Getting Started - Java

COURSE INTRODUCTION, WHAT IS JAVA, INSTALLING JDK & IDE

Course Introduction

What you will learn:

- Core Java fundamentals
- Object-Oriented Programming
- Control flow, Collections, Exception Handling
- Mini Projects & Final Project

Why learn Java?

- Widely used, platform-independent
- Popular for backend, Android, enterprise apps

What is Java?

Java is a high-level, class-based, object-oriented programming language

Developed by Sun Microsystems in 1995 (now Oracle)

Key features:

- Write Once, Run Anywhere (WORA)
- Platform-independent bytecode
- Robust, secure, and portable

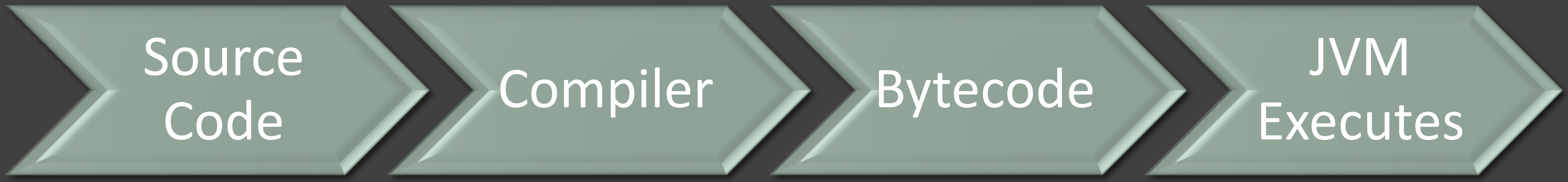
Java Platform Components

JDK (Java Development Kit): Tools to develop Java programs

JRE (Java Runtime Environment): Runs Java programs

JVM (Java Virtual Machine): Executes Java bytecode on any platform

Program flow:



Installing JDK

Step 1: Download JDK from Oracle's official site or OpenJDK

Step 2: Run the installer and follow instructions

Step 3: Set environment variables (JAVA_HOME & PATH)

Verify installation by running `java -version` in terminal/command prompt

Note: Step 3 is only required if path is not auto set, however, with the latest version of JDK installation this is auto taken care during installation.

Installing an IDE

What is an IDE?

- Integrated Development Environment to write, compile, and debug code

Popular Java IDEs:

- IntelliJ IDEA (Community Edition - free)
- Eclipse
- NetBeans

Installing Git

What is GIt?

- Git is a free, open-source version control system tool. It helps developers track changes in their code, collaborate with others, and manage project history efficiently.

Step 1: Download

- Visit git website
- Click “Download for Windows”

Step 2: Run installer

- Double click the .exe file
- Proceed with setup and accept default settings

Step 3: Verify Installation

- Open Git Bash or Command Prompt
- git --version

Git Platform

What are GitHub, GitLab, and Bitbucket?

- They are **cloud-based platforms** that host your **Git repositories online**, allowing teams to **collaborate**, **review code**, and **manage software projects**.

Feature	GitHub	GitLab	Bitbucket
🏢 Owner	Microsoft	GitLab Inc.	Atlassian
🔒 Private Repos	Free & unlimited	Free & unlimited	Free (up to 5 users per repo)
🤝 CI/CD Support	GitHub Actions (built-in)	Built-in CI/CD (powerful)	Bitbucket Pipelines
🧠 DevOps Tools	Issue tracking, Actions, Copilot	Full DevOps lifecycle tools	Integrated with Jira
👫 Collaboration	Popular for open-source	Used for enterprise and self-hosting	Often used with Jira users

Summary & Homework

Today's recap:

- Course overview
- Java basics and platform components
- Installed JDK and IDE

Homework:

- Install JDK and IDE on your system
- Explore IDE interface
- Write a simple "Hello World" Java program (preview for next day)

JVM, JRE, JDK
First Java Program

UNDERSTANDING PROGRAM EXECUTION FLOW

Recap

Course intro & Why Java

Java platform overview

Installed JDK & IDE

Homework review: JDK & IDE setup

JDK, JRE, and JVM

JDK (Java Development Kit)

- For developers
- includes compiler (javac)
- Debugger
- JRE

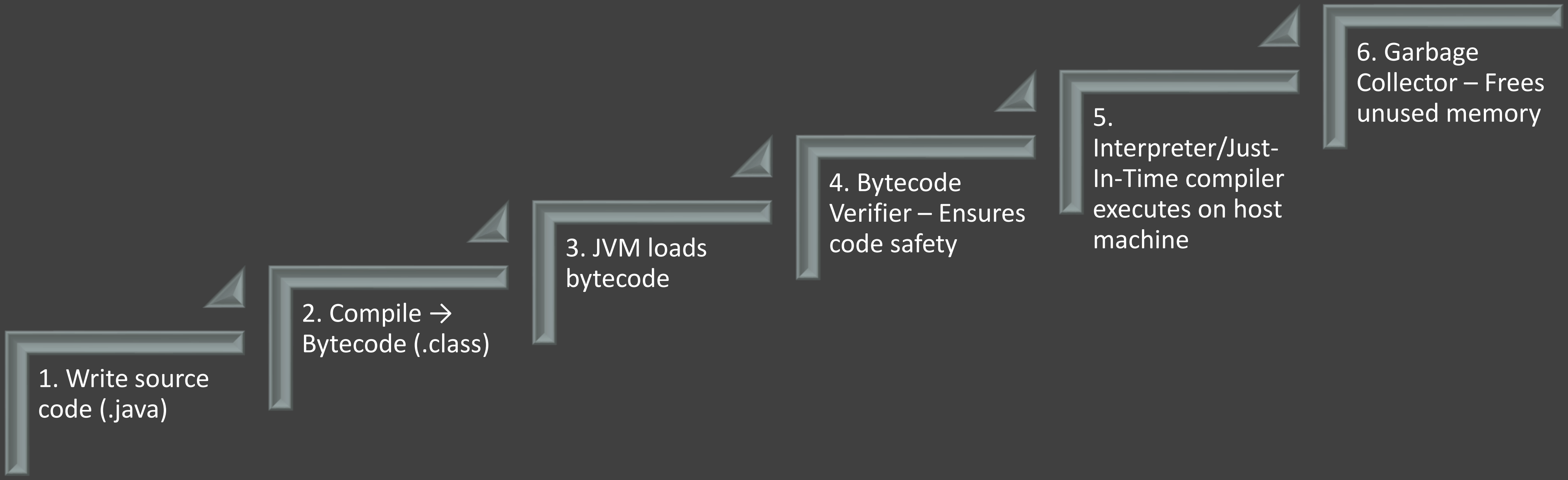
JRE (Java Runtime Environment)

- For running Java apps
- contains JVM + libraries

JVM (Java Virtual Machine)

- Executes bytecode
- platform-independent

How Java Code Executes



1. Write source code (.java)

2. Compile → Bytecode (.class)

3. JVM loads bytecode

4. Bytecode Verifier – Ensures code safety

5. Interpreter/Just-In-Time compiler executes on host machine

6. Garbage Collector – Frees unused memory

Flow: .java → .class → JRE → JVM → Execute

Writing Your First Program

class → defines a class

public static void main → entry point

System.out.println → prints text

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

Steps to Run First Program

Open IDE (e.g., IntelliJ IDEA)

Create new Java project

Write HelloWorld program

Compile & Run

- Compile: javac HelloWorld.java
- Run: java HelloWorld

Summary & Homework

Today's Recap:

- Difference between JDK, JRE, JVM
- Java program execution flow
- Wrote & ran first Java program

Homework:

- Modify HelloWorld to print your name
- Add 2 numbers and print the result

Java Syntax, Variables, and Data Types

BUILDING BLOCKS OF JAVA PROGRAMMING

Recap

JDK, JRE, JVM basics

Java program execution flow

Wrote & executed first program

Java Program Structure

```
public class Main {
    public static void main(String[] args) {
        String name = “Skill Wise”;
        System.out.println("Hello " + name);
    }
}
```

Java Syntax Rules

Case-sensitive language

Class name should start with a capital letter

File name = Class name (public class)

Each statement ends with ;

Code blocks enclosed in { }

Variables in Java

A variable is a container to store data in memory

Declaration: datatype variable_Name = value;

- Example:
  - int age = 25;
  - String name = "John";

Types of Variables

- Local Variables – Declared inside methods
- Instance Variables – Belong to an object
- Static Variables – Belong to a class (shared among all objects)

Java Data Types

- Primitive Data Types (8 total):
  - byte, short, int, long
  - float, double
  - char, boolean
- Non-Primitive (Reference) Types:
  - Strings, Arrays, Classes, Interfaces

Primitive Data Types Table

Type	Size	Example
byte	1 byte	127
int	4 bytes	12345
double	8 bytes	12.34
char	2 bytes	'A'
boolean	1 bit	true/false

Summary & Homework

- Recap:
  - Java syntax & program structure
  - Variables (local, instance, static)
  - Primitive & non-primitive data types
- Homework:
  - Create a program that stores your name, age, and marks
  - Print them in a formatted output

Operators in Java

ARITHMETIC, RELATIONAL & LOGICAL

Recap

- Java syntax & structure
- Variables & their types
- Primitive & non-primitive data types

What Are Operators?

- Special symbols used to perform operations on variables and values

  Example:

```
int a = 10, b = 5;
int sum = a + b; // + is an operator
```

Arithmetic Operators

Operator	Description	Example (a=10,b=5)	Result
+	Addition	a+b	15
-	Subtraction	a-b	5
*	Multiplication	a*b	50
/	Division	a/b	2
%	Modulus	a%b	0

Relational Operators

Operator	Description	Example (a=10,b=5)	Result
==	Equal to	a==b	false
!=	Not equal	a!=b	true
>	Greater than	a>b	true
<	Less than	a<b	false
>=	Greater than or equal	a>=b	true
<=	Less than or equal	a<=b	false

Logical Operators

Operator	Description	Example	Result
&&	Logical AND	(a>b)&&(b>0)	true
||	Logical OR	(a>b)||(b<0)	true
!	Logical NOT	!(a>b)	false

Operator Precedence

- Java evaluates operators in a specific order
- Highest precedence → Lowest precedence:
  - () Parentheses
  - * / % Multiplication/Division/Modulus
  - + - Addition/Subtraction
  - < > <= >= Relational
  - == != Equality
  - && AND
  - || OR
- Example: int result = 10 + 5 * 2; // 20, not 30

Practice Example

```
int a = 10, b = 5, c = 20;
System.out.println((a > b) && (c > a)); // true
System.out.println((a + b) * 2);      // 30
```

Summary & Homework

- Recap:
- - Arithmetic operators (+, -, *, /, %)
- - Relational operators (==, !=, >, <, >=, <=)
- - Logical operators (&&, ||, !)
- Homework:
- - Write a program to compare two numbers and print:
- • Sum, Difference, Multiplication, Division, Modulus
- • Which number is greater

Scanner Class: Taking User Input in Java

MAKING OUR PROGRAMS INTERACTIVE WITH USER INPUT

Why Take User Input?

Hardcoding values isn’t flexible.

Real-world apps need data from users.

Example: ATM asks for PIN, Quiz asks for answers

Java provides the Scanner class for this purpose.

Introducing the Scanner Class

Part of java.util package

Used to read input from Keyboard (System.in) or Files

Must import before use:

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

Creating a Scanner Object

```
import java.util.Scanner;

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

Common Methods in Scanner

Method	Purpose
nextInt()	Reads an integer
nextDouble()	Reads a decimal
nextLine()	Reads a full line
next()	Reads a single word
nextBoolean()	Reads true/false

Example – Calculator Input

```
import java.util.Scanner;

class Calculator {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter first number: ");
        int a = sc.nextInt();
        System.out.print("Enter second number: ");
        int b = sc.nextInt();
        System.out.println("Sum = " + (a+b));
    }
}
```

Handling Input Issues

- nextInt() leaves a newline → may cause issues with nextLine()

- Solution: use an extra sc.nextLine() to consume newline

```
int age = sc.nextInt();
sc.nextLine(); // consume leftover newline
String name = sc.nextLine();
```

Practice Exercise

Task: Write a program that asks the user:

- Name
- Age
- Favorite Hobby

Then print:

"Hello [Name], you are [Age] years old and you love [Hobby]!"

Recap

Scanner class allows interactive programs

Key methods: nextInt(), nextLine(), etc.

Always import from java.util

Handle newline issues carefully

Looking Ahead

Today: Taking input with Scanner

Tomorrow: Type Casting + Practice Exercises

Type Casting in Java

CONVERTING ONE DATA TYPE INTO ANOTHER

Why Type Casting?

- Variables store different data types (int, double, char, etc.)

- Sometimes we need to convert one type to another

- Example:

  5 / 2 → 2 (int division)

  (double)5 / 2 → 2.5

Two Types of Casting

1. Implicit Casting (Type Promotion)

   - Done automatically by Java

   - Converts smaller type → larger type

2. Explicit Casting (Type Conversion)

   - Done manually by the programmer

   - May cause data loss

Implicit Casting Example

```
public class Main {
    public static void main(String[] args) {
        int num = 10;
        double result = num;  // int → double
        System.out.println("Result: " + result);
    }
}
```

Explicit Casting Example

```
public class Main {
    public static void main(String[] args) {
        double num = 9.7;
        int result = (int) num;  // double → int
        System.out.println("Result: " + result);
    }
}
```

Types of Casting – Quick Compare

Casting Type	Who Performs	Direction	Risk of Data Loss	Example

Implicit Casting	Java (Auto)	Smaller → Larger	❌ No	int a = 5;
				double b = a;

Explicit Casting	Programmer	Larger → Smaller	✅ Yes	double d = 9.5;
				int i = (int) d;

Common Casting Scenarios

- int → double (safe)
- double → int (data loss)
- char ↔ int (ASCII values)

```
char ch = 'A';
int ascii = ch;  // 65
```

Casting with User Input

```
import java.util.Scanner;


public class CastingDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a decimal number: ");
        double d = sc.nextDouble();
        int i = (int) d;
        System.out.println("You entered " + d + ", after casting: " + i);
    }
}
```

Practice Exercise 1

Write a program that:

- Asks the user for their age (int)

- Stores it in a double variable

- Prints both values

Practice Exercise 2

Write a program that:

- Takes a decimal number as input

- Casts it to an integer

- Prints both the original decimal and the integer result

Recap

- Type Casting: Converting between data types

- Implicit Casting → Automatic, safe

- Explicit Casting → Manual, possible data loss

- Practice helps avoid surprises in calculations

Mini Project: Console Calculator

REVISION OF CONCEPTS

Project Goal

- Build a simple calculator that runs in the console
- Revision of:
  - Variables & Data Types
  - Operators
  - Scanner Class
  - Control Flow

Features Required

- Take two numbers from the user

- Allow user to choose an operation (+, -, *, /)

- Perform the operation

- Display the result

Step 1: Scanner Setup

```
import java.util.Scanner;


Scanner sc = new Scanner(System.in);

System.out.print("Enter first number: ");

double num1 = sc.nextDouble();

System.out.print("Enter second number: ");

double num2 = sc.nextDouble();
```

Step 2: Choose Operation

```
System.out.println("Choose operation: + - * /");

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

Step 3: Switch Case

```
double result;
switch(op) {
   case '+': result = num1 + num2; break;
   case '-': result = num1 - num2; break;
   case '*': result = num1 * num2; break;
   case '/': result = num1 / num2; break;
   default: System.out.println("Invalid Operation"); return;
}
```

Step 4: Display Result

```
System.out.println("Result = " + result);
```

Complete Example

```
import java.util.Scanner;

public class Calculator {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    System.out.print("Enter first number: ");

    double num1 = sc.nextDouble();

    System.out.print("Enter second number: ");

    double num2 = sc.nextDouble();

    System.out.print("Choose operation (+ - * /): ");

    char op = sc.next().charAt(0);

double result;
```

```
switch(op) {

      case '+': result = num1 + num2; break;

      case '-': result = num1 - num2; break;

      case '*': result = num1 * num2; break;

      case '/': result = num1 / num2; break;

      default: System.out.println("Invalid Operation"); return;

    }

   System.out.println("Result = " + result);

  }

}
```

Practice Enhancements

- Add modulus (%) operation

- Handle division by zero

- Allow user to perform multiple calculations in a loop

- Show a menu and exit option

Recap

- Used Scanner for input

- Applied operators

- Used control flow (switch)

- Built a working console calculator

Introduction to Classes and Objects

THE FOUNDATION OF OBJECT-ORIENTED PROGRAMMING IN JAVA

Why OOP?

- Organizes code into reusable units

- Models real-world entities (Car, Student, BankAccount)

- Encourages modularity, readability, and reusability

- Java is 100% Object-Oriented (except primitives)

What is a Class?

- A blueprint for creating objects

- Defines attributes (fields) and behaviors (methods)

```
class ClassName {
    // fields
    // methods
}
```

Example Class

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

    void drive() {
        System.out.println("The car is driving...");
    }
}
```

What is an Object?

- An instance of a class

- Created using new keyword

```
Car myCar = new Car();
```

- Multiple objects can be created from the same class

Example: Creating and Using Objects

```
class Car {
    String color;
    void drive() {
        System.out.println(color + " car is driving...");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.color = "Red";
        c1.drive();
    }
}
```

Memory Representation

- Class → only definition (no memory until object created)

- Object → occupies memory when created

- Example: Two Car objects each with their own color

Real-Life Analogy

- Class = Blueprint of a House

- Object = Actual Houses built from the blueprint

- All houses share the design but have their own features

Practice Exercise

Create a class Student with:

- Fields: name, age

- Method: displayInfo() to print details

In main(), create two student objects and display their info

Recap

- Class → Blueprint

- Object → Instance of a class

- Use new keyword to create objects

- Objects have their own copy of fields

Constructors and Methods

BUILDING AND USING OBJECT FUNCTIONALITY IN JAVA

What are Methods?

- Block of code that performs a task

- Helps reuse code

Syntax:

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

Example Method

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

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int sum = calc.add(5, 3);
        System.out.println("Sum: " + sum);
    }
}
```

Types of Methods

- Instance Methods → Need an object

- Static Methods → Belong to class, no object needed

Example:

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

What is a Constructor?

- Special method to initialize objects

- Same name as class

- Called automatically when object is created

- No return type

Constructor Example

```
class Student {
    String name;

    Student(String n) {
        name = n;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice");
        System.out.println(s1.name);
    }
}
```

Types of Constructors

- Default Constructor (no parameters)

- Parameterized Constructor

Example:

```
Student() { }

Student(String n) { name = n; }
```

Constructor vs Method

- Constructor:
  - Initializes object
  - No return type
  - Same name as class

- Method:
  - Defines behavior
  - Has return type
  - Any name

Practice Exercise

Create a class Book with:

- Fields: title, author

- Constructor to initialize fields

- Method showDetails() to print book info

Create two books in main() and display their details

Recap

- Methods: define behaviors, reusable code

- Static vs Instance methods

- Constructors: initialize objects

- Types: Default and Parameterized

Access Modifiers, this & static

CONTROLLING ACCESS AND SHARING DATA ACROSS OBJECTS

Access Modifiers

- Define visibility of classes, variables, and methods

- Types in Java:

  - public – Accessible everywhere

  - private – Accessible only in the same class

  - protected – Accessible in the same package + subclasses

  - default (no modifier) – Accessible within the same package

Note: For classes only public and default modifiers are allowed.

Access Modifiers

Modifier	Class	Package	Subclass (in other package)	World
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default	✅	✅	❌	❌
private	✅	❌	❌	❌

Example – Access Modifiers

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

    public void setAge(int a) {
        age = a;
    }
    public int getAge() {
        return age;
    }
}
```

The this Keyword

- Refers to current object

- Used to:

  - Differentiate between instance variable and parameter

  - Call other constructors

  - Pass the current object

Example – this

```
class Student {
    String name;

    Student(String name) {
        this.name = name; // distinguish between variable & parameter
    }
}
```

The static Keyword

- Belongs to the class, not to objects

- Shared across all objects

- Can be:

  - Static variable

  - Static method

  - Static block

Example – static

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

   Counter() {
      count++;
      System.out.println("Count: " + count);
   }
}

public class Main {
   public static void main(String[] args) {
      new Counter();
      new Counter();
   }
}
```

Combining Concepts

```
class Library {
    static int totalBooks = 0;
    String title;

    Library(String title) {
        this.title = title;
        totalBooks++;
    }
}
```

Practice Exercise

Create a class BankAccount with:

- private balance

- public methods: deposit, withdraw, getBalance

- static field: bankName (same for all accounts)

Recap

- Access Modifiers: control visibility

- this: refers to current object

- static: belongs to the class, shared across objects

Inheritance and super

REUSING CODE WITH PARENT–CHILD RELATIONSHIPS

What is Inheritance?

- Mechanism to acquire properties & methods of another class

- Promotes code reusability and method overriding

- Syntax:

class Child extends Parent { }

Example of Inheritance

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

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

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

Types of Inheritance in Java

- Single Inheritance – One parent, one child

- Multilevel Inheritance – Child → Parent → Grandparent

- Hierarchical Inheritance – One parent, many children

⚠ Java does not support Multiple Inheritance (avoids ambiguity)

The super Keyword

- Refers to the parent class
- Used to:
  - Access parent class variables
  - Call parent class methods
  - Call parent class constructor

Example – super with Constructor

```
class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }
}

class Dog extends Animal {
    Dog() {
        super();
        System.out.println("Dog constructor");
    }
}
```

Example – super with Methods

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

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

IS-A Relationship

- Dog IS-A Animal

- Student IS-A Person

- Helps represent real-world hierarchies

Practice Exercise

Create a class Vehicle with fields brand, speed.

- Create subclass Car with field seats.

- Use constructor chaining with super.

- Print all details using a method.

Recap

- Inheritance: extends keyword

- Promotes code reuse

- super: access parent class methods & constructors

- Types: Single, Multilevel, Hierarchical

Encapsulation & Abstraction

HIDING DETAILS AND EXPOSING ONLY WHAT'S NECESSARY

What is Encapsulation?

- Wrapping data (fields) and code (methods) together

- Protects data from direct access

- Achieved using:

  - private variables

  - public getters & setters

Encapsulation Example

```
class BankAccount {
    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        if(balance >= amount) balance -= amount;
        else System.out.println("Insufficient balance");
    }
    public double getBalance() { return balance; }
}
```

Benefits of Encapsulation

- Data hiding
- Controlled access
- Flexibility & maintainability
- Increased security

What is Abstraction?

- Showing essential features and hiding details

- Achieved using:

  - Abstract Classes

  - Interfaces

Abstract Class Example

```
abstract class Vehicle {
    abstract void start();
}
class Car extends Vehicle {
    void start() { System.out.println("Car starts with a key"); }
}
```

Interface Example

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

Encapsulation vs Abstraction

Feature	Encapsulation	Abstraction
Purpose	Hides data	Hides Implementation
How	Private fields + public getter/settters	Abstract classes & inheritances
Focus	Data Protection	Design level

Practice Exercise

Create a class Employee with:

- private fields: name, salary

- public getters & setters

- Abstract class Role with abstract method work()

- Subclass Manager implements work()

Recap

- Encapsulation → data hiding with getters/setters

- Abstraction → focus on essential, hide implementation

- Tools: private, abstract classes, interfaces

Polymorphism: Overloading & Overriding

ONE NAME, MANY FORMS

What is Polymorphism?

- “Poly” = many, “Morph” = forms

- Same method name → different behaviors

- Two types in Java:

  1. Compile-Time Polymorphism (Method Overloading)

  2. Runtime Polymorphism (Method Overriding)

Method Overloading

- Same method name
- Different parameter (type, number, or order)
- Resolved at compile time

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

Rules for Overloading

✓ Must differ in number/type/order of parameters

✓ Can have different return types if parameters differ

✗ Cannot overload only by changing return type

Method Overriding

- Redefining a method of parent class in child class

- Same method name & parameters

- Resolved at runtime

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

Using @Override Annotation

- Helps compiler check correctness

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

Difference Between Overloading & Overriding

Feature	Overloading	Overriding
Binding Time	Compile-Time	Runtime
Parameters	Must differ	Must be same
Return Type	Can Differ	Must be same/subtype
Inheritance	Not required	Requires Inheritance

Example with Both

```
class Shape {
    void area() { System.out.println("Shape area"); }
}
class Circle extends Shape {
    @Override
    void area() { System.out.println("Circle area"); }
    double area(double r) { return 3.14 * r * r; }
}
```

Practice Exercise

Create a class Calculator that:

- Overloads method multiply for 2 and 3 numbers

- Create subclass ScientificCalculator that overrides multiply to show “Scientific multiplication done”

Recap

- Polymorphism = one name, many forms

- Overloading → Compile-Time

- Overriding → Runtime

- Use @Override for clarity

If-Else & Nested If

DECISION-MAKING IN JAVA PROGRAMS

Introduction to Control Flow

- Control flow decides which code block executes
- Uses conditions (true / false)
- Common structures:
  - if
  - if-else
  - nested if
  - switch

If Statement

```
if (condition) {
    // code runs if condition is true
}
```

Example:

```
int age = 20;
if(age >= 18) {
    System.out.println("You are an adult.");
}
```

If-Else Statement

```
if (condition) {
   // executes if true
} else {
   // executes if false
}
```

Example:

```
int marks = 40;
if(marks >= 50)
   System.out.println("Pass");
else
   System.out.println("Fail");
```

Else-If Ladder

```
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");
```

Nested If

- An if statement inside another if

- Useful for multiple conditions

```
int age = 25;
boolean hasID = true;

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

Best Practices

- Keep nesting minimal for readability

- Use else-if ladder when many conditions

- Prefer switch for multiple discrete values

Practice Exercise

Write a program to check:

- If marks ≥ 90 → "Excellent"

- If marks 75–89 → "Very Good"

- If marks 50–74 → "Good"

- Otherwise → "Needs Improvement"

Recap

- if → single condition

- if-else → two outcomes

- else-if ladder → multiple conditions

- nested if → condition inside another

Switch Case in Java

SIMPLIFYING MULTIPLE CHOICE DECISIONS

What is Switch Case?

- A control statement to handle multiple choices

- Cleaner than multiple else-if statements

- Works with:

  - int, char, String, enums

Syntax of Switch

```
switch(expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code if no match
}
```

Example – Days of Week

```
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;
    default: System.out.println("Invalid day");
}
```

Switch with String

```
String fruit = "Apple";
switch(fruit) {
    case "Apple": System.out.println("Red fruit"); break;
    case "Mango": System.out.println("King of fruits"); break;
    default: System.out.println("Unknown fruit");
}
```

Important Points

- break prevents fall-through

- default is optional

- Case values must be unique constants

- switch works with primitives, enums, and String

Example Without Break

```
int number = 2;
switch(number) {
   case 1: System.out.println("One");
   case 2: System.out.println("Two");
   case 3: System.out.println("Three");
}
```

Output:

```
Two
Three
```

Practice Exercise

Write a program using switch:

- Input a number 1–7

- Print the day of the week (Monday–Sunday)

- If invalid, print “Invalid choice”

Recap

- Switch is an alternative to multiple if-else

- Use break to avoid fall-through

- Can work with int, char, String, enum

While Loop & Do-While Loop

REPEATING TASKS UNTIL A CONDITION IS FALSE

What is a Loop?

- A loop executes a block of code repeatedly
- Controlled by a condition
- Java loops:
  - while
  - do-while
  - for

While Loop

```
while (condition) {
   // code executes while condition is true
}
```

Example:

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

Flow of While Loop

1. Check condition

2. If true → execute body

3. Increment/update

4. Repeat until condition false

Do-While Loop

```
do {
   // code executes once, then checks condition
} while (condition);
```

Example:

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

Difference Between While & Do-While

Feature	While Loop	Do-While Loop
Condition Check	Before body execution	After body execution
Executes Once	Not guaranteed	Always at least once

Example – Menu Driven

```
int choice;
Scanner sc = new Scanner(System.in);
do {
    System.out.println("1. Say Hello\n2. Exit");
    choice = sc.nextInt();
    if(choice == 1)
        System.out.println("Hello!");
} while(choice != 2);
```

Practice Exercise

Write a program to print the multiplication table of a number using:

- while loop

- do-while loop

Recap

- while loop → checks condition first

- do-while loop → executes at least once

- Use when you need repeated execution

For Loop & Nested Loops

MASTERING ITERATION WITH FOR LOOPS

Introduction

- For loop → executes a block repeatedly

- Syntax combines:

  - Initialization

  - Condition

  - Update

- Suitable for count-controlled loops

For Loop Syntax

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

Example:

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

Flow of For Loop

1. Initialization (run once)

2. Check condition

3. Execute body

4. Update

5. Repeat until condition false

Nested For Loops

- A loop inside another loop

- Useful for:

  - Printing patterns

  - Working with 2D arrays

Example:

```
for(int i = 1; i <= 3; i++) {
    for(int j = 1; j <= 3; j++) {
        System.out.print(i + "," + j + " ");
    }
    System.out.println();
}
```

Example – Multiplication Table

```
for(int i = 1; i <= 5; i++) {
    for(int j = 1; j <= 10; j++) {
        System.out.print(i*j + " ");
    }
    System.out.println();
}
```

Printing Pattern

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

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Practice Exercise

Write a program using nested for loops to print a right-angled triangle of numbers:

```
1

1 2

1 2 3

1 2 3 4
```

Recap

- For loop combines init, condition, update

- Nested loops help with patterns & grids

- Be careful with performance in deep nesting

Java Collections: ArrayList & LinkedList

WORKING WITH DYNAMIC DATA STRUCTURES

Introduction to Collections

- Collections are resizable data structures

- Provide ready-made methods for data handling

- Common interfaces: List, Set, Map

- Today: ArrayList and LinkedList

ArrayList Overview

- Implements List interface

- Dynamic array (resizable)

- Allows duplicates and maintains insertion order

- Fast random access, slower insert/delete

ArrayList Example

```
import java.util.*;


class Example {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");


        System.out.println(fruits);
    }
}
```

LinkedList Overview

- Implements List & Deque interfaces

- Doubly linked list implementation

- Allows duplicates and maintains insertion order

- Fast insert/delete, slower random access

LinkedList Example

```
import java.util.*;


class Example {
    public static void main(String[] args) {
        LinkedList<Integer> numbers = new LinkedList<>();
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);


        System.out.println(numbers);
    }
}
```

ArrayList vs LinkedList

Feature	ArrayList	LinkedList
Memory	Less memory	More memory (links)
Access Speed	Fast (index-based)	Slow (sequential)
Insertion/Del	Slower (shifting)	Faster (no shifting)

Practice Exercise

- Create an ArrayList of student names

- Add 5 names, remove 1, and display all

- Create a LinkedList of integers

- Insert numbers at beginning and end

Recap

- ArrayList → dynamic array

- LinkedList → doubly linked list

- Use ArrayList for fast access, LinkedList for frequent insert/delete

Java Collections: HashSet & HashMap

WORKING WITH DYNAMIC DATA STRUCTURES

Introduction to HashSet

- A part of Java Collections Framework
- Implements Set interface
- Stores unique elements only
- No guaranteed order of elements
- Allows one null element

HashSet Example

Example:

```
HashSet<String> set = new HashSet<>();

set.add("Apple");

set.add("Banana");

set.add("Apple"); // Duplicate ignored

System.out.println(set);
```

Introduction to HashMap

- Implements Map interface
- Stores key-value pairs
- Keys must be unique, values can be duplicate
- Allows one null key and multiple null values
- No guaranteed order of keys

HashMap Example

Example:

```
HashMap<Integer, String> map = new HashMap<>();

map.put(1, "Apple");

map.put(2, "Banana");

map.put(3, "Cherry");

System.out.println(map);
```

HashSet vs HashMap

HashSet:

- Stores only unique elements
- Backed internally by HashMap
- No mapping, just values

HashMap:

- Stores key-value pairs
- Keys must be unique
- Useful for fast lookup

Practice Exercises

1. Create a HashSet of integers and remove duplicates.

2. Create a HashMap storing student roll numbers and names.

3. Retrieve a value from HashMap using a key.

4. Iterate through a HashMap and print key-value pairs.

Exception Handling Basics

TRY-CATCH-FINALLY

What is an Exception?

- An exception is an event that disrupts the normal flow of a program.
- Occurs during runtime.
- Examples: Divide by zero, null pointer access, file not found.

Types of Exceptions

- Checked Exceptions – Must be handled using try-catch or declared with throws.

  Example: IOException, SQLException

- Unchecked Exceptions – Occur at runtime, not checked by compiler.

  Example: NullPointerException, ArithmeticException

try-catch Block

Syntax:

```
try {
   // Code that may cause exception
} catch (ExceptionType e) {
   // Handling code
}
```

finally Block

- Always executes, whether exception occurs or not.
- Used for cleanup code like closing files or database connections.

Syntax:

```
try {
  // Code
} catch (Exception e) {
  // Handle
} finally {
  // Cleanup code
}
```

Example: try-catch-finally

Example:

```
try {
   int a = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("Division by zero!");
} finally {
   System.out.println("End of program.");
}
```

Practice Exercises

1. Write a program to divide two numbers and handle division by zero.

2. Create a program that tries to read a file and handles FileNotFoundException.

3. Demonstrate a finally block with a database connection close simulation.

Custom Exceptions And throw vs throws

WORKING WITH EXCEPTION HANDLING

What are Custom Exceptions?

- User-defined exceptions created by extending Exception class.
- Useful when built-in exceptions are not sufficient.
- Helps in making code more readable and meaningful.

Creating a Custom Exception

Syntax:

```
class MyException extends Exception {
   public MyException(String message) {
       super(message);
   }
}
```

Using a Custom Exception

Example:

```
public class TestCustomException {
  static void validate(int age) throws MyException {
    if(age < 18)
      throw new MyException("Not valid age");
    else
      System.out.println("Welcome!");
  }
  public static void main(String args[]) {
    try { validate(16); }
    catch(Exception e) { System.out.println(e); }
  }
}
```

throw vs throws

throw:

- Used to explicitly throw an exception.
- Followed by an instance of Exception.

Example: throw new ArithmeticException("Error");

throws:

- Used in method signature to declare exceptions.
- Indicates method might throw exceptions.

Example: void method() throws IOException

Example: throw and throws

```
public void readFile() throws IOException {
   FileReader fr = new FileReader("file.txt");
   throw new IOException("File error");
}
```

Practice Exercises

1. Create a custom exception for invalid password.

2. Write a program that throws a custom exception if a student’s marks are negative.

3. Create a method that throws IOException and handle it in main().

Java 8 Basics - Lambdas & Streams

OPTIONAL FOR FAST LEARNERS

Introduction to Java 8 Features

- Java 8 introduced functional programming

- Lambdas and Streams are key features

- Improves code readability and efficiency

What is a Lambda Expression?

- A lambda is an anonymous function

- Syntax: (parameters) -> expression/body

- Example: (a, b) -> a + b

- Used to implement functional interfaces

Functional Interfaces

- Interface with one abstract method

- Examples: Runnable, Comparator, Predicate

- Annotated with @FunctionalInterface

Lambda Syntax Examples

Example 1:

```
Runnable r = () -> System.out.println("Hello");
```

Example 2:

```
Comparator<Integer> c = (a, b) -> a - b;
```

What is Stream API?

- Stream API processes collections in a functional style

- Supports map, filter, reduce, and more

- Enables easy parallel processing

Stream API - Example

```
List<String> names = Arrays.asList("John", "Jane", "Jack");

names.stream()

    .filter(name -> name.startsWith("J"))

    .forEach(System.out::println);
```

Benefits of Lambdas & Streams

- Concise and readable code
- Encourages functional programming
- Parallel processing made easy
- Reduces boilerplate code

Practice Exercises

- Create a Runnable using lambda

- Sort a list using Comparator lambda

- Use Stream to filter, map and collect

Mini Project: Library Management System

BRINGING OOP CONCEPTS TOGETHER

Project Overview

- Build a console-based Library Management System
- Concepts covered:
  - Classes & Objects
  - Constructors
  - Access Modifiers
  - Inheritance & Polymorphism
  - Encapsulation & Abstraction

Features

- Add new books

- Display available books

- Issue a book

- Return a book

Class Design

- Book
  - Fields: id, title, author, isIssued
- Library
  - Collection of books
  - Methods: addBook(), showBooks(), issueBook(), returnBook()
- Main
  - Menu-driven program

Book Class

```
class Book {
    int id;
    String title, author;
    boolean isIssued;

    Book(int id, String title, String author) {
        this.id = id;
        this.title = title;
        this.author = author;
        this.isIssued = false;
    }
}
```

Library Class

```
import java.util.*;


class Library {
    ArrayList<Book> books = new ArrayList<>();


    void addBook(Book b) { books.add(b); }


    void showBooks() {
        for(Book b : books)
            System.out.println(b.id + " " + b.title + " by " + b.author +
                               (b.isIssued ? " [Issued]" : " [Available]"));
    }
}
```

Issue & Return Methods

```
void issueBook(int id) {
    for(Book b : books) {
        if(b.id == id && !b.isIssued) {
            b.isIssued = true;
            System.out.println("Book issued: " + b.title);
            return;
        }
    }
    System.out.println("Book not available.");
}
```

```
void returnBook(int id) {
    for(Book b : books) {
        if(b.id == id && b.isIssued) {
            b.isIssued = false;
            System.out.println("Book returned: " + b.title);
            return;
        }
    }
    System.out.println("Invalid return.");
}
```

Main Class

```
public class Main {
    public static void main(String[] args) {
        Library lib = new Library();
        lib.addBook(new Book(1, "Java Basics", "John Doe"));
        lib.addBook(new Book(2, "OOP Concepts", "Jane Smith"));
        lib.showBooks();
    }
}
```

Practice Task

Extend the project with:

- Menu-driven input using Scanner

- Option to add new books from user

- Handle cases where book ID is not found

Recap

- Applied all OOP concepts

- Designed multiple classes

- Used ArrayList for collections

- Practiced real-world project design

Project: Menu-Driven ATM Simulator

CONTROL FLOW + ARRAYS

Project Overview

- Build a console-based ATM Simulator

- Concepts used:

  - if-else and switch

  - loops

  - arrays

  - methods

Features

- User Login with PIN

- Check Balance

- Deposit Money

- Withdraw Money

- Exit Application

Class Design

- ATM Class
  - balance (double)
  - pin (int)
  - Methods: checkBalance(), deposit(), withdraw()
- Main Class
  - Menu-driven interface

ATM Class Example

```
class ATM {
    double balance;
    int pin;

    ATM(double balance, int pin) {
        this.balance = balance;
        this.pin = pin;
    }

    void checkBalance() {
        System.out.println("Balance: Rs." + balance);
    }

    void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited Rs." + amount);
    }

    void withdraw(double amount) {
        if(amount <= balance) {
            balance -= amount;
            System.out.println("Withdrew Rs." + amount);
```

Main Program Structure

```
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ATM atm = new ATM(10000, 1234);

        System.out.print("Enter PIN: ");
        int inputPin = sc.nextInt();

        if(inputPin == atm.pin) {
            int choice;
            do {
                System.out.println("1.Check Balance\n2.Deposit\n3.Withdraw\n4.Exit");
                choice = sc.nextInt();
                switch(choice) {
                    case 1: atm.checkBalance(); break;
                    case 2: System.out.print("Enter amount: ");
                            atm.deposit(sc.nextDouble()); break;
                    case 3: System.out.print("Enter amount: ");
                            atm.withdraw(sc.nextDouble()); break;
                    case 4: System.out.println("Thank you for using ATM"); break;
                    default: System.out.println("Invalid choice");
                }
```

Practice Task

- Enhance the project with:
  - Multiple users using arrays
  - Daily withdrawal limit
  - Display transaction history

Recap

- Built a console ATM Simulator

- Applied control flow, loops, and arrays

- Practiced real-world application design

Final Project Presentation

WRAP-UP & DEMONSTRATION

Objective of Final Project

- Apply all Java concepts learned

- Demonstrate problem-solving and coding skills

- Showcase project structure, logic, and UI (if any)

- Practice explaining code to others

Project Requirements

- Must include OOP concepts: Classes, Inheritance, Polymorphism

- Use of Collections (ArrayList, HashMap, etc.)

- Proper control flow and exception handling

- Optional: File handling, Java 8 features

Suggested Project Ideas

- Library Management System

- Student Grade Tracker

- Simple Banking Application

- Online Quiz Application

Presentation Guidelines

- Briefly explain the problem your project solves

- Walk through the code and logic

- Demonstrate running the program

- Highlight use of key Java concepts

Evaluation Criteria

- Functionality and completeness
- Use of Java concepts
- Code structure and readability
- Presentation clarity

Tips for Success

- Start with a clear plan

- Break code into small manageable parts

- Test frequently

- Keep code clean and commented

Q&A and Feedback

- Be ready to answer questions about your code

- Accept feedback positively

- Use it to improve future projects