Hello"; // Using string literal

String s2 = new String("Hi"); // Using 'new' keyword

**Comparison of Strings, StringBuilder, and StringBuffer in Java:**

**1.String**

Definition:

- A String in Java is an immutable sequence of characters.
- Once created, it cannot be changed — any modification creates a new String object.

**Example Code:**

```
public class StringExample {
   public static void main(String[] args) {
      String str = "Hello";
      str.concat(" World"); // trying to add another word
      System.out.println("String: " + str); // original string remains same

      String newStr = str.concat(" World"); // new object created
      System.out.println("New String: " + newStr);
   }
}
```

**Output:**

String: Hello

New String: Hello World

**Explanation**:

- Strings are immutable — operations like concat() return a new object.
- Modifying strings repeatedly creates multiple objects, which is less memory efficient.

**2. StringBuilder**

Definition:

- StringBuilder is a mutable class - meaning the contents can be modified after creation.
- It is faster than StringBuffer.

**3.StringBuffer**

Definition:

- StringBuffer is similar to StringBuilder (mutable)
- It is suitable for multi-threaded environments.

**Example:**

```
public class StringComparison {
   public static void main(String[] args) {
      String s = "Hello";
      StringBuilder sb1 = new StringBuilder("Hello");
      StringBuffer sb2 = new StringBuffer("Hello");

      s.concat(" World");
      sb1.append(" World");
      sb2.append(" World");
      System.out.println("String: " + s);
      System.out.println("StringBuilder: " + sb1);
      System.out.println("StringBuffer: " + sb2);
   }
}
```

**Output:**
String: Hello
StringBuilder: Hello World
StringBuffer: Hello World

**Key Differences:**

| Feature | String | StringBuilder | StringBuffer |
|---|---|---|---|
| **Mutability** | Immutable | Mutable | Mutable |
| **Performance** | Slower (creates new object on modification) | Faster | Slightly slower |
| **Usage** | When few modifications needed | When frequent modifications needed | When frequent modifications needed |
| **Package** | java.lang.String | java.lang.StringBuilder | java.lang.StringBuffer |

**7.3 Vectors**
**Definition:**

A Vector in Java is a dynamic array that can grow or shrink in size automatically.

It is part of the java.util package and stores elements of the same type.

Unlike normal arrays, vectors can resize themselves and are synchronized — meaning they are thread-safe (can be safely used by multiple threads).

**Syntax:**
Vector<Type> vectorName = new Vector<Type>();

Example:

Vector<Integer> numbers = new Vector<Integer>();

◆ **Example** :

```
import java.util.Vector;

public class VectorExample {

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

```
// Create a Vector of Strings
Vector<String> fruits = new Vector<>();

// Adding elements
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Cherry");
fruits.add("Mango");

// Displaying elements
System.out.println("Fruits Vector: " + fruits);

// Accessing elements
System.out.println("First fruit: " + fruits.firstElement());
System.out.println("Last fruit: " + fruits.lastElement());
System.out.println("Element at index 2: " + fruits.get(2));

// Removing an element
fruits.remove("Banana");
System.out.println("After removing Banana: " + fruits);

// Inserting element at specific position
fruits.add(1, "Grapes");
System.out.println("After inserting Grapes at index 1: " + fruits);
// Checking size and capacity
```

```
        System.out.println("Size: " + fruits.size());
        System.out.println("Capacity: " + fruits.capacity());
    }
}
```

**Output:**

Fruits Vector: [Apple, Banana, Cherry, Mango]
First fruit: Apple
Last fruit: Mango
Element at index 2: Cherry
After removing Banana: [Apple, Cherry, Mango]
After inserting Grapes at index 1: [Apple, Grapes, Cherry, Mango]
Size: 4
Capacity: 10

**Explanation:**

| Method | Description | Example / Output |
|---|---|---|
| `add()` | Adds elements to the vector | Adds "Apple", "Banana", etc. |
| `remove()` | Removes specified element | Removes "Banana" |
| `add(index, element)` | Inserts element at specific position | Adds "Grapes" at index 1 |
| `get(index)` | Returns element at specified index | Returns "Cherry" |
| `firstElement()` | Returns the first element | "Apple" |
| `lastElement()` | Returns the last element | "Mango" |
| `size()` | Returns number of elements | 4 |
| `capacity()` | Returns current capacity (default 10) | 10 |

## 7.4 Wrapper Classes

**Definition:**

In Java, Wrapper Classes are object representations of primitive data types.

They “wrap” primitive values (like int, char, boolean) inside objects so they can be used in collections (like ArrayList) and other object-based operations.

All wrapper classes are found in the java.lang package and are immutable (once created, they cannot change).

🔷 List of Primitive Types and Their Wrapper Classes

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

**Example:**

```
public class WrapperExample {
    public static void main(String[] args) {
        // Primitive data types
        int a = 10;
        double b = 25.75;
        char c = 'A';
        boolean d = true;
```

```
        // Converting primitives to objects (Boxing)
        Integer objA = Integer.valueOf(a);
        Double objB = Double.valueOf(b);
        Character objC = Character.valueOf(c);
        Boolean objD = Boolean.valueOf(d);

        // Converting objects back to primitives (Unboxing)
        int x = objA.intValue();
        double y = objB.doubleValue();
        char z = objC.charValue();
        boolean w = objD.booleanValue();

        // Display values
        System.out.println("Boxed Integer: " + objA);
        System.out.println("Boxed Double: " + objB);
        System.out.println("Boxed Character: " + objC);
        System.out.println("Boxed Boolean: " + objD);
        System.out.println("Unboxed int: " + x);
        System.out.println("Unboxed double: " + y);
        System.out.println("Unboxed char: " + z);
        System.out.println("Unboxed boolean: " + w);
    }
}
```

**Output:**

Boxed Integer: 10

Boxed Double: 25.75

Boxed Character: A

Boxed Boolean: true

Unboxed int: 10

Unboxed double: 25.75

Unboxed char: A

Unboxed boolean: true

**Explanation:**

| Concept | Description | Example |
|---|---|---|
| Boxing | Converting a **primitive type → object** (wrapper) | `Integer obj = Integer.valueOf(a);` |
| Unboxing | Converting an **object → primitive type** | `int x = obj.intValue();` |
| Autoboxing | Java automatically converts primitive to object | `Integer obj = a;` |
| Auto-unboxing | Java automatically converts object to primitive | `int x = obj;` |

**Example with Autoboxing & Auto-unboxing:**

```
public class AutoBoxingExample {
    public static void main(String[] args) {
        int num = 50;

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

        // Auto-unboxing: object to primitive automatically
        int newNum = obj + 10;

        System.out.println("Autoboxed value: " + obj);
        System.out.println("Auto-unboxed value after addition: " + newNum);
    }
}
```

**Output:**

```
Autoboxed value: 50
Auto-unboxed value after addition: 60
```

**Why Use Wrapper Classes?**

- Collections Framework: Collections (like Vector,ArrayList,Stack) store only objects, not primitives.
- Utility Methods: Wrapper classes provide useful static methods (e.g., Integer.parseInt("100")).
- Type Conversion: Easy to convert between strings and numbers.
- Object-oriented Features: Wrappers allow primitives to behave like objects.

**7.5 Enumerated Types**
**Definition:**

- An enum (enumeration) in Java is a special data type that represents a group of named constants (fixed set of values).
- Enums are used when we have a predefined list of values that a variable can take — like days of the week, directions, months, etc.
- Enums were introduced in Java 5 as part of type-safe enumerations.
- They are more powerful than simple constants because they are objects of an enum class.

**Syntax:**

```
enum EnumName {
   CONSTANT1, CONSTANT2, CONSTANT3, ...
}
```

**Example:**

```
enum Direction { NORTH, SOUTH, EAST, WEST }
```

**Example Code:**

```
public class EnumExample {
   // Define an enum
   enum Day {
      SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY
   }
```

```
public static void main(String[] args) {
        Day today = Day.WEDNESDAY;

        System.out.println("Today is: " + today);

        // Loop through all enum values
        System.out.println("All days of the week:");
        for (Day d : Day.values()) {
            System.out.println(d);
        }

        // Using enum in a switch statement
        switch (today) {
            case MONDAY:
                System.out.println("Start of the week!");
                break;
            case FRIDAY:
                System.out.println("Almost weekend!");
                break;
            case SUNDAY:
                System.out.println("Weekend!");
                break;
            default:
                System.out.println("Midweek day.");
        }
    }
}
```

**Output:**

Today is: WEDNESDAY

All days of the week:

SUNDAY

MONDAY

TUESDAY

WEDNESDAY

THURSDAY

FRIDAY

SATURDAY

Midweek day.

**Explanation:**

- enum Day defines seven constants — one for each day of the week.
- Day today = Day.WEDNESDAY; assigns one constant to a variable.
- values() returns an array of all constants inside the enum.
- Enums can be used directly in switch statements, making code clean and readable.

# Chapter – 8 IMPLEMENTATION OF INHERITANCES

## 8.1 Defining and extending classes

### Single Inheritance

- Single Inheritance occurs when one subclass inherits the properties and methods of one superclass.
- The subclass can use the members of the superclass and also have its own members.

**Example:**

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

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

// Main class
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog(); // create object of subclass

        d.eat();  // inherited method from Animal
        d.bark(); // method of Dog
    }
}
```

**Output**

```
Animal eats food
Dog barks
```

**Explanation**

- Animal is the superclass.

- Dog is the subclass using extends keyword.
- The object d of subclass Dog can access:
    - eat() → inherited from Animal
    - bark() → defined in Dog

Single inheritance is the simplest form of inheritance and helps in **code reusability**

**Multilevel Inheritance:**

**Multilevel Inheritance** occurs when a class is derived from a subclass, forming a **chain of inheritance**.

- Example: **Grandparent → Parent → Child**
- The child class inherits members of the parent, and the parent inherits members of the grandparent.

**Example:**

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

class Mammal extends Animal
{
       void walk()
      {   System.out.println("Mammal walks");  }
}

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

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

```
        d.eat();  // from Animal
        d.walk(); // from Mammal
        d.bark(); // from Dog
    }
}
```

**Output:**
Animal eats
Mammal walks
Dog barks
**Explanation**

- Animal → Grandparent class
- Mammal → Parent class, inherits Animal
- Dog → Child class, inherits Mammal (and indirectly Animal)
- Object of Dog can access all methods from the chain of classes.Multilevel inheritance allows reusing code across multiple levels and forms a hierarchical chain.

**Hierarchical Inheritance**

Hierarchical Inheritance occurs when multiple subclasses inherit from a single superclass.

- One parent class → many child classes.
- Each child class can have its own methods in addition to the inherited methods.

**Example:**

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

 }
```

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

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

public class Main
{
   public static void main(String[] args)
{
       Dog d = new Dog();
       d.eat();
       d.bark();

       Cat c = new Cat();
       c.eat();
       c.meow();
   }
}
```

**Output:**

```
Animal eats
Dog barks
Animal eats
Cat meows
```

**Explanation**

- Animal → Superclass
- Dog and Cat → Subclasses
- Both Dog and Cat inherit the eat() method from Animal
- Each subclass has its own specific method (bark() for Dog, meow() for Cat)
- Hierarchical inheritance allows code reusability and lets multiple classes share common features from a single parent.

**Note:**

- Inheritance → one class acquires properties/methods of another.
- Keyword → extends

**Advantages:**

- Code reusability
- Easy maintenance
- Supports method overriding
- Types: Single, Multilevel, Hierarchical, Multiple (via interfaces)

### 8.2 Implementing Interfaces

An **interface** in Java is a collection of **abstract methods** (methods without body) and **constants**.
A class implements an interface to provide the method definitions.

Why Interfaces?

Interfaces are used for:

- Achieving **abstraction**
- Supporting **multiple inheritance**
- Defining **rules** that a class must follow

**Syntax**

```
interface InterfaceName
{
   // abstract methods
   void method1();
   void method2();
}

class ClassName implements InterfaceName
{
   public void method1() {
      // implementation
   }
   public void method2() {
      // implementation
   }
}
```

**Example :** Simple Interface Implementation

Program:

```
interface Animal {
   void sound();     // abstract method
   void eat();       // abstract method
}

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

   public void eat() {
      System.out.println("Dog eats bones");
   }
}

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

**Output:**
Dog barks
Dog eats bones

### 8.2.1 Multiple Interfaces

Java does **not support multiple inheritance** with classes, but **supports it using interfaces**.

Program:

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

interface Showable {
   void show();
}


class Demo implements Printable, Showable {
   public void print() {
      System.out.println("Printing data...");
   }

   public void show() {
      System.out.println("Showing data...");
   }
}

class Test {
   public static void main(String[] args) {
      Demo obj = new Demo();
      obj.print();
      obj.show();
   }
}
```

**Output:**
Printing data...
Showing data...

### 8.2.2 Interface Variables

Variables in interfaces are:

- **public**
- **static**
- **final**

**Exampe::**

```
interface Vehicle {
   int speed = 60;   // public static final by default
   void run();
}

class Car implements Vehicle {
   public void run() {
      System.out.println("Car is running at speed: " + speed);
   }
}

class Test {
   public static void main(String[] args) {
      Car c = new Car();
      c.run();
   }
}
```

**Output:**

```
Car is running at speed: 60
```

Note:

- A class **must override all abstract methods** of the interface.
- An interface can extend **another interface** using extends.
- A class can implement **multiple interfaces** using commas:
  class A implements X, Y

## 8.3 Multiple Inheritance and Polymorphism

### What is Multiple Inheritance?

Multiple inheritance means a class **inherits properties from more than one parent class**.
Example (general idea)
Class C inherits from Class A and Class B

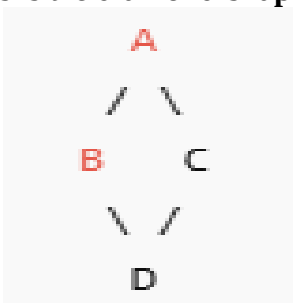
### Java does NOT support multiple inheritance through classes

Java avoids it to prevent **ambiguity** (known as the Diamond Problem).

✕ Not allowed:

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

What is Diamond  Problem ?
**Understanding the Structure**
This is the **diamond-shaped inheritance**:




- B and C both inherit from A
- D tries to inherit from both B and C
- If all have the same method, D becomes confused → **Diamond Problem**

  Since Java cannot decide **which parent's method to use** it does not allow Multiple inheritances

✓ **Java supports multiple inheritance using Interfaces**

**Example:**

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

interface B {
   void display();
}

class C implements A, B {
   public void show() { System.out.println("Show method"); }
   public void display() { System.out.println("Display method"); }
}
```

**Output:**

Show method
Display method

Why Java uses interfaces for multiple inheritance?

- No ambiguity
- Flexible
- Safe and clean design
- Allows a class to implement many interfaces

**What is Polymorphism?**

**Polymorphism** = many forms. One action behaves differently in different situationsTwo types of polymorphism in Java:

1. **Compile-time polymorphism (Method Overloading)**
2. **Runtime polymorphism (Method Overriding)**

- **Compile-Time Polymorphism (Method Overloading)**

Same method name, **different parameters**.

**Example:**

```
class Add {
   int sum(int a, int b) { return a + b; }
   int sum(int a, int b, int c) { return a + b + c; }
}
```

- **Runtime Polymorphism (Method Overriding)**

Child class gives its **own implementation** of a method from the parent class.

**Example:**

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

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

class Test {
   public static void main(String[] args) {
      Animal obj = new Dog();   // Runtime binding
      obj.sound();
   }
}
```

**Output:**

Dog barks

**Advantages of** Polymorphism:

- Code reusability
- Flexibility
- Clean class hierarchy
- Supports dynamic method binding

# Chapter – 9 PACKAGES

## 9.1 Defining Packages

A **package** in Java is a group of **related classes, interfaces, and sub-packages**. Packages help organize code and avoid naming conflicts.

Syntax to Define a Package
package packagename;

```
public class ClassName
{
   // class code
}
```

## 9.2 Advantages of Packages

1. **Modularity**
   - Organizes classes into separate folders/modules.
2. **Reusability**
   - Classes inside a package can be reused across applications.
3. **Name Conflict Avoidance**
   - Same class names can exist in different packages.
4. **Access Protection**
   - Package-level access using access modifiers.
5. **Maintainability**
   - Code becomes clean, manageable, and structured.

## 9.3 Built-In Packages

Java provides many ready-made packages under the **Java API**.

Common Built-In Packages

| Package | Description |
|---|---|
| java.lang | Basic classes (String, Math, System) |
| java.util | Utility classes (ArrayList, Scanner, Date) |
| java.io | Input/output classes (File, BufferedReader) |

| Package | Description |
|---|---|
| java.net | Networking classes (Socket, URL) |
| java.awt | GUI components |
| javax.swing | Advanced GUI components |
| java.sql | Database connectivity (JDBC) |

**Example Using Built-In Package**

```
import java.util.Scanner;

class Demo {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter a number: ");
      int n = sc.nextInt();
      System.out.println("You entered: " + n);
   }
}
```

### 9.4 User-Defined Packages

A user-defined package is created by programmers to group related classes.

Steps to Create a Package

1. Create a folder (e.g., mypack)
2. Inside it, create a Java file with package keyword
3. Compile using javac -d . filename.java
4. Use the package in another program

Example: User-Defined Package

**File: A.java**

```
package mypack;

public class A {
```

```
    public void display() {
        System.out.println("This is class A from package mypack");
    }
}
```

**9.5 Compiling and Running Java Packages**

Step 1: Compile the package
javac -d . A.java

-d . → Creates the package folder automatically.

Step 2: Use the package in another program(Importing Package)

**File: Test.java**

```
import mypack.A;

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

Step 3: Compile the user program
javac Test.java
Step 4: Run the program
java Test

**Output:**

This is class A from package mypack

**9.6 Accessing Packages**

We can access classes inside a package using:

1. import packagename.classname
import mypack.A;
2. import packagename. (all classes)*
import mypack.*;

3. Fully Qualified Name (FQN)

(No import required)

```
class Test {
    public static void main(String[] args) {
        mypack.A obj = new mypack.A();
        obj.display();
    }
}
```

How Access Modifiers Affect Packages:

| **Modifier** | **Access in Same Package** | **Access in Other Package** |
|---|---|---|
| **Public** | Yes | Yes |
| **Protected** | Yes | Yes (through inheritance) |
| **default** (no keyword) | Yes | No |
| **Private** | No | No |

# Chapter – 10 EXCEPTION HANDLING

## 10.1 Purpose of Exception Handling

An **exception** is an unwanted event that interrupts the normal flow of a program.

Purpose of Exception Handling

1. **To maintain normal program flow**
   Prevents program termination when an error occurs.
2. **To handle runtime errors gracefully**
   Example: division by zero, file not found, invalid input.
3. **To separate error-handling code from normal logic**
4. **To provide meaningful error messages**
5. **To avoid abnormal program termination**
6. **To ensure resource cleanup**
   (closing files, database connections)

## 10.2 The try and catch blocks

**try block**

Contains code that may cause exceptions.

**catch block**

Handles the exception thrown in the try block.

**Syntax**

```
try {
   // risky code
}
catch (ExceptionType e) {
   // handling code
}
```

**Example**

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

```
            int a = 10 / 0;
        }
        catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

**Output**

Cannot divide by zero!

### 10.3 The throw, throws and finally blocks

**throw keyword**

Used to **manually raise an exception**.

Example
```
throw new ArithmeticException("Invalid operation");
```

**throws keyword**

Used in method signature to indicate that a method **may throw** an exception.

Example
```
void myMethod() throws IOException {
    // code
}
```

**finally block**

Executes **whether exception occurs or not**.
Used for cleanup (close files, release resources).

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

```
        try {
            int a = 5 / 0;
        }
        catch (ArithmeticException e) {
            System.out.println("Error: " + e);
        }
        finally {
            System.out.println("Finally block executed.");
        }
    }
}
```

**Output**

```
Error: java.lang.ArithmeticException: / by zero
Finally block executed.
```

### 10.4 Nested try blocks

A try block inside another try block.

**Example**

```
class NestedTry {
    public static void main(String args[]) {
        try {
            int a = 10 / 2;

            try {
                int arr[] = {1, 2, 3};
                System.out.println(arr[5]); // Error
            }
            catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Inner catch: " + e);
            }
        }
        catch (Exception e) {
            System.out.println("Outer catch: " + e);
        }
    }
}
```

**Output**

Inner catch: java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds

### 10.5 Multiple catch statements

Used when the try block may generate **more than one type** of exception.

**Syntax**

```
try {
   // risky code
}
catch (ExceptionType1 e) { }
catch (ExceptionType2 e) { }
catch (Exception e) { }
```

**Example**

```
class MultipleCatchExample {
   public static void main(String[] args) {
      try {
         int a = 10 / 0;           // ArithmeticException
         int arr[] = new int[3];
         arr[5] = 20;              // ArrayIndexOutOfBoundsException
      }
      catch (ArithmeticException e) {
         System.out.println("Arithmetic Exception occurred!");
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Array Index Out Of Bounds Exception!");
      }
      catch (Exception e) {
         System.out.println("General Exception occurred!");
      }
   }
}
```

**Output**

Arithmetic Exception occurred!

- Only the **first exception** is caught.
- Once an exception occurs, the remaining try code is skipped.

### 10.6 Creating User Defined Exceptions

We can create our own exception by extending the **Exception** class.

Steps

1. Create a class extending Exception.
2. Throw an object of that class.
3. Catch it using catch block.

**Example**

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

class Test {
   static void checkAge(int age) throws InvalidAgeException {
      if (age < 18)
         throw new InvalidAgeException("Age must be 18 or above.");
      else
         System.out.println("Valid age!");
   }

   public static void main(String[] args) {
      try {
         checkAge(15);
      }
      catch (InvalidAgeException e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }
}
```

**Output**

Exception caught: Age must be 18 or above.