OOP-JAVA
(Question Bank)

1) Discuss the characteristics of Object Oriented Programming.

- Object-Oriented Programming (OOP) is characterized by:

  - **Class:** A blueprint or template that defines the structure and behavior (attributes and methods) of objects. Classes encapsulate data and functions that operate on the data.

  - **Object:** An instance of a class. Objects represent real-world entities and have a state (attributes) and behavior (methods). Each object is created based on a class and operates independently.

  - **Encapsulation:** Bundling data and methods together within a class while restricting access to the object's internals. This promotes modularity and data protection.

  - **Abstraction:** Hiding complex implementation details and exposing only what's necessary through a clear interface. This simplifies interaction with objects.

  - **Inheritance:** Allowing new classes to inherit properties and behaviors from existing ones. This promotes code reuse and hierarchical organization.

  - **Polymorphism:** Enabling objects to be treated as instances of their parent class, allowing for flexibility in method implementation. This supports dynamic method binding and code generalization.

2) Write a note on Java Virtual Machine.

- The Java Virtual Machine (JVM) is a key component of Java's platform-independent architecture.
- It acts as an abstract computing machine that enables Java programs to be executed on any device or operating system without needing to be rewritten for each one.
- The JVM takes Java bytecode—the intermediate representation of compiled Java code—and interprets or compiles it into machine code specific to the host system.

- Key responsibilities of the JVM include:

  - Bytecode execution: Running compiled Java programs.
  - Memory management: Handling allocation and garbage collection of objects.
  - Security: Enforcing Java's sandbox security model by isolating program execution.
  - Platform independence: Allowing Java applications to run on any operating system without modification.

- The JVM is integral to Java's "write once, run anywhere" promise, making it a versatile tool for developers.

3) JVM is platform dependent. Justify.

- While the Java Virtual Machine (JVM) allows Java programs to be platform-independent at the application level, the JVM itself is platform-dependent. This means that the JVM must be specifically implemented and compiled for each operating system and hardware architecture it runs on.

- Reasons for JVM's Platform Dependence:

  - **Native Code Generation:** The JVM translates Java bytecode into native machine code, which varies across different hardware and operating systems.

- **Integration with OS Resources:** The JVM interacts with OS resources like memory and files, requiring platform-specific implementations to manage these effectively.

- **Optimization for Performance:** JVM implementations are tailored to leverage the unique features of different platforms, enhancing performance.

4) How is for-each loop is used in Java? Illustrate with a suitable example.

- The for-each loop in Java, also known as the enhanced for loop, is used to iterate over elements in arrays or collections like ArrayList, HashSet, etc.
- It simplifies the iteration process by eliminating the need for an explicit iterator or index counter.
- Syntax :

```
for (Type variable : collection) {
    // Code to be executed for each element
}
```

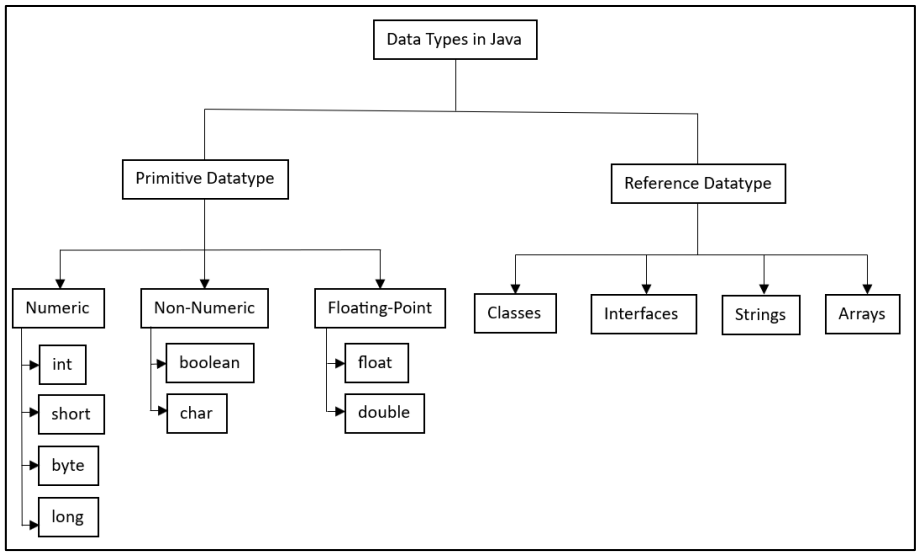
  - Type: The type of elements in the collection or array.
  - variable: A temporary variable that holds the current element during each iteration.
  - collection: The array or collection you want to iterate over.

- Example :

```
public class ForEachExample {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};

        // Using for-each loop to iterate over the array
        for (int num : numbers) {
            System.out.println("Number: " + num);
        }
    }
}
```

5) Explain Data Types in detail with example.



OOP-JAVA
(Question Bank)

- Java provides two categories of data types:

  - **Primitive Data Types :**

    1. byte
       - Used for small numbers
       - Size : 1 byte
       - Example : **byte n = 25;**

    2. Short
       - Larger than “byte” but still smaller than “int”.
       - Size : 2 byte
       - Example : **short n = 15000;**

    3. Int
       - The most commonly used integer type.
       - Size : 4 byte
       - Example : **int n = 1000000;**

    4. Long
       - Used when a wider range than “int” is needed.
       - Size : 8 byte
       - Example : **long n = 149600000000L;**

    5. Float
       - Used for decimal and precise values.
       - Size : 4 byte
       - Example : **float n = 36.6f;**

    6. Double
       - Default choice for decimals, representing a double-precision floating point.
       - Size : 8 byte
       - Example : **double n = 3.14159;**

    7. Char
       - Used for storing single characters, represented in Unicode.
       - Size : 2 byte
       - Example : **char n = 'A';**

    8. Boolean
       - Used for true/false values, often in control statements.
       - Size : 1 byte
       - Example : **boolean n = true;**

  - **Reference/Object Data Types**

    1. String
       - Represents a sequence of characters.
       - Example : **String greeting = "Hello, World!";**

2. Arrays
   - A container that holds a fixed number of values of a single type.
   - Example : **int[] numbers = {10, 20, 30, 40, 50};**

3. Class
   - Blueprint for creating objects.
   - Example :

```
class Product {
   int id;
   String name;
   double price;
}
Product product = new Product();
```

4. Interfaces
   - Interfaces define methods that a class must implement. They are used to achieve abstraction and multiple inheritance in Java.
   - Example :

```
interface Animal {
   void eat();
}
```

6) Explain any three methods of StringBuffer class with appropriate example.

- The StringBuffer class in Java is used to create mutable (modifiable) strings.
- It provides several methods to manipulate the content of the string. Here are three commonly used methods of the StringBuffer class with examples:
  - **append():** adds data to the end of the StringBuffer.

```
public class StringBufferExample {
   public static void main(String[] args) {
      StringBuffer sb = new StringBuffer("Hello");
      sb.append(" World!");
      System.out.println(sb);
   }
}
```

  - **insert():** adds data at a specified position within the StringBuffer.

```
public class StringBufferExample {
   public static void main(String[] args) {
      StringBuffer sb = new StringBuffer("Hello");
      sb.insert(5, " World"); // Inserts " World" at index 5
      System.out.println(sb); // Output: Hello World
   }
}
```

- **reverse()**: reverses the entire content of the StringBuffer.

```
public class StringBufferExample {
   public static void main(String[] args) {
      StringBuffer sb = new StringBuffer("Hello");
      sb.insert(5, " World"); // Inserts " World" at index 5
      System.out.println(sb); // Output: Hello World
   }
}
```

7) Create a two dimensional array. Instantiate and Initialize it.

- In Java, a two-dimensional array is an array of arrays, where each element of the primary array is an array itself.
- Example :

```
public class TwoDimensionalArrayExample {
   public static void main(String[] args) {
      // Creating and instantiating a 2D array with 3 rows and 3 columns
      int[][] matrix = new int[3][3];

      // Initializing the 2D array
      matrix[0][0] = 1;
      matrix[0][1] = 2;
      matrix[0][2] = 3;
      matrix[1][0] = 4;
      matrix[1][1] = 5;
      matrix[1][2] = 6;
      matrix[2][0] = 7;
      matrix[2][1] = 8;
      matrix[2][2] = 9;

      // Alternatively, you can initialize the 2D array at the time of declaration
      // int[][] matrix = {
      //    {1, 2, 3},
      //    {4, 5, 6},
      //    {7, 8, 9}
      // };

      // Printing the 2D array
      for (int i = 0; i < 3; i++) {
         for (int j = 0; j < 3; j++) {
            System.out.print(matrix[i][j] + " ");
         }
         System.out.println(); // Move to the next line after each row
      }
   }
}
```

8) Differentiate String class and StringBuffer class.

- In Java, the String class and the StringBuffer class are both used to handle sequences of characters, but they differ in terms of mutability, performance, and thread-safety.

Feature	String	StringBuffer
Mutability	Immutable	Mutable
Perfomance	Slower for frequent changes	Faster for frequent modifications
Thread-Safety	Thread-safe (due to immutability)	Thread-safe (methods are synchronized)
Memory Uage	Higher (creates new objects)	Lower (modifies the same object)
Use	For constant strings	For strings requiring frequent modifications, in a thread-safe environment

```
// String (Immutable)
String str = "Hello";
str = str + " World";  // Creates a new String object

// StringBuffer (Mutable)
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");  // Modifies the existing object
```

9) What is a Jagged Array in Java ? How is a Jagged array declared and defined in Java.

- A Jagged Array in Java is a multi-dimensional array where the rows can have different lengths.
- Unlike a regular two-dimensional array where each row has the same number of columns, a jagged array allows each row to be of varying sizes.
- This is useful when you need a table-like structure but with rows of different lengths.
- Example : Declaring, Defining, and Initializing a Jagged Array

```
public class JaggedArrayExample {
    public static void main(String[] args) {
        // Declare and instantiate a jagged array with 3 rows
        int[][] jaggedArray = new int[3][];

        // Define the number of columns in each row
        jaggedArray[0] = new int[3]; // First row with 3 elements
        jaggedArray[1] = new int[2]; // Second row with 2 elements
        jaggedArray[2] = new int[4]; // Third row with 4 elements

        // Initialize the jagged array with values
        jaggedArray[0][0] = 1;
        jaggedArray[0][1] = 2;
        jaggedArray[0][2] = 3;

        jaggedArray[1][0] = 4;
        jaggedArray[1][1] = 5;

        jaggedArray[2][0] = 6;
        jaggedArray[2][1] = 7;
        jaggedArray[2][2] = 8;
```

```
        jaggedArray[2][3] = 9;

        // Printing the jagged array
        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
                System.out.print(jaggedArray[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

10) Define following. 1) Byte code 2) Java Virtual Machine

1) Bytecode

- Bytecode is an intermediate, platform-independent code generated by the Java compiler from source code.
- It is designed to be executed by the Java Virtual Machine (JVM), enabling Java's "write once, run anywhere" capability.
- Example: Compiling “HelloWorld.java” with javac produces a “HelloWorld.class” file containing bytecode.

2) Java Virtual Machine (JVM)

- The Java Virtual Machine (JVM) is a virtual environment that runs Java bytecode.
- It acts as an intermediary between bytecode and the underlying hardware/OS, interpreting the bytecode into native machine code for execution.
- Example: Running a Java program with the “java” command invokes the JVM to execute the bytecode on the host system.

11) Explain type-conversion in java.

- Type conversion in Java refers to the process of converting a variable from one data type to another. Java supports two main types of type conversion:
  1. **Implicit Type Conversion (Widening)**

     - This type of conversion does not require any explicit cast by the programmer because there is no loss of data.
     - It is also known as widening conversion, occurs automatically when a smaller data type is converted to a larger data type.
     - Example :

```
public class ImplicitConversion {
    public static void main(String[] args) {
        int intValue = 100;
        double doubleValue = intValue; // int to double (implicit)

        System.out.println("Integer value: " + intValue);
        System. out.println("Double value: " + doubleValue);
    }
}
```

2. Explicit Type Conversion (Narrowing)

- This type of conversion requires a cast because it may result in loss of data or precision.
- It is also known as narrowing conversion, occurs when a larger data type is converted to a smaller data type.
- Example :

```
public class ExplicitConversion {
   public static void main(String[] args) {
      double doubleValue = 100.04;
      int intValue = (int) doubleValue; // double to int (narrowing conversion)

      System.out.println("Double value: " + doubleValue);
      System.out.println("Integer value: " + intValue);
   }
}
```

12) Compare different types of variables in Java with a suitable example.

1. **Local Variables**:
   - Declared within methods or blocks.
   - Accessible only within those blocks.
   - Must be initialized before use.

2. **Instance Variables**:
   - Declared within a class but outside methods.
   - Accessible by all instance methods.
   - Exist as long as the instance exists.

3. **Class Variables (Static Variables):**
   - Declared with the static keyword.
   - Shared among all instances.
   - Accessed via the class name or instances.

- **Example :**

```
public class VariableExample {

   static int classVariable = 30; // Class variable (static)

   int instanceVariable = 20; // Instance variable

   public void display() {
      int localVariable = 10; // Local variable
      System.out.println("Class Variable: " + classVariable);
      System.out.println("Instance Variable: " + instanceVariable);
      System.out.println("Local Variable: " + localVariable);
   }
```

```
  public static void main(String[] args) {
    VariableExample example = new VariableExample();
    example.display();
  }
}
```

13) What is the difference between while and do-while statement.

Aspect	While loop	do-while loop
Syntax	while (condition) { // body }	do { // body } while (condition);
Condition Check	Condition is checked before entering the loop body	Condition is checked after the loop body has executed
Execution	The loop body may not execute if the condition is false initially	The loop body is guaranteed to execute at least once
Use	Used when the number of iterations is not known and may not need to execute	Used when the loop should execute at least once regardless of the condition
Flow	If the condition is false, the loop body is skipped entirely	The loop body runs once before the condition is evaluated
Example	java while (x < 10) { x++; }	java do { x++; } while (x < 10);

14) Discuss any three features of Java Programming Language.

1. **Platform Independence**
   - Java is platform-independent, meaning code written in Java can run on any device or OS with a Java Virtual Machine (JVM).
   - Java source code is compiled into bytecode, which can be executed on any platform with a JVM, making it cross-platform.
   - Example: A Java program compiled on Windows can run on Linux or macOS without changes.

2. **Object-Oriented Programming (OOP)**
   - Java uses an object-oriented approach, organizing code into classes and objects. This promotes reusability, modularity, and organization.
   - Key Concepts:
     - Encapsulation: Bundling data and methods in a class, restricting access to certain components.
     - Inheritance: Allows a class to inherit properties and methods from another, promoting reuse.
     - Polymorphism: Enables treating objects as instances of their parent class, allowing flexible method calls.
     - Abstraction: Hides complex details and exposes only essential features.

3. **Automatic Memory Management (Garbage Collection)**
   - Java provides automatic memory management through its built-in garbage collector. This feature helps manage memory allocation and deallocation automatically, reducing the risk of memory leaks and improving the stability of applications.
   - Benefits:
     - Reduces Programmer Burden: No need for manual memory management, reducing errors.
     - Improves Performance: Optimizes memory usage, maintaining application performance.

15) Differentiate between constructor and method of a class.

Aspect	Constructor	Method
Purpose	Initializes objects	Defines behavior and operations
Name	Same as the class name	Any valid method name
Return Type	No return type	Must specify a return type (including void)
Invocation	Called automatically during object creation	Called explicitly using object reference
Overloading	Can be overloaded	Can be overloaded
Inheritance	Not inherited; can be called using "super()"	Inherited and can be overridden

16) How is "this" keyword used to access the data members in a class? Explain it with a suitable Java program.

- The `this` keyword in Java is a reference to the current object within a class, used to access the object's instance variables, methods, and constructors, especially when there's same naming between instance variables and method parameters.
- Use :
  - When a parameter has the same name as an instance variable, this helps distinguish between the instance variable and the parameter. Without this, the parameter would shadow the instance variable, making it inaccessible within the method or constructor.
- Example:

```
public class Employee {
    // Instance variables
    private int id;
    private String name;

    // Constructor with parameters having the same names as instance variables
    public Employee(int id, String name) {
        // Using 'this' keyword to refer to instance variables
        this.id = id;
        this.name = name;
    }

    public void display() {
        System.out.println("Employee ID: " + this.id);
        System.out.println("Employee Name: " + this.name);
    }

    public static void main(String[] args) {

        Employee emp = new Employee(101, "John Doe");
        emp.display();
    }
}
```

17) Declare a class "Product" which has private data members such as "id", "name" and "price". Write constructors for "Product" class, display() method to display values of the objects.

```
public class Product {

   private int id;
   private String name;
   private double price;

   public Product(int id, String name, double price) {
      this.id = id;
      this.name = name;
      this.price = price;
   }
   public void display() {
      System.out.println("Product ID: " + id);
      System.out.println("Product Name: " + name);
      System.out.println("Product Price: $" + price);
   }

   public static void main(String[] args) {

      Product product = new Product(101, "Laptop", 999.99);
      product.display();
   }
}
```

18) Write a program to create circle class with area function to find area of circle.

```
import java.util.Scanner;
public class Circle {

   double radius;

   public Circle(double radius) {
      this.radius = radius;
   }

   public double calculateArea() {
      return Math.PI * radius * radius;
   }

   public void display() {
      System.out.println("Area: " + calculateArea());
   }

   public static void main(String[] args) {

      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter the radius of the circle: ");
```

```
        double userRadius = scanner.nextDouble();

        Circle circle = new Circle(userRadius);
        circle.display();
        scanner.close();
    }
}
```

19) Enlist types of constructors. Explain any two with example.

1. **Default Constructor**
   - A default constructor is a no-argument constructor that is automatically provided by the Java compiler if no other constructors are defined in the class.
   - It initializes objects with default values. If the class has no fields or methods to initialize, the default constructor does nothing.
   - Example :

```
class Dog {

    String name;

    // Default constructor
    public Dog() {
        this.name = "Unknown";
    }

    public void display() {
        System.out.println("Dog's name: " + name);
    }

    public static void main(String[] args) {
        Dog dog1 = new Dog();
        dog1.display();  // Outputs: Dog's name: Unknown
    }
}
```

2. **Parameterized Constructor**
   - A parameterized constructor is one that takes arguments. It allows you to pass different values by creating objects to initialize values.
   - It provides flexibility in initializing objects with different data at the time of object creation.
   - Example :

```
class Dog {

    String name;

    // Parameterized constructor
    public Dog(String name) {
        this.name = name; // Initializes the name with the provided value
    }
```

```
    public void display() {
        System.out.println("Dog's name: " + name);
    }

    public static void main(String[] args) {
        Dog dog1 = new Dog("Buddy");
        dog1.display();  // Outputs: Dog's name: Buddy
    }
}
```

3. Copy Constructor

- A copy constructor creates a new object by copying the values from an existing object.
- It is used to create a duplicate of an existing object with the same state.
- Example :

```
class Dog {
    String name;

    // Parameterized constructor
    public Dog(String name) {
        this.name = name;
    }

    // Copy constructor
    public Dog(Dog another) {
        this.name = another.name; // Copies the name from the passed object
    }

    public void display() {
        System.out.println("Dog's name: " + name);
    }

    public static void main(String[] args) {
        Dog dog1 = new Dog("Buddy");
        Dog dog2 = new Dog(dog1);  // Calls the copy constructor

        dog1.display();  // Outputs: Dog's name: Buddy
        dog2.display();  // Outputs: Dog's name: Buddy (copy of dog1)
    }
}
```

20) Explain static variable and static method with example.

- **Static Variable :**
  - A static variable is a class-level variable that is shared among all instances of the class.
  - The static variable is initialized only once at the start of the execution and retains its value across all instances of the class.
  - Example :

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

   // Constructor to increment count whenever a new object is created
   public Counter() {
      count++;
   }

   // Method to display the count
   public void displayCount() {
      System.out.println("Count: " + count);
   }

   public static void main(String[] args) {
      Counter c1 = new Counter();  // count = 1
      Counter c2 = new Counter();  // count = 2

      c1.displayCount();  // Outputs: Count: 2
      c2.displayCount();  // Outputs: Count: 2
   }
}
```

- **Static Method :**
  - A static method is a method that belongs to the class rather than to any specific instance.
  - It can be called without creating an instance of the class.
  - Static methods can only access other static variables and static methods of the class.
  - Example :

```
class MathOperations {
   // Static method to add two numbers
   public static int add(int a, int b) {
      return a + b;
   }

   // Static method to multiply two numbers
   public static int multiply(int a, int b) {
      return a * b;
   }

   public static void main(String[] args) {
      // Calling static methods without creating an instance
      int sum = MathOperations.add(5, 10);
      int product = MathOperations.multiply(5, 10);

      System.out.println("Sum: " + sum);   // Outputs: Sum: 15
      System.out.println("Product: " + product);   // Outputs: Product: 50
   }
}
```

21) What are the different access control/specifiers in Java. Explain "default" access control using the Java code.

Modifier / Access Control	public	Protected	default (friendly)	Private
Same Class	Yes	Yes	Yes	Yes
Subclass in Same Package	Yes	Yes	Yes	No
Other Class in same Package	Yes	Yes	Yes	No
Subclass in other Package	Yes	Yes	No	No
Non-subclasses in other package	Yes	No	No	No

Dr. K. V. Mehre, SOCSET, ITM SLS Baroda University 73

1. **public:** Accessible from any other class.
2. **protected:** Accessible within the same package and subclasses.
3. **private:** Accessible only within the same class.
4. **default (no modifier):** Accessible only within the same package.

❖ **"default" access control :**

- When no access modifier is specified, the access level is default. This means that the member is accessible only within classes in the same package.
- Example :

```
package PackageA;

class Person {
   // Default access variable
   String name = "John Doe";

   // Default access method
   void printName() {
      System.out.println("Name: " + name);
   }
}

public class TestPerson {
   public static void main(String[] args) {
      Person person = new Person();
      person.printName();  // Accessing default access method
      System.out.println("Person's name: " + person.name);  // Accessing default access variable
   }
}
```

22) Illustrate the concept of polymorphism/method overloading in Java with a suitable example.

- **Polymorphism** in Java allows methods to perform different tasks based on the object or parameters passed to them. One common form of polymorphism is **method overloading**.
- **Method Overloading :**

- Method overloading occurs when two or more methods in the same class have the same name but different parameter lists (different number, type, or order of parameters).
- It allows a class to perform a similar operation in different ways depending on the input parameters.
- Example :

```
class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }
    // Overloaded method to add two double values
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calls the method to add two integers
        System.out.println(calc.add(10, 20)); // Outputs: 30

        // Calls the overloaded method to add three integers
        System.out.println(calc.add(10, 20, 30)); // Outputs: 60

        // Calls the overloaded method to add two doubles
        System.out.println(calc.add(5.5, 4.5)); // Outputs: 10.0
    }
}
```

23) What is wrapper class? What is the use of wrapper class in Java?

- Wrapper classes in Java encapsulate primitive data types in objects. The eight primitive types have corresponding wrapper classes:

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

- Uses :
  1. **Object Manipulation:**
     - Purpose: Java collections (e.g., ArrayList, HashMap) work with objects, not primitives.
     - Example: "**ArrayList<Integer> list = new ArrayList<>(); list.add(10);**"

2. **Autoboxing and Unboxing:**
   - Purpose: Automatically converts between primitives and their wrapper objects.
   - Example: "**Integer num = 5; int value = num;**"

3. **Utility Methods:**
   - Purpose: Provides methods for converting strings to primitives and more.
   - Example: "**int num = Integer.parseInt("123");**"

4. **Nullability:**
   - Purpose: Wrapper classes can be null, unlike primitives.
   - Example: "**Integer num = null;**"

24) Differentiate between method overloading and method overriding.

Sr. No.	Method Overloading	Method Overriding
1)	Used to increase the readability of the program.	Used to provide a specific implementation of a method in a subclass.
2)	Performed within the same class.	Occurs between two classes in an inheritance (IS-A) relationship.
3)	Parameters/signature must be different.	Parameters/signature must be the same.
4)	Example of compile-time polymorphism.	Example of run-time polymorphism.
5)	Cannot change the return type alone; parameters must also differ.	Return type must be the same or covariant (subtype).

25) Explain the uses of "static " keyword in Java.

- **"static" Keyword :**
  - The "static" keyword in Java indicates that a particular member (variable, method, or block) belongs to the class itself rather than to instances of the class.
  - Static members are shared across all instances of the class.
  - Uses :

    1. **Static Variables:** Shared among all instances; useful for class-wide constants or counters.
    2. **Static Methods:** Can be called without creating an instance; ideal for utility functions.
    3. **Static Blocks:** Used for static initialization of class-level resources.
    4. **Static Nested Classes:** Group related classes without requiring access to the outer class's instance members.

  - The "static" keyword is essential for optimizing memory usage and defining class-level behaviors in Java.

26) Write a program to implement the Fibonacci series using for loop control statement.

```
import java.util.Scanner;

public class Fibonacci {

   public static int fibo(int n) {
      if (n == 0) {
         return 0;
```

```
    } else if (n == 1) {
        return 1;
    } else {
        return fibo(n - 1) + fibo(n - 2);
    }
  }

  public static void main(String[] args) {

    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter the number of terms in the Fibonacci series: ");
    int n = scanner.nextInt();

    System.out.println("Fibonacci Series :");

    for (int i = 0; i < n; i++) {
        System.out.println(fibo(i));
    }
  }
}
```

27) Explain features of JAVA.

a) **Simple:** Easy to learn with a clean syntax, removing complex features like pointers.
b) **Object-Oriented:** Follows OOP principles like inheritance, polymorphism, encapsulation, and abstraction.
c) **Platform-Independent:** Write once, run anywhere—Java bytecode runs on any system with a JVM.
d) **Robust:** Exception handling and garbage collection make Java reliable and error-resistant.
e) **Multithreaded:** Supports concurrent execution of threads for better performance.
f) **High Performance:** Optimized with Just-In-Time (JIT) compilation for faster execution.
g) **Portable:** Code can run on different platforms without modification.

28) Explain "instanceof" operator.

- The instanceof operator in Java is used to test whether an object is an instance of a specific class or interface.
- It returns a boolean value (true or false), helping to determine the type of an object at runtime.
- Syntax :

```
object instanceof ClassName
```

- Example :

```
class Animal {}
class Dog extends Animal {}

public class InstanceOfExample {
    public static void main(String[] args) {
        Dog dog = new Dog();

        // Check if dog is an instance of Dog
        System.out.println(dog instanceof Dog);       // Output: true
```

```
        // Check if dog is an instance of Animal
        System.out.println(dog instanceof Animal);      // Output: true

        // Check if dog is an instance of Object
        System.out.println(dog instanceof Object);      // Output: true

        // Check against a different class
        System.out.println(dog instanceof String);      // Output: false
    }
}
```

29) Explain the concept of multilevel inheritance with a suitable example.

- Multilevel inheritance is a type of inheritance where a class derives from another class, forming a hierarchy.
- In this structure, the subclass inherits properties and methods from its immediate superclass and all ancestor classes.
- Example :

```
// GrandParent (Super class)
class GrandParent {
    void func1() {
        System.out.println("GrandParent.");
    }
}
// Parent (Derived class)
class Parent extends GrandParent {
    void func2() {
        System.out.println("Parent.");
    }
}
// Child (Sub class)
class Parent extends Child {
    void func3() {
        System.out.println("Child.");
    }
}
// Main class to test multilevel inheritance
public class Multilevel {
    public static void main(String[] args) {
        Child myChild = new Child();
        // Calling methods from all classes
        myChild.func1();  // Inherited from GrandParent
        myChild.func2(); // Inherited from Parent
        myChild.func3(); // Specific to Child
    }
}
```

30) Explain the concept of multiple inheritance with a suitable example.

- Multiple inheritance is a feature where a class can inherit from more than one superclass.

- However, Java does not support multiple inheritance with classes to avoid ambiguity and complexity. Instead, Java allows multiple inheritance through interfaces.
- Example :

```
// First interface
interface GrandParent {
   void displayGrandParent();
}
// Second interface
interface Parent {
   void displayParent();
}
// Class implementing both interfaces
class Child implements GrandParent, Parent {
   public void displayGrandParent() {
      System.out.println("This is the GrandParent.");
   }
   public void displayParent() {
      System.out.println("This is the Parent.");
   }
}
// Main class to test multiple inheritance
public class MultipleInheritanceExample {
   public static void main(String[] args) {
      Child myChild = new Child();
      // Calling methods from both interfaces
      myChild.displayGrandParent(); // Output: This is the GrandParent.
      myChild.displayParent(); // Output: This is the Parent.
   }
}
```

31) Describe Inheritance and its type with suitable example.

- Inheritance is a fundamental Object-Oriented Programming (OOP) concept that allows a new class (childclass) to inherit properties and behaviors (methods) from an existing class (parent class).
- This promotes code reusability and establishes a hierarchical relationship between classes.
- Types of Inheritance :
  - **a) Single Inheritance :**
    - A class inherits from one superclass.
    - Example :

```
class Parent {
   void display() {
      System.out.println("This is the Parent.");
   }
}

class Child extends Parent {
   void show() {
      System.out.println("This is the Child.");
   }
}
```

b) Multilevel Inhertance :

- A class inherits from another class, forming a chain of inheritance.
- Example :

```
class GrandParent {
   void display() {
      System.out.println("This is the GrandParent.");
   }
}

class Parent extends GrandParent {
   void show() {
      System.out.println("This is the Parent.");
   }
}

class Child extends Parent {
   void print() {
      System.out.println("This is the Child.");
   }
}
```

c) Hierarchical Inheritance :

- Multiple classes inherit from a single superclass.
- Example :

```
class GrandParent {
   void display() {
      System.out.println("This is the GrandParent.");
   }
}

class Parent extends GrandParent {
   void show() {
      System.out.println("This is the Parent.");
   }
}

class Child extends GrandParent {
   void print() {
      System.out.println("This is the Child.");
   }
}
```

d) Multiple Inhertance :

- A class implements multiple interfaces.
- Example :

```
interface GrandParent {
   void display();
}

interface Parent {
   void show();
}

class Child implements GrandParent, Parent {
   public void display() {
      System.out.println("This is the GrandParent interface.");
   }

   public void show() {
      System.out.println("This is the Parent interface.");
   }
}
```

32) Discuss Following with example: (i) super Keyword (ii) final Keyword (iii) this Keyword.

I. super Keyword

- The super keyword in Java is used to refer to the immediate parent class's properties and methods.
- Example :

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

class Dog extends Animal {
   void sound() {
      super.sound(); // Calls the sound method from Animal
      System.out.println("Dog barks.");
   }
}

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

II. final Keyword

- The final keyword in Java is used to declare constants, prevent method overriding, and prevent inheritance of classes.
- Example :

```
class FinalClass {
    final void show() {
        System.out.println("This method cannot be overridden.");
    }
}

// class SubClass extends FinalClass { } // This will cause a compile-time error

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

III. this Keyword

- The this keyword in Java refers to the current object instance.
- Example :

```
class Person {
    String name;

    Person(String name) {
        this.name = name; // 'this' refers to the instance variable
    }
    void display() {
        System.out.println("Name: " + this.name);
    }
}

public class ThisKeywordExample {
    public static void main(String[] args) {
        Person person = new Person("Alice");
        person.display();
    }
}
```

33) What is dynamic method dispatch? Explain with suitable example.

- Dynamic Method Dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than compile-time.
- This is an essential feature of Java's polymorphism, allowing a program to determine which method to execute based on the object being referred to.
- Example :

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

```
    }
}

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

// Another Subclass
class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows.");
    }
}

public class DynamicMethod {
    public static void main(String[] args) {
        Animal myAnimal;

        myAnimal = new Dog(); // Animal reference to Dog object
        myAnimal.sound();     // Output: Dog barks.

        myAnimal = new Cat(); // Animal reference to Cat object
        myAnimal.sound();     // Output: Cat meows.
    }
}
```

34) What do you mean by Interface? Explain with an example.

- An interface in Java is a reference type that defines a contract for classes to implement. It can contain method signatures (abstract methods), default methods, static methods, and constants, but not instance fields or constructors.
- The interface in Java is a mechanism to achieve abstraction.
- Syntax :

```
interface InterfaceName {
    returnType methodName(parameters);
}
```

- Example :

```
// Define the interface
interface Animal {
    void sound(); // Abstract method
}
// Implementing the interface in classes
class Dog implements Animal {
    public void sound() {
```

```
        System.out.println("Dog barks.");
    }
}
class Cat implements Animal {
    public void sound() {
        System.out.println("Cat meows.");
    }
}
// Main class to test the interface
public class InterfaceExample {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();
        dog.sound(); // Output: Dog barks.
        cat.sound(); // Output: Cat meows.
    }
}
```

35) Explain Exception handling in JAVA.

- Exception handling in Java is a mechanism to manage runtime errors, allowing programs to continue or terminate gracefully instead of crashing.
- Handling :
  - **Try-Catch Block :** Encloses code that may throw exceptions. If an exception occurs, control transfers to the catch block.
  - **Syntax :**

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handle exception
}
```

  - **Example :**

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0); // Throws ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Caught an exception: " + e.getMessage());
        }
    }
    public static int divide(int a, int b) {
        return a / b; // Division by zero causes an exception
    }
}
```

36) Discuss exception and error in Java.

a) Exception

- An event that disrupts the normal flow of a program. Exceptions can be caught and handled.
- Handling : Managed using try-catch blocks.
- Example :

```
public class ExceptionExample {
   public static void main(String[] args) {
      try {
         int result = 10 / 0; // Throws ArithmeticException
      } catch (ArithmeticException e) {
         System.out.println("Caught an exception: " + e.getMessage());
      }
   }
}
```

b) Error

- It is an unexpected situation that occurs during the execution of a system or application.It leads to the termination of the code.
- Handling : Errors are not meant to be caught or handled in typical application code.
- Example :

```
public class ErrorExample {
   public static void main(String[] args) {
      try {
         // This may lead to StackOverflowError
         recursiveMethod();
      } catch (StackOverflowError e) {
         System.out.println("Caught an error: " + e.getMessage());
      }
   }
   static void recursiveMethod() {
      recursiveMethod(); // Infinite recursion
   }
}
```

37) When do we use throw statement? Explain it by giving an example

- The throw keyword is used to explicitly throw an exception when a specific error condition is met.
- Syntax :

```
throw new ExceptionType("Error message");
```

- Example :

```
public class DivisionExample {
   public static void main(String[] args) {
      try {
         // Attempting to divide by zero
         int result = divide(10, 0);
         System.out.println("Result: " + result);
```

```
    } catch (ArithmeticException e) {
        System.out.println("Exception caught: " + e.getMessage());
    }
}

// Method that throws an exception if the denominator is zero
public static int divide(int a, int b) {
    if (b == 0) {
        throw new ArithmeticException("Cannot divide by zero."); // Throwing an exception
    }
    return a / b;
}
}
```

38) What is the keyword throws used for? Illustrate with a suitable example

- The **throws** keyword in Java is used to declare exceptions that a method might throw during its execution. It informs the compiler that the method could potentially cause an exception, and the calling method must handle it.
- Primarily used for checked exceptions (e.g., IOException, SQLException) that must be either caught or declared.
- Syntax :

```
returnType methodName() throws ExceptionType {
    // method code
}
```

- Example :

```
public class ThrowsExample {
    public static void main(String[] args) {
        try {
            divide(10, 0); // This will throw an exception
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }

    // Method declares it might throw an ArithmeticException
    public static void divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        System.out.println("Result: " + (a / b));
    }
}
```

39) Explain any three built-in exceptions with examples.

a) ArithmeticException

- This exception is thrown when an arithmetic operation fails, such as dividing by zero.
- Example :

```
public class ArithmeticExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

b) ArrayIndexOutOfBoundsException

- This exception is thrown when an attempt is made to access an array with an invalid index
- Example :

```
public class ArrayIndexOutOfBoundsExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        try {
            int num = numbers[5]; // Accessing invalid index
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

c) NullPointerException

- This exception is thrown when an application attempts to use null where an object is required.
- Example :

```
public class NullPointerExceptionExample {
    public static void main(String[] args) {
        String str = null;
        try {
            int length = str.length(); // Trying to access length of a null reference
        } catch (NullPointerException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

40) Compare interface and abstract class with suitable example

Feature	Interface	Abstract Class
Definition	A contract defining methods without implementation.	A class that cannot be instantiated and can contain both abstract and concrete methods.
Keyword	Defined using interface.	Defined using abstract.
Method types	All methods are abstract	Can have both abstract and concrete methods.
Inhertance	Supports multiple inheritance.	Supports single inheritance only.
Fields	Only static final fields (constants).	Can have instance variables and constants.
Access Modifiers	Methods are implicitly public.	Can use any access modifier.

Interface Example :

```
interface Vehicle {
    void start(); // abstract method
}
class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting.");
    }
}
public class InterfaceExample {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start(); // Output: Car is starting.
    }
}
```

Abstract Class Example :

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

    void info() { // concrete method
        System.out.println("This is an animal.");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks.");
    }
}
public class AbstractClassExample {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound(); // Output: Dog barks.
        myDog.info();  // Output: This is an animal.
    }
}
```

41) Why is finally block used in exception handling? Illustrate with a suitable example.

- The finally block in Java is used in exception handling to ensure that a block of code is executed regardless of whether an exception occurs or not.
- **Optional:** It is not mandatory to use a finally block, but it ensures that critical code runs (like closing database connections).
- **Used with try-catch:** It's commonly used in conjunction with try and catch blocks.

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int data = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("Finally block executed.");
        }
    }
}
```

42) Write a program to rise and handle divide by zero exception.

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int data = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("Finally block executed.");
        }
    }
}
```

43) Explain the following keywords: a) try b) catch c) throw

a) try

- The try block contains code that might throw an exception. It helps prevent program crashes by detecting exceptions at runtime.
- Example :

```
try {
    int result = 10 / 0; // Risky code
}
```

b) catch

- The catch block handles the exception thrown by the try block. It catches the specified exception and provides a way to handle it gracefully.
- Example :

```
catch (ArithmeticException e) {
    System.out.println("Error: Division by zero");
}
```

c) throw

- The throw keyword is used to explicitly throw an exception when a specific error condition is met.
- Example :

```
if (b == 0) {
    throw new ArithmeticException("Cannot divide by zero");
}
```