Name: Harsh Hande.

1. Java Basics

1. What is Java? Explain its features.

Java is a high-level, object-oriented programming language that was developed by Sun Microsystems (now owned by Oracle). It's designed to be platform-independent, meaning that code written in Java can run on any device that supports the Java Virtual Machine (JVM).

Key Features of Java :-

1. Object-Oriented : Everything in Java is treated as an object, allowing for modular programs and reusable code.

2. Platform-Independent : Java code is compiled into bytecode, which can run on any platform with a JVM.

3. Simple and Familiar : Java's syntax is clean and easy to understand, especially for those familiar with C or C++.

4. Secure : Java has built-in security features like bytecode verification, sandboxing, and runtime security checks.

5. Robust : It handles errors gracefully with strong memory management and exception handling

. 2 .Explain the Java Program Execution Process

The execution process of a Java program involves the following steps:

1. **Writing the Code :- The Java program is written using a text editor or IDE and saved with a .java extension.**
2. **Compilation :- The source code is compiled using the Java Compiler (javac). This compiler translates the .java file into bytecode, which is saved as a**
   .class file.
3. **Class Loading :- The ClassLoader loads the compiled .class files into memory during runtime.**

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

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

4. What are data types in Java? List and explain them

Primitive	**Non-Primitive**
Stores simple values	Refers to objects
Fixed memory size	Memory depends on object
Cannot call methods	Can call methods
Not null	Can be null

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

JDK (Java Development Kit) is a complete software development package required for developing Java applications. It includes tools like the compiler (javac), debugger, and other utilities. The JDK also contains the JRE (Java Runtime Environment), so it can both develop and run Java programs.

JRE (Java Runtime Environment) provides the libraries, Java Virtual Machine (JVM), and other components needed to run Java applications.

However, it does not include development tools like a compiler or debugger, so you can't write or compile code with just the JRE.

JVM (Java Virtual Machine) is the core part of both JDK and JRE. It is responsible for executing the Java bytecode, which is produced after the source code is compiled. The JVM makes Java platform-independent by allowing the same bytecode to run on any device that has a compatible JVM.

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

1. **Arithmetic Operators**

Used for basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

---

2. Relational (Comparison) Operators Used to compare two values.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b

<=	Less than or equal to	a <= b

---

3. Logical Operators

Used to combine multiple boolean expressions.

Operator	Description	Example
&&	Logical AND	a > 10 && b < 20
`		`
!	Logical NOT	!(a > b)

---

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	a = 5
+=	Add and assign	a += 2
-=	Subtract and assign	a -= 3
*=	Multiply and assign	a *= 4
/=	Divide and assign	a /= 2
%=	Modulus and assign	`a %=

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

1. if Statement Syntax:

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

Example: int
age = 18; if
(age >= 18) {

```
   System.out.println("You are eligible to vote.");
}
```

---

2. if-else Statement

The if-else statement provides an alternative. If the condition is true, one block runs; otherwise, the else block runs. if (condition) {

```
   // code if condition is true
} else {
   // code if condition is false
}
```

Example:

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

---

3. switch Statement

The switch statement is used to choose one out of many blocks of code to be executed.

Syntax: switch

```
(expression) {
case value1:        //
code        break;
case value2:        //
code        break;
default:
      // code
}
```

Example:

```
int day = 2;
switch
```

```
(day) {
case 1:
       System.out.println("Monday");
break;     case 2:
       System.out.println("Tuesday");
break;
   default:
```

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 = 4;
       if (num % 2 ==
0)

System.out.println(
"Even");
       else

System.out.println(
"Odd");
   }
}
```

Sample Output

```
Enter a number: 7
7 is Odd.

Enter a number: 12

12 is Even.
```

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

while Loop

- **The condition is checked first.**
- **If the condition is true, the loop body executes.**
- **If the condition is false at the start, the loop does not execute at all.**

Syntax:

```
while (condition) {
   // code to execute
}
```

Example:

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

---

do-while Loop

- **The loop body is executed at least once, before the condition is checked.**
- **After the first execution, it continues as long as the condition is true.**

Syntax: do

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

Example:

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

• [Object-Oriented Programming (OOPs)]

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

The main principles of OOPs in Java are encapsulation, inheritance, polymorphism, and abstraction. These principles help in designing programs that are modular, reusable, and easier to maintain.

- **Encapsulation: Binding data and methods together, controlling access.**
- **Inheritance: One class inherits from another, enabling reuse.**
- **Polymorphism: Same method or object behaves differently in different contexts.**
- **Abstraction: Hiding details, showing only necessary parts.**

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

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 (functions)
}
```

Example:

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

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

Object in Java:

An object is an instance of a class. When a class is defined, no memory is allocated. When we create an object of the class using the new keyword, memory is allocated and methods/variables can be used.

Syntax:

ClassName obj = new ClassName(); Example:

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

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

```
class Rectangle {
   int length, breadth;

   Rectangle(int l, int b) {
      length = l;
      breadth = b;
   }

   int area() {
      return length * breadth;
   }

   public static void main(String[] args) {
      Rectangle r = new Rectangle(5, 3);
      System.out.println("Area: " + r.area());
   }
}
```

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

Inheritance is one of the main principles of Object-Oriented Programming (OOP) in Java. It allows one class (called the child or subclass) to inherit the properties and methods of another class (called the parent or superclass). This promotes code reusability and a hierarchical classification.

Real-life Example of Inheritance:

Example:
A Car is a type of Vehicle.
All Vehicles have common properties like speed, color, and methods like start() or stop().
But Car may have some extra features like air conditioning or music system.

Java Code Example:

```
class Vehicle {
   String brand = "Honda";
 void start() {
      System.out.println("Vehicle is starting...");
   }
}
class Car extends Vehicle {
String model = "City";   void
playMusic() {
      System.out.println("Playing music...");
   }
}

public class InheritanceExample {
public static void main(String[] args) {
      Car myCar = new Car();
 System.out.println("Brand: " + myCar.brand);
      myCar.start();
 System.out.println("Model: " + myCar.model);
      myCar.playMusic();
   }
}
```

Output:

```
Brand: Honda
```

```
Vehicle is starting...

Model: City

Playing music...
```

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

Polymorphism means "many forms". It allows one interface or method to behave differently based on the context.
In Java, polymorphism is mainly of two types:

Types of Polymorphism:

Type	Also Known As	How it Works
Compile-time	Method Overloading	Same method name with different parameters
Runtime	Method Overriding	Subclass provides specific implementation

---

1. Compile-time Polymorphism (Method Overloading)

Example:

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

```
public class CompileTimeExample {

    public static void main(String[] args) {

        Calculator calc = new Calculator();

        System.out.println("Sum (2 values): " + calc.add(10, 20));

        System.out.println("Sum (3 values): " + calc.add(5, 10, 15));

    }

}
```

Output:

```
Sum (2 values): 30

Sum (3 values): 30
```

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

Both are OOP features in Java that support polymorphism, but they are used in different ways.

Method Overloading (Compile-time Polymorphism)

Definition:
Method overloading means having multiple methods with the same name but different parameters (type, number, or order) in the same class.

Example:

```
class Calculator {    int

add(int a, int b) {

return a + b;

   }

double add(double a, double b) {

       return a + b;

   }

 int add(int a, int b, int c) {

return a + b + c;

   }

}


public class OverloadingExample {

   public static void main(String[] args) {

       Calculator c = new Calculator();

       System.out.println("Add 2 ints: " + c.add(10, 20));
```

```
        System.out.println("Add 2 doubles: " + c.add(5.5, 4.5));
System.out.println("Add 3 ints: " + c.add(1, 2, 3));

    }

}
```

Output:

```
Add 2 ints: 30

Add 2 doubles: 10.0

Add 3 ints: 6
```

Method Overriding (Runtime Polymorphism)

Definition:
Method overriding means a subclass provides a specific implementation of a method that is already defined in its parent class.

Example: class

```
Animal {     void

sound() {

        System.out.println("Animal makes sound");

    }

}


class Cat extends Animal {

    @Override

void sound() {

        System.out.println("Cat meows");

    }

}
```

```
public class OverridingExample {
public static void main(String[] args) {
Animal a = new Cat(); // Upcasting
      a.sound();  // Calls overridden method from Cat class
   }
}
```

Output: nginx

CopyEdit

```
Cat meows
```

7. 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 singleunit (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)

Real-life Example:

A bank account should not allow direct access to its balance from outside the class.
Instead, access is given through controlled methods (getBalance, deposit, withdraw).

```
class BankAccount {     private double balance;  public BankAccount(double

initialBalance) {        balance =

initialBalance;

    }

 public double getBalance() {

return balance;

    }

  public void deposit(double amount) {

        if (amount > 0) {

balance += amount;

        } else {

            System.out.println("Invalid deposit amount");

        }

    }

    public void withdraw(double

amount) {        if (amount > 0 &&

amount <= balance) {            balance -=

amount;

        } else {

            System.out.println("Insufficient balance or invalid amount");
        }
    }
}

public class EncapsulationExample {
public static void main(String[] args)
{
        BankAccount account = new BankAccount(1000);  // Initial balance
System.out.println("Initial Balance: ₹" + account.getBalance());
account.deposit(500);
        System.out.println("After Deposit: ₹" + account.getBalance());
account.withdraw(300);
        System.out.println("After Withdrawal: ₹" + account.getBalance());
account.withdraw(1500); // Invalid
    }
}
```

Output:

```
Initial Balance: ₹1000.0
After Deposit: ₹1500.0
After Withdrawal: ₹1200.0
Insufficient balance or invalid amount
```

8. 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

---

1. Using Abstract Class

```
abstract class Animal {

abstract void sound();

void eat() {

      System.out.println("Animal is eating...");

   }

}

class Dog extends Animal {

void sound() {

      System.out.println("Dog barks");

   }

}


public class AbstractClassExample {

public static void main(String[] args)

{       Animal a = new Dog();

      a.sound();
```

```
        a.eat();
    }
}Output:
 Dog barks
 Animal is eating...
```

---

2. Using Interface interface Shape

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

public class InterfaceExample {
public static void main(String[]
args) {        Shape s = new Circle();
        s.draw();
    }
}
```

Output:

```
Drawing a circle
```

9. Explain the difference between abstract class and interface.

An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (without body) as well as concrete methods (with body).

Example:

```
abstract class Animal {

abstract void makeSound();

void eat() {

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

   }

}class Dog extends Animal {

void makeSound() {

      System.out.println("Dog barks");

   }

}
```

Interface:

An interface is a completely abstract type used to define only method signatures and constants. From Java 8 onward, interfaces can have default and static methods with body.

Example:

```
interface Animal {    void makeSound(); //

implicitly public and abstract

}
```

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

   }
}
```

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

Java Program: Interface Example — Payment System

```
interface Payment {     void pay(int amount);
}
class CreditCard implements Payment {
public void pay(int amount) {
      System.out.println("Paid ₹" + amount + " using Credit Card.");
   }}
class PayPal implements Payment {
public void pay(int amount) {
      System.out.println("Paid ₹" + amount + " using PayPal.");
   }}
public class PaymentDemo {
public static void main(String[]
args) {        Payment
paymentMethod;
paymentMethod = new
CreditCard();

paymentMethod.pay(2550);

paymentMethod = new PayPal();

paymentMethod.pay(63450);

   }}
```

Output:

```
Paid ₹2550 using Credit Card.

Paid ₹63450 using PayPal.
```