JAVA ASSIGNMENT

Name: Ragini Dattatray Telange
Batch Code: ANP-D1544

Student Code:AF04953320

Java Basics & OOPs Assignment Questions

1. Java Basics

1. What is Java? Explain its features.

   → **Java** is a high-level, object-oriented programming language developed by Sun Microsystems. It is used to build software for various platforms and is known for being simple, secure, and platform-independent.

   - **Simple** – Easy to learn and write.
   - **Object-Oriented** – Based on objects and classes.
   - **Platform Independent** – Runs on any OS using JVM.
   - **Secure** – Provides safety from viruses and threats.
   - **Robust** – Handles errors and manages memory well.
   - **Multithreaded** – Can do many tasks at once.
   - **Portable** – Code runs on any system.
   - **High Performance** – Faster than traditional languages.
   - **Distributed** – Supports network-based applications.
   - **Dynamic** – Loads code and classes at runtime.

---

2. Explain the Java program execution process.

   → **Write Code** – You write a .java file using a text editor.
   Compile – The Java compiler (javac) converts code to **bytecode** (.class file).
   Load – The bytecode is loaded into the **Java Virtual Machine (JVM)**.
   Verify – JVM checks the bytecode for security and correctness.
   Execute – JVM uses the **Interpreter** or **JIT compiler** to run the program.

   Source Code → Bytecode → JVM → Output

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 are used to define the type of data a variable can hold. They help the compiler understand how much memory is needed and what operations can be performed on the data.

Java data types are divided into two main categories:

---

1. Primitive Data Types

There are 8 primitive data types in Java:

Data Type	Description	Example
int	Stores whole numbers	int a = 10;
float	Stores decimal numbers	float b = 10.5f;
double	Stores large decimal values	double d = 20.55;
char	Stores a single character	char c = 'A';
boolean	Stores true or false	boolean flag = true;
byte	Stores small numbers (1 byte)	byte x = 100;
short	Stores short integers	short y = 1000;
long	Stores large integers	long l = 123456L;

2. Non-Primitive Data Types

These are also called reference data types. They refer to objects and include:

- String – A sequence of characters (e.g., "Hello")
- Array – A collection of values
- Class – A blueprint for objects
- Interface – A collection of abstract methods

5. What is the difference between JDK, JRE, and JVM?

→

1. JVM (Java Virtual Machine):

- It is the **engine** that runs Java bytecode.
- It makes Java **platform-independent**.
- **Responsibilities:** Loads, verifies, and executes the program.

Example:

Bytecode (.class file) runs on JVM.

2. JRE (Java Runtime Environment):

- It provides the **libraries + JVM** required to run Java programs.
- It does **not** include development tools (like compiler).

Use:

Install JRE if you only want to **run** Java programs.

3. JDK (Java Development Kit):

- It includes everything: **JRE + Development tools** (like javac).
- Used to **write, compile, and run** Java programs.

Use:

Install JDK if you want to **develop** Java programs.

---

6. What are variables in Java? Explain with examples.

→

In Java, a **variable** is a name given to a memory location used to store data. Variables hold values that can be used and changed during program execution.

Types of Variables in Java:

1. **Local Variable:**
    - Declared inside a method.
    - Accessible only within that method.
    - Example:
    -

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

2. **Instance Variable:**
    - Declared inside a class but outside any method.
    - Each object has its own copy.
    - Example:

-

```
class Student {
  int marks = 85; // instance variable
}
```

3. **Static Variable:**
   - Declared with the static keyword.
   - Shared by all objects of the class.
   - Example:

```
class Student {
  static String college = "MIT"; // static variable
}
```

Example Program:

```
public class Example {
  static String college = "MIT"; // static variable
  int age = 20; // instance variable

  public void show() {
    int roll = 101; // local variable
    System.out.println("Roll: " + roll);
    System.out.println("Age: " + age);
    System.out.println("College: " + college);
  }
}
```

---

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

→

Types of Operators:

1. **Arithmetic Operators** – +, -, *, /, %
   (Used for mathematical operations)
2. **Relational (Comparison) Operators** – ==, !=, >, <, >=, <=
   (Used to compare values)
3. **Logical Operators** – &&, ||, !
   (Used with boolean values)
4. **Assignment Operators** – =, +=, -=, *=, /=
   (Used to assign values)
5. **Unary Operators** – +, -, ++, --, !
   (Used with a single operand)
6. **Bitwise Operators** – &, |, ^, ~, <<, >>
   (Used for bit-level operations)

7. **Ternary Operator** – ? :
   (Used for short if-else conditions)

   Example:

   ```
   int a = 10, b = 5;
   System.out.println(a + b); // Arithmetic
   System.out.println(a > b); // Relational
   ```

---

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

   Control statements in Java are used to control the flow of execution based on conditions.

   1. if Statement

   - Executes a block if the condition is true.

```
if (a > b) {
   System.out.println("A is greater");
}
```

   2. if-else Statement

   - Executes one block if the condition is true, otherwise the else block.

```
if (a > b) {
   System.out.println("A is greater");
} else {
   System.out.println("B is greater");
}
```

   3. switch Statement

   - Selects one block to execute from many options based on value.

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

---

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

```
public class EvenOdd {
   public static void main(String[] args) {
      int num = 7; // you can change this number

      if (num % 2 == 0) {
         System.out.println(num + " is Even");
      } else {
         System.out.println(num + " is Odd");
      }
   }
}
```

---

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

while Loop	**do-while Loop**
Condition is checked **before** loop runs	Condition is checked **after** loop runs
May **not execute** even once	Executes **at least once**
Syntax:	Syntax:

```
while(condition) {
  // code
}
``` |
```java
do {
  // code
} while(condition);
``` |
Example:

```
int i = 5;
while (i < 5) {
   System.out.println("While loop"); // won't run
}

do {
   System.out.println("Do-while loop"); // runs once
   i++;
} while (i < 5);
```

2. Object-Oriented Programming (OOPs)

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

→
OOPs stands for **Object-Oriented Programming System**. Java follows four main OOPs principles:

1. Encapsulation

- Wrapping data and methods into a single unit (class).
- Data is hidden using private and accessed through get/set methods.

Example:

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

2. Inheritance

- One class inherits features of another class using extends keyword.
- Promotes code reuse.

Example:

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

3. Polymorphism

- One task can be done in many ways (method overloading/overriding).

Example:

```
class Demo {
   void show(int a) { }
   void show(String b) { }  // Method Overloading
}
```

4. Abstraction

- Hiding complex details and showing only essential info.
-
- Achieved using abstract class or interface.

Example:

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

---

12. What is a class and an object in Java? Give examples.

→

Class:

- A **class** is a blueprint or template used to create objects.
- It defines variables and methods.

Example:

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

Object:

- An **object** is an instance of a class.
- It is used to access class members (variables and methods).

Example:

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

---

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

→

```
// Class to calculate area of a rectangle
```

```
class Rectangle {
   int length;
   int width;

   void setData(int l, int w) {
      length = l;
      width = w;
   }

   void calculateArea() {
      int area = length * width;
      System.out.println("Area of Rectangle: " + area);
   }
}

public class Main {
   public static void main(String[] args) {
      Rectangle rect = new Rectangle(); // Create object
      rect.setData(10, 5);              // Set values
      rect.calculateArea();             // Call method
   }
}
```

---

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

→

Inheritance is one of the main principles of OOP in Java. It allows one class (child) to **inherit properties and methods** from another class (parent).

Real-Life Example:

- **Parent Class**: Vehicle
- **Child Class**: Car

A **Car** is a type of **Vehicle**. It inherits common features like engine, wheels, etc., from Vehicle and can also have its own features.

Java Example:

```
// Parent class
class Vehicle {
   void run() {
      System.out.println("Vehicle is running");
```

```
  }
}

// Child class
class Car extends Vehicle {
  void features() {
    System.out.println("Car has AC and Music System");
  }
}

// Main class
public class Main {
  public static void main(String[] args) {
    Car myCar = new Car();  // Creating object of child class
    myCar.run();          // Inherited method
    myCar.features();     // Own method
  }
}
```

Output:

```
Vehicle is running
Car has AC and Music System
```

---

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

→

Polymorphism means **"many forms"**. In Java, it allows objects to take on more than one form depending on the context. It enables one interface to be used for a general class of actions. The two main types are:

- **1. Compile-Time Polymorphism (Static Polymorphism):**

This is achieved **through method overloading**. The method to be executed is determined at **compile time**.

Example – Method Overloading:

```
class MathOperation {
  // 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) {
      MathOperation mo = new MathOperation();
System.out.println("Sum of 2 numbers: " + mo.add(5, 10));
      System.out.println("Sum of 3 numbers: " + mo.add(5, 10, 15));
   }
}
```

Output:

```
Sum of 2 numbers: 15
Sum of 3 numbers: 30
```

2. Runtime Polymorphism (Dynamic Polymorphism):

This is achieved **through method overriding**. The method that gets executed is determined at **runtime** using **inheritance and upcasting**.

Example – 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");
   }
}

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

public class Main {
   public static void main(String[] args) {
      Animal a1 = new Dog(); // Upcasting
      Animal a2 = new Cat();
```

```
        a1.sound();  // Calls Dog's sound()
        a2.sound();  // Calls Cat's sound()
    }
}
```

- **Output:**

```
Dog barks
Cat meows
```

---

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

→

Method overloading means **defining multiple methods** in the same class **with the same name but different parameters** (type, number, or order). It is resolved **at compile time**.

Example:

```
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("Sum of 2 ints: " + calc.add(5, 10));
        System.out.println("Sum of 3 ints: " + calc.add(5, 10, 15));
        System.out.println("Sum of 2 doubles: " + calc.add(3.5, 2.5));
    }
}
```

Output:

Sum of 2 ints: 15
Sum of 3 ints: 30
Sum of 2 doubles: 6.0

Method Overriding (Runtime Polymorphism)

Definition:

Method overriding occurs when a **subclass provides its own implementation** of a method that is **already defined in its superclass**. It is resolved **at runtime**.

Example:

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

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

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

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

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

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

Output:

Dog barks
Cat meows

---

17. What is encapsulation? Write a program demonstrating encapsulation
→

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It means **binding data (variables)** and the **code (methods)** that operates on the data into a **single unit (class)** and **restricting direct access** to some of the object's components.

In Java, encapsulation is achieved using:

private access modifiers for data members (variables)
public getter and setter methods to access and update data safely

Benefits of Encapsulation

Protects data from unauthorized access
Improves code maintainability and flexibility
Allows data hiding
Enables control over data (validation logic in setters)

Java Program Demonstrating Encapsulation

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

  // Getter for name
  public String getName() {
    return name;
  }

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

  // Getter for age
  public int getAge() {
```

```
        return age;
    }

    // Setter for age with validation
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Age must be positive.");
        }
    }
}

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

        // Setting values using setters
        s.setName("Ragini");
        s.setAge(20);

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

Output:

Student Name: Ragini
Student Age: 20

---

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

→

Abstraction is an **Object-Oriented Programming (OOP)** principle that focuses on **hiding the internal implementation details** and **showing only the essential features** of an object.

In simple terms:

It lets you use **what an object does**, not **how it does it**.

Why use Abstraction?

To reduce complexity

To increase code reusability
To focus on **"what to do"** rather than **"how to do"**
To implement security — hide sensitive logic from the user

How is Abstraction Achieved in Java?

In Java, abstraction is achieved in two ways:

Method	Description
Abstract class	A class declared with the abstract keyword. Can have abstract and concrete methods.
Interface	A blueprint of a class. All methods are abstract by default (Java 7) or can include default methods (Java 8+).

1. Using Abstract Class

```
abstract class Shape {
   // Abstract method (no body)
   abstract void draw();

   // Concrete method
   void show() {
      System.out.println("This is a shape.");
   }
}

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

public class Main {
   public static void main(String[] args) {
      Shape s = new Circle();  // Upcasting
      s.draw();   // Calls overridden method
      s.show();   // Calls base class method
   }
}
```

Output

```
Drawing a Circle
This is a shape.
```

2. Using Interface

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

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

public class Main {
   public static void main(String[] args) {
      Animal a = new Dog();
      a.sound();
   }
}
```

Output:

Dog barks

---

19. Explain the difference between abstract class nd interface.

→

Abstract Class vs Interface in Java

Feature	**Abstract Class**	**Interface**
Keyword used	abstract	interface
Method types	Can have both **abstract** and **concrete (defined)** methods	Can have **abstract**, default, static, and private methods (Java 8+)
Variables	Can have instance variables (with any access modifier)	Only public static final (i.e., constants)
Constructor	Can have a constructor	Cannot have a constructor
Inheritance	A class **extends** an abstract class	A class **implements** an interface
Multiple Inheritance	Not supported (only single inheritance)	Supported (a class can implement multiple interfaces)

Feature	Abstract Class	Interface
Access Modifiers for methods	Can be public, protected, or private	Methods are public by default
When to use	When you want to provide **partial abstraction** and code reuse	When you want to provide **full abstraction** and define only method signatures
Speed	Slightly faster as it has implemented methods	Slightly slower due to extra abstraction layer

Example of Abstract Class:

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

   void sleep() {
      System.out.println("Animal is sleeping");
   }
}

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

Example of Interface:

```
interface Animal {
   void sound();  // implicitly public and abstract
}

class Cat implements Animal {
   public void sound() {
      System.out.println("Cat meows");
   }
}
```

---

20. Create a Java program to demonstrate the use of interface.
→

Java Program to Demonstrate Interface

```
// Define the interface
interface Vehicle {
   // Abstract method
   void start();

   // Default method (Java 8+ feature)
   default void stop() {
      System.out.println("Vehicle stopped.");
   }
}

// Implementing class 1
class Car implements Vehicle {
   public void start() {
      System.out.println("Car started with key.");
   }
}

// Implementing class 2
class Bike implements Vehicle {
   public void start() {
      System.out.println("Bike started with kick.");
   }
}

// Main class
public class Main {
   public static void main(String[] args) {
      // Using interface reference
      Vehicle v1 = new Car();
      Vehicle v2 = new Bike();

      // Call methods
      v1.start();   // Car-specific implementation
      v1.stop();    // Common method from interface

      v2.start();   // Bike-specific implementation
```

```
        v2.stop();    // Common method from interface
    }
}
```

- **Output:**

```
Car started with key.
Vehicle stopped.
Bike started with kick.
Vehicle stopped.
```

Explanation:

Vehicle is an **interface** that defines a contract for start() and stop() behavior.
Car and Bike **implement** this interface and provide their own version of the start() method.
The stop() method is a **default method** in the interface — shared by all classes.
