Name: Punit Manoj Bhatarkar

1.What is Java? Explain its features.

Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle). It is widely used for building desktop, web, and mobile applications. Java is popular because of its **platform independence** — you can write code once and run it anywhere using the **Java Virtual Machine (JVM)**.

Key Features of Java:

1. **Platform Independent**: Java programs can run on any system with JVM — *"Write once, run anywhere."*
2. **Object-Oriented**: Java uses objects and classes, supporting principles like inheritance, encapsulation, and polymorphism.
3. **Simple and Easy to Learn**: Java has a clean and readable syntax.
4. **Secure**: Java provides a secure environment through runtime checking, bytecode verification, and restricted access.
5. **Robust**: Strong memory management and error-handling features (like try-catch blocks).
6. **Multithreaded**: Supports multiple threads of execution (helps in games, animations, etc.).
7. **High Performance**: Uses Just-In-Time (JIT) compiler for faster execution.
8. **Distributed**: Java can build distributed applications using RMI and sockets.
9. **Dynamic**: Java programs can load classes at runtime, making it flexible.

2. Explain the Java program execution process.

Java Program Execution Process Using Notepad, CMD, and JDK

1. **Install JDK**
    - Download and install the Java Development Kit (JDK) from Oracle's official site.
2. **Set Environment Variable**

- Add the JDK bin folder path to the system's PATH variable so you can run javac and java from anywhere in CMD.

3. **Write Code in Notepad**
   - Open Notepad, write your Java code, and save the file with a .java extension (e.g., Hello.java).
4. **Open CMD**
   - Open Command Prompt and navigate to the folder where your .java file is saved using the cd command.
5. **Compile the Program** ○ Use javac FileName.java to compile the file. This creates a .class bytecode file.
6. **Run the Program** ○ Use java FileName (without .class extension) to run the program.
7. **JVM Executes the Program** ○ The Java Virtual Machine loads the .class file and executes the bytecode.

Java Program Execution Using VS Code and JDK

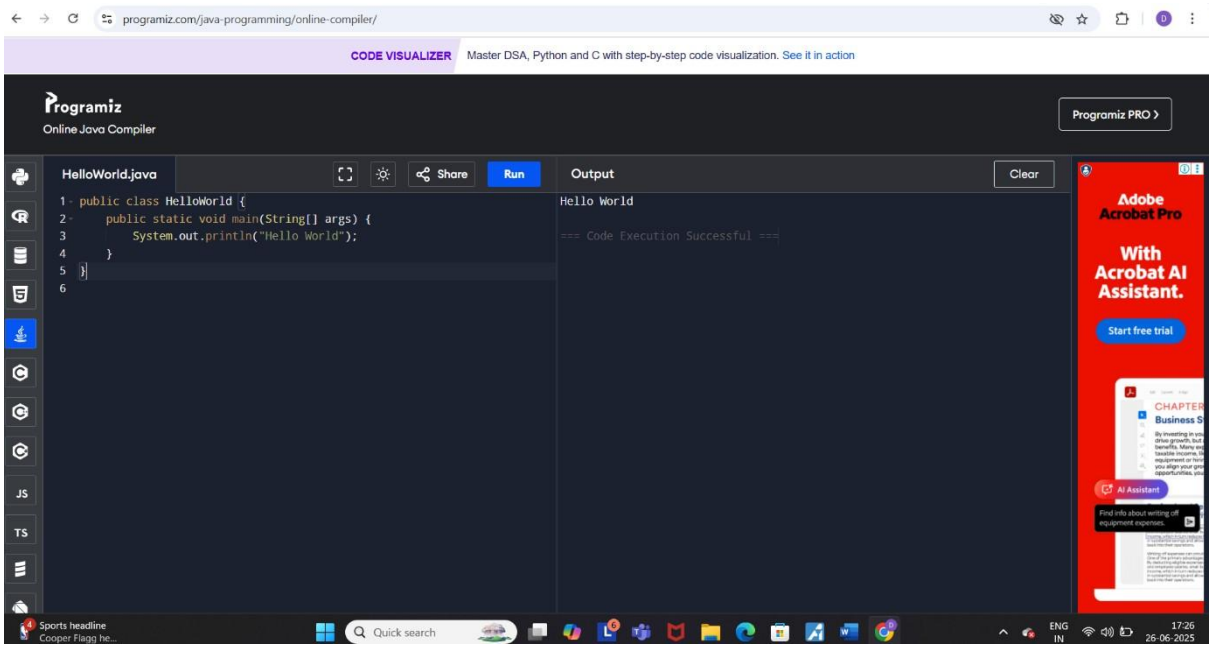
1. **Install JDK** ○ Download and install the latest **Java Development Kit (JDK)**.
2. **Install VS Code** ○ Download and install **Visual Studio Code** editor.
3. **Install Java Extension Pack in VS Code**
   - Open VS Code, go to Extensions, and install **"Java Extension Pack"** by Microsoft.
4. **Write Java Program**
   - Create a new file with .java extension (e.g., HelloWorld.java) and write your Java code.
5. **Save the File** ○ Save your file in a dedicated folder/project directory.
6. **Run the Program**
   - Click the **Run button** (▶) at the top or right-click and select **"Run Java"**.
   - OR use the terminal:
     - Compile: javac HelloWorld.java

- Run: java HelloWorld

7. **Program Executes** ○ The output will be displayed in the **terminal window** inside VS Code.

3.Write a simple Java program to display 'Hello World'

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



4. What are Data Types in Java? List and Explain Them.

In Java, **data types** specify the kind of values a variable can store. Java is a statically-typed language, so each variable must be declared with a data type

1. Primitive Data Types (Basic)

These are the built-in, fixed-size types used to store simple values like numbers, characters, and logical values.

- **byte** – small whole numbers (e.g., age)

- **short** – slightly larger whole numbers
- **int** – commonly used for integers
- **long** – very large whole numbers
- **float** – decimal numbers (less precision)
- **double** – decimal numbers (more precision)
- **char** – single character (e.g., 'A', '3')
- **boolean** – stores true or false

These types directly store values in memory and are not objects.

2. Non-Primitive Data Types (Reference Types)

These types are based on **classes** and are used to store complex data.

a. Class

A class is a blueprint for creating objects. It defines **properties (variables)** and **methods (functions)**. For example, a Car class might have color, speed, and a method drive(). **b. Object**

An object is an instance of a class. It represents a real-world entity created from the class blueprint. For example, Car myCar = new Car();

c. Interface

An interface defines a set of abstract methods (without implementation). A class implements an interface to follow its structure. It's used to achieve abstraction and multiple inheritance. Example: interface Printable { void print(); }

d. String – Sequence of characters (e.g., "Hello")

e. Arrays – Collection of elements (e.g., int[] arr = {1, 2, 3};)

5. What is the Difference Between JDK, JRE, and JVM?

1. JVM (Java Virtual Machine)

- **JVM** is the **engine** that runs Java bytecode.
- It provides a **runtime environment** for executing Java programs.
- It is **platform-dependent**, meaning each OS has its own version of JVM.
- It handles memory management, garbage collection, and bytecode execution.

2. JRE (Java Runtime Environment)

- **JRE** is a **package** that contains **JVM + Java class libraries** (like rt.jar).
- It is used to **run** Java applications but **cannot compile** them.
- It does **not include development tools** like the Java compiler.

3. JDK (Java Development Kit)

- **JDK** is a full **development package** for Java.
- It contains everything in the **JRE**, **plus** tools like javac (compiler), javadoc, and debugger.
- It is used to **write, compile, and run** Java programs.

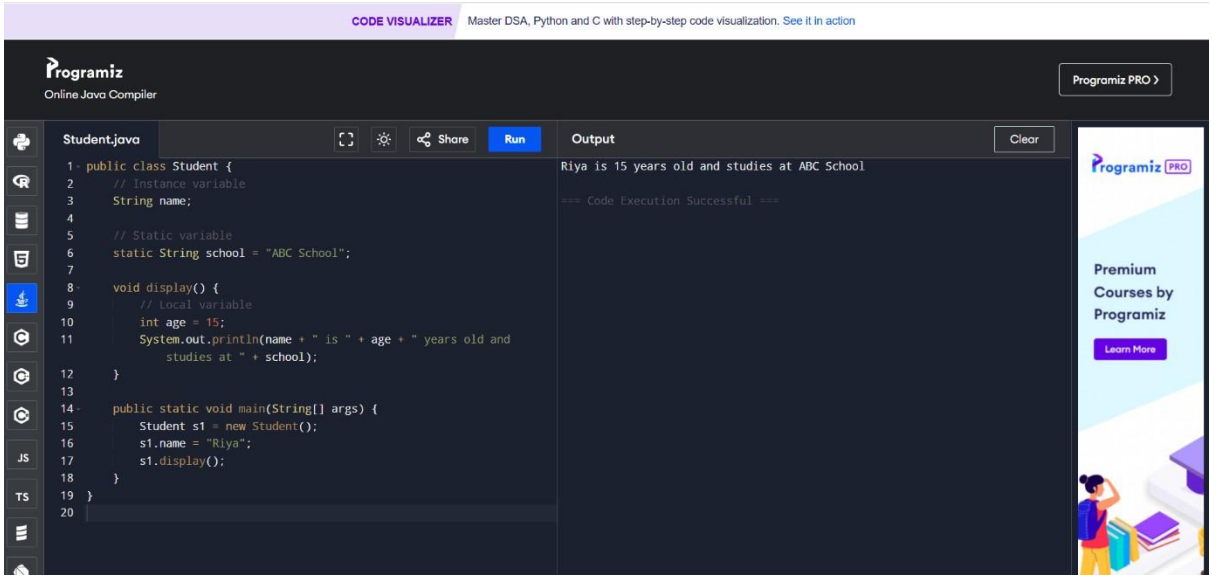
6. What are Variables in Java? Explain with Examples.

What is a Variable?

In Java, a **variable** is a **name given to a memory location** that stores a value. It acts like a container to hold data during the execution of a program. Every variable must have a **data type**, which defines what kind of value it can store.

Types of Variables in Java:

1. **Local Variable** ○ Declared inside a method or block. ○ Only accessible within that block.
    - Must be initialized before use.
2. **Instance Variable** ○ Declared inside a class but **outside methods**. ○ Belongs to the object (each object has its own copy).
    - Does **not use static** keyword.
3. **Static Variable** (Class Variable) ○ Declared with the static keyword.
    - Belongs to the **class**, not objects. ○ All objects share the same static variable.



7.What are the different types of operators in Java?

In Java, **operators** are special symbols used to perform **operations** on variables and values. They help in performing tasks like arithmetic, comparison, assignment, etc.

Types of Operators in Java:

1. Arithmetic Operators

Used for basic mathematical operations:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus - Remainder) **2. Relational (Comparison) Operators**

Used to compare two values:

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

3. Logical Operators

Used for logical conditions:

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

4. Assignment Operators

Used to assign values to variables:

- = (Assign)
- +=, -=, *=, /=, %= (Compound assignment)

5. Unary Operators

Operate on a single operand:

- +, - (Unary plus, minus)
- ++ (Increment)
- -- (Decrement)
- ! (Logical NOT) **6. Bitwise Operators** Operate on bits:
- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise Complement)
- <<, >>, >>> (Bit shifts)

8. Explain control statements in Java (if, if-else, switch).

`if` Statement

Syntax

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

Ex.

```
int number = 10; if
(number > 0) {
   System.out.println("Positive number");
```

}

if-else

Statement

Syntax if

(condition) { //

code if true

} else {

  // code if false

}

Ex.

int number = -5; if

(number >= 0) {

  System.out.println("Positive");

} else {

  System.out.println("Negative");

}

else-if Ladder

Syntax if

(condition1) {

// code

} else if (condition2) {

  // code

} else {

  // default code

}

Ex.

int marks = 75; if

(marks >= 90) {

  System.out.println("Grade A"); }

else if (marks >= 60) {

  System.out.println("Grade B"); }

else {

  System.out.println("Grade C");

}

switch Statement

Syntax switch

(expression) {

case value1: //

code break;

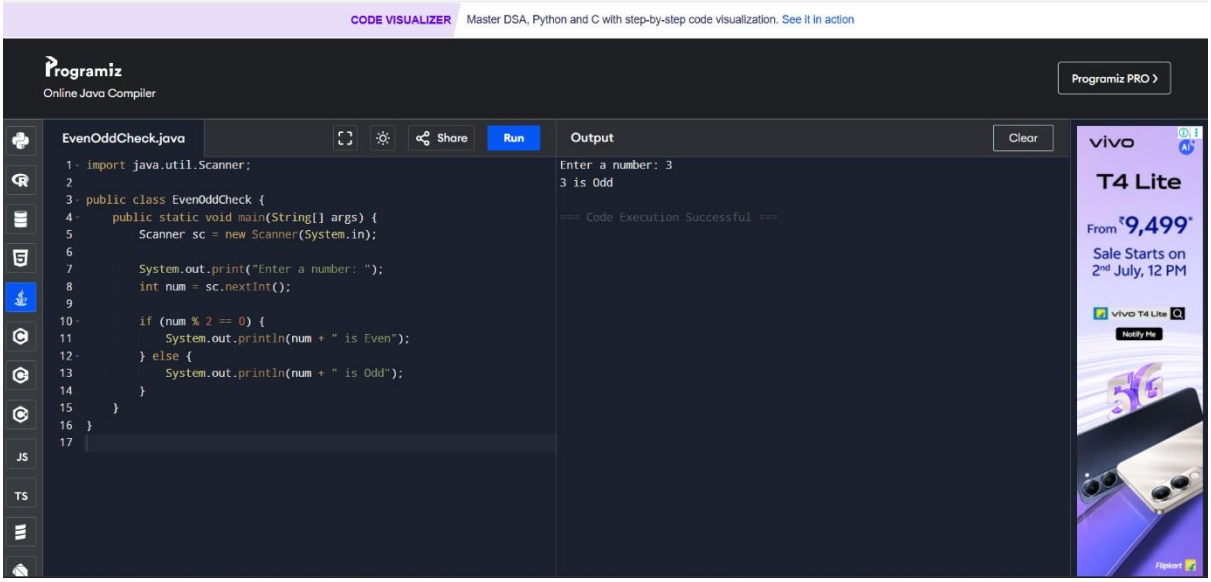
case value2: //

```
code        break;
default:
      // default code
}
```

Ex.

```
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");
}
```

9. Write a Java program to find whether a number is even or odd.



10.What is the difference between while and do-while loop?

While loop

In a **while loop**, the condition is checked **before** the loop body executes

A while loop may **not run at all** if the condition is false initially.

Use while when you want to check the condition **first**, and do-while when the loop must run **at least once**. Syntax while (condition) {

```
    // code to execute repeatedly
}
```

Ex.

```
int i = 1; while
(i <= 3) {
    System.out.println("Count: " + i);
    i++;
}
```

Output

Count: 1

Count: 2

Count: 3

Do-While

In a **do-while loop**, the condition is checked **after** the loop body executes.

A do-while loop **always runs at least once**, even if the condition is false.

Syntax

```
do {
    // code to execute repeatedly
} while (condition);
```

Ex.

```
int i
= 1; do
{
    System.out.println("Count: " + i);
    i++;
} while (i <= 3);
```

Output

Count: 1

Count: 2

Count: 3

2.Object-Oriented Programming (OOPs)

1. What are the main principles of OOPs in Java? Explain each.

Object-Oriented Programming (OOP) in Java is based on a set of core principles that aim to make software more modular, reusable, and easier to maintain. The main principles of OOP in Java are:

1. Encapsulation

Definition: Encapsulation is the practice of hiding the internal state and behavior of an object and exposing only what is necessary.

How it works:

- Achieved by using **private** variables and **public** getter and setter methods.
- Keeps the data safe from outside interference and misuse.

Ex.

```
public class Person {

private String name;


   public String getName() {

return name;

   }


   public void setName(String name) {

this.name = name;

   }

}
```

2. Inheritance

Definition: Inheritance allows one class to acquire the properties (fields) and behaviors (methods) of another class.

How it works:

- The class that inherits is called the **subclass** or **child class**.
- The class being inherited from is the **superclass** or **parent class**.
- Promotes **code reuse**.

Ex.

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


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

3. Polymorphism

Definition: Polymorphism means "many forms" and allows objects to be treated as instances of their parent class rather than their actual class.

Types:

- **Compile-time polymorphism (Method Overloading)**: Multiple methods with the same name but different parameters.
- **Runtime polymorphism (Method Overriding)**: A subclass provides a specific implementation of a method already defined in its superclass.

Ex.

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

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

4. Abstraction

Definition: Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object.

How it works:

- Achieved using **abstract classes** and **interfaces**.
- Helps in reducing complexity and isolating impact of changes.

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

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

2. What is a Class and an Object in Java?

Class in Java

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

Syntax:

```
class ClassName {
```

```
    // fields (variables)
    // methods
} Ex.
class Car {
    // Properties
    String color;
    int speed;

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

Object in Java

An **object** is a real-world **instance** of a class. It contains actual values and can use the methods defined in the class

Syntax:

```
ClassName obj = new ClassName();
```

Example:

```
public class Main {     public static void
main(String[] args) {        Car myCar = new
Car();   // Creating object        myCar.color =
"Red";     // Setting property        myCar.speed
= 100;

        System.out.println("Car color: " + myCar.color);
System.out.println("Car speed: " + myCar.speed);        myCar.drive();
// Calling method
```

```
    }
}
```

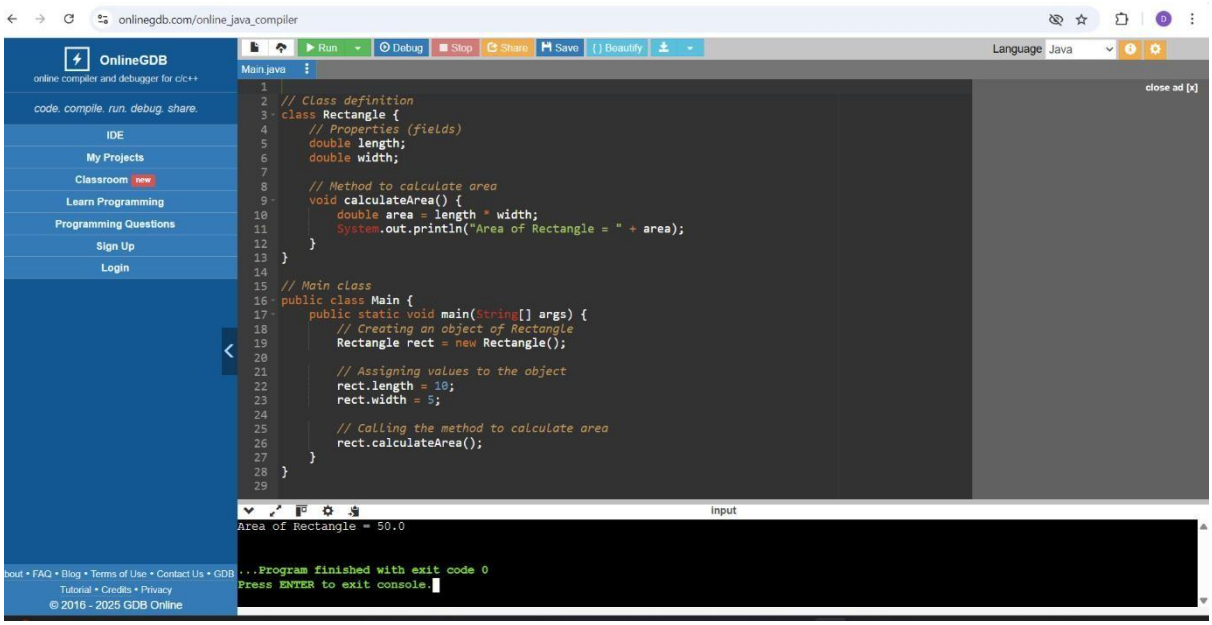
Output:

Car color: Red

Car speed: 100

The car is driving.

3. Write a program using class and object to calculate area of a rectangle



4. Explain inheritance with real-life example and Java code.

What is Inheritance?

Inheritance is one of the main principles of OOP. It allows a class (called **child** or **subclass**) to **inherit** properties and methods from another class (called **parent** or **superclass**).

Real-Life Example:

Think of a **Vehicle** class as a base class.
Then you have **Car**, **Bike**, and **Bus** as subclasses that inherit common properties (like speed, color) and behaviors (like move) from **Vehicle**.

Java Code Example:

```java
// Superclass
class Vehicle {
    String brand = "Generic Vehicle";

    void move() {
        System.out.println("The vehicle is moving.");
    }
}

// Subclass
class Car extends Vehicle {
    String model = "Sedan";

    void honk() {
        System.out.println("The car honks: Beep! Beep!");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();

        // Accessing inherited members
        System.out.println("Brand: " + myCar.brand);
        myCar.move();

        // Accessing subclass members
        System.out.println("Model: " + myCar.model);
        myCar.honk();
    }
}
```

Output

```
input
Brand: Generic Vehicle
The vehicle is moving.
Model: Sedan
The car honks: Beep! Beep!


...Program finished with exit code 0
Press ENTER to exit console.
```

5. What is polymorphism? Explain with compile-time and runtime examples

What is Polymorphism in Java?

Definition:

Polymorphism means "**many forms**". It allows the **same method** or **action** to behave **differently** based on the **object** that is invoking it.

In Java, polymorphism is of **two types**:

1. Compile-Time Polymorphism (Method Overloading)

- **Same method name**, but **different parameter**

list.

- Decided **at compile time**.
- Also called **Static Polymorphism**.

Example:

Main.java

```java
class Calculator {
    // Method with 2 int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    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 1: " + calc.add(5, 10));
        System.out.println("Sum 2: " + calc.add(3, 4, 5));
    }
}
```

input

```
Sum 1: 15
Sum 2: 12


...Program finished with exit code 0
Press ENTER to exit console.
```

2. Runtime Polymorphism (Method Overriding) •

Subclass overrides a method of its **superclass**.

- Resolved **at runtime**.
- Also called **Dynamic Polymorphism**
- **Example:**

Main.java F9

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

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

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

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

        a = new Dog();    // object of subclass
        a.sound();        // Output: Dog barks

        a = new Cat();    // another subclass object
        a.sound();        // Output: Cat meows
    }
}
```

input

```
Dog barks
Cat meows


...Program finished with exit code 0
Press ENTER to exit console.
```

6. What is Method Overloading and Method Overriding in Java?

1. Method Overloading

Definition:

Method Overloading means **defining multiple methods with the same name** in the **same class**, but with **different parameters** (number, type, or order).

- Happens at **compile-time** → **Compile-Time Polymorphism**.
- Improves **readability** and **reusability**.

Example of Method Overloading:

Main.java

```java
class MathOperations {
    // Method with 2 int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations obj = new MathOperations();
        System.out.println("Sum 1: " + obj.add(5, 10));
        System.out.println("Sum 2: " + obj.add(3, 4, 2));
        System.out.println("Sum 3: " + obj.add(2.5, 3.5));
    }
}
```

input

```
Sum 1: 15
Sum 2: 9
Sum 3: 6.0


...Program finished with exit code 0
Press ENTER to exit console.
```

2. Method Overriding

Definition:

Method Overriding means a **subclass** provides a **specific implementation** of a method that is **already defined in its superclass**.

- Happens at **runtime** → **Runtime Polymorphism**.
- Must have **same method name, parameters, and return type**.
- Use @Override annotation (optional but recommended).

Example of Method Overriding:

Main.java

```java
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 Dog(); // Upcasting
        a.sound();            // Calls overridden method in Dog
    }
}
```

input

```
Dog barks


...Program finished with exit code 0
Press ENTER to exit console.
```

7. What is encapsulation? Write a program demonstrating encapsulation.

What is Encapsulation in Java?

Definition:

Encapsulation is the OOP principle of **hiding the internal details** of a class and **restricting direct access** to some of its components.

It's used to **protect data** by making variables **private** and providing **public getter and setter methods**. **Key Concepts:**

- **Private variables** → can't be accessed directly from outside.
- **Public methods** (getters/setters) → used to read/update the values. **Real-Life Analogy:**

  Think of a capsule (medicine). You can use it, but you can't see or modify what's inside — this is **encapsulation**.

Java Program Demonstrating Encapsulation:

```java
// Class with encapsulated fields
class Student {
    // Private data members
    private String name;
    private int age;

    // Public setter method for name
    public void setName(String newName) {
        name = newName;
    }

    // Public getter method for name
    public String getName() {
        return name;
    }

    // Public setter method for age
    public void setAge(int newAge) {
        if (newAge > 0) {
            age = newAge;
        }
    }

    // Public getter method for age
    public int getAge() {
        return age;
    }
}

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

        // Setting values using setter methods
        s.setName("Diksha");
        s.setAge(20);

        // Getting values using getter methods
        System.out.println("Name: " + s.getName());
        System.out.println("Age: " + s.getAge());
    }
}
```

input

```
Name: Diksha
Age: 20


...Program finished with exit code 0
Press ENTER to exit console.
```

8. What is abstraction in Java? How is it achieved?

What is Abstraction in Java?

Definition:

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

It helps reduce complexity by letting you focus on **what an object does**, instead of **how it does it**.

Example in Real Life:

When you drive a car, you use the **steering wheel, accelerator, brake** – you don't know the internal mechanics.
That's abstraction — you interact with only the **necessary parts**.

How Abstraction is Achieved in Java:

Java provides two main ways:

1. **Abstract Classes**
2. **Interfaces**

1. Abstract Class

- A class declared with the abstract keyword.
- Can have both **abstract methods** (without body) and **regular methods**.
- Cannot be instantiated directly.

Example:

Main.java

```java
// Abstract class
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Concrete method
    void eat() {
        System.out.println("Animal eats food");
    }
}

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

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound();  // Calls overridden method
        myDog.eat();    // Calls inherited method
    }
}
```

input

```
Dog barks
Animal eats food


...Program finished with exit code 0
Press ENTER to exit console.
```

2. Interface

- Contains only **abstract methods** (by default) and **constants**.
- A class implements the interface and provides method definitions.

- Supports **multiple inheritance**.

Example:

Main.java

```java
// Interface
interface Vehicle {
    void start();
}

// Class implementing the interface
class Car implements Vehicle {
    public void start() {
        System.out.println("Car starts with key");
    }
}

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Create object
        myCar.start();          // Call start method
    }
}
```

input

```
Car starts with key


...Program finished with exit code 0
Press ENTER to exit console.
```

8. Explain the difference between abstract class and interface.

Abstract Class

- Can have both abstract and regular (concrete) methods.
- Can have constructors.
- Can have variables (fields) with any access modifier.
- Supports single inheritance only.
- Used when you want to share common behavior among subclasses. • **Syntax**

Example

abstract class Animal { abstract void

sound(); // abstract method void eat() {

// concrete method

System.out.println("Animal eats food");

}

}

Interface

- Contains only abstract methods (until Java 7), and only public methods.
- Cannot have constructors.
- Variables are always public, static, and final.
- Supports multiple inheritance (a class can implement multiple interfaces).
- Used when you want to define a set of rules or a contract. Syntax Example interface

  Vehicle { void start(); // abstract method

}

9. Create a Java program to demonstrate the use of interface

Main.java

```java
// Define an interface
interface Animal {
    void sound();   // abstract method
    void sleep();   // another abstract method
}

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

    public void sleep() {
        System.out.println("Dog sleeps in kennel");
    }
}

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();   // Create object
        myDog.sound();           // Call method from interface
        myDog.sleep();           // Call another method from interface
    }
}
```

input

```
Dog barks
Dog sleeps in kennel


...Program finished with exit code 0
Press ENTER to exit console.
```

Explanation:

- The Animal interface defines **two abstract methods**: sound() and sleep().

- The Dog class **implements** the interface and provides concrete definitions for both methods.
- In the main() method, we create a Dog object and call those methods to demonstrate the use of the interface.

10.What is method overloading and method overriding? Show with examples.

1. Method Overloading

Definition:

Method Overloading means **defining multiple methods with the same name** in the **same class**, but with **different parameters** (number, type, or order).

It is an example of **compile-time polymorphism**.

Main.java

```java
class Calculator {
    // Method with 2 int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    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("Add 2 ints: " + calc.add(10, 20));
        System.out.println("Add 3 ints: " + calc.add(5, 10, 15));
        System.out.println("Add 2 doubles: " + calc.add(2.5, 3.5));
    }
}
```

input

```
Add 2 ints: 30
Add 3 ints: 30
Add 2 doubles: 6.0


...Program finished with exit code 0
Press ENTER to exit console.
```

2. Method Overriding

Definition:

Method Overriding means a **subclass** provides a specific implementation of a method that is **already defined in its superclass**, with the **same name and parameters**.

It is an example of **runtime polymorphism**.

Example:

Main.java

```java
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 Dog();  // Upcasting
        a.sound();             // Calls overridden method in Dog
    }
}
```

input

```
Dog barks


...Program finished with exit code 0
Press ENTER to exit console.
```