1. List and explain the Java buzzwords OR List and explain salient features of Java

Java is known for its unique characteristics that make it one of the most popular programming languages. Here are the key buzzwords (features):

1. Simple

- Java has a clean syntax, similar to C++ but without complex features like pointers and operator overloading.

2. Object-Oriented

- Java uses the concept of "objects," allowing for easier program structure by modeling real-world entities.

3. Platform-Independent

- Java code is compiled into bytecode that can run on any machine with the Java Virtual Machine (JVM), making it "write once, run anywhere."

4. Secured

- Java provides features like bytecode verification, no explicit pointer handling, and security management to prevent harmful code.

5. Robust

- Java has strong memory management, exception handling, and automatic garbage collection.

6. Multithreaded

- Java allows the execution of multiple parts of a program simultaneously, improving performance.

7. Portable

- Java bytecode can run on any system, ensuring portability across platforms.

8. High Performance

- Java's Just-In-Time (JIT) compiler improves the speed of the application.

9. Distributed

- Java has built-in networking libraries to build distributed applications.

10. Dynamic

- Java can dynamically link new class libraries, methods, and objects at runtime.

Q2. Explain object-oriented principles

Java’s Object-Oriented Programming (OOP) model is based on the following four main principles:

1. Encapsulation

- *Definition:* Encapsulation is the concept of wrapping data (variables) and code (methods) together into a single unit, usually a class, and restricting access to some components.
- *Benefits:* It protects an object’s internal state and hides unnecessary details from the outside world.

- **Example:**

```
class Car {
    private String model;  // Private: can't be accessed directly
    public void setModel(String model) { this.model = model; }
    public String getModel() { return model; }
}
```

Output:

```
Car Model: Tesla
```

2. Inheritance

- *Definition:* Inheritance allows one class to inherit properties and methods from another class, enabling code reuse and creating a parent-child relationship.
- *Benefits:* Reusability of code, method overriding, and organized code structure.

- **Example:**

```
class Vehicle {
    void start() {
        System.out.println("Vehicle started");
    }
}

class Car extends Vehicle {
    void honk() {
        System.out.println("Car honking");
    }
}

public class TestVehicle {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();  // Inherited method from Vehicle
        myCar.honk();   // Method from Car class
    }
}
```

Output:

```
Vehicle started
Car honking
```

3. Polymorphism

- *Definition:* Polymorphism allows a single action to be performed in different ways. It's mainly achieved through method overriding (runtime) and method overloading (compile-time).
- *Benefits:* Flexibility and ease in maintaining code.

- **Example:**

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

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

```
public class TestAnimal {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Animal object
        Animal myDog = new Dog();        // Dog object upcasted as Animal

        myAnimal.sound();  // Animal's sound
        myDog.sound();     // Dog's sound (runtime polymorphism)
    }
}
```

Output:

```
Animal makes sound
Dog barks
```

4. Abstraction

- *Definition:* Abstraction is the process of hiding the implementation details and showing only the functionality to the user.
- *Benefits:* Simplifies complex systems and promotes reusability.

- **Example:**

```
abstract class Vehicle {
    abstract void start();
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts");
    }
}

public class TestAbstraction {
    public static void main(String[] args) {
        Vehicle myCar = new Car();  // Abstraction at work
        myCar.start();
    }
}
```

Output:

```
Car starts
```

Q3. Explain different lexical issues (tokens) in Java

In Java, lexical tokens are the smallest meaningful units in a program. The key lexical components are:

1. Keywords

- *Definition:* Reserved words in Java that have a predefined meaning and cannot be used for anything else, like class, if, public, void.
- **Example:**

```
public class MyClass {
    public void method() {
        System.out.println("Method executed");
    }
}
```

Output:

```
Method executed
```

2. Identifiers

- *Definition:* Names given to classes, methods, variables, and objects. They should start with a letter or an underscore, and they must not be Java keywords.
- **Example:**

```
class Car {
    int speed;
    String model;
}
```

3. Literals

- *Definition:* Constant values assigned to variables. They can be integers, floating-point numbers, characters, strings, or booleans.

4. Operators

- *Definition:* Symbols used to perform operations on variables and values, like +, -, *, /.

- **Example:**

```
public class TestOperators {
    public static void main(String[] args) {
        int sum = 5 + 10;  // Using the addition operator
        System.out.println("Sum: " + sum);
    }
}
```

Output:

```
Sum: 15
```

5. Separators

- *Definition:* Special symbols used to separate code components, such as braces {}, parentheses (), and semicolons ;.

- **Example:**

```
public class TestSeparators {
    public static void main(String[] args) {
        int x = 10;  // Semicolon separates statements
        System.out.println("x: " + x);
    }
}
```

Output:

```
x: 10
```

6. Comments:

Documentation within the code that is ignored by the compiler.

Example:

```
// Single-line comment
/* Multi-line comment */
```

Q4. Explain the scope and lifetime of a variable with examples:

The scope of a variable refers to the region of the program where the variable is accessible. The lifetime of a variable is the duration for which the variable exists in memory.

1.Local Variables: Declared inside a method or block and accessible only within that method or block. They are created when the method is called and destroyed when the method exits.

Example:

```
public class Example {
    public void display() {
        int localVar = 10; // Local variable
        System.out.println(localVar);
    }
}
```

2. Instance Variables:
Declared inside a class but outside any method. They are accessible by all methods of the class and exist as long as the object exists.
Example:

```
public class Example {
    int instanceVar; // Instance variable

    public void display() {
        System.out.println(instanceVar);
    }
}
```

3. Static Variables: Declared as static and shared among all instances of a class. They are created when the class is loaded and destroyed when the program ends.

```
public class Example {
    static int staticVar; // Static variable

    public static void display() {
        System.out.println(staticVar);
    }
}
```

5. Explain the Structure of a Java Program and Its Keywords

A typical Java program consists of the following structure:

1 Package Declaration:
Specifies the package to which the class belongs. It is optional.
package
mypackage;

2. Import Statements: Used to import other classes or packages.

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

3.Class Declaration: Defines a class using the class keyword.

```
public class MyClass {
    // Class body
}
```

4.Main Method: The entry point of any Java program. It is always

```
public static void main(String[] args).
public static void main(String[] args) {
    // Code to be executed
}
```

Example code:

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

Output: Hello, World!

6. Explain the Concept of Arrays in Java with an Example

An array is a collection of similar types of data stored in contiguous memory locations. It is used to store multiple values in a single variable.

1. Declaration: Arrays are declared by specifying the data type followed by square brackets.

```
int[] numbers;
```

2. Instantiation: Arrays are instantiated using the new keyword.

```
numbers = new int[5];
```

3. Initialization: Arrays can be initialized at the time of declaration.

```
int[] numbers = {1, 2, 3, 4, 5};
```

Example Program:

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        for (int i = 0; i < numbers.length; i++) {
            System.out.println(numbers[i]);
        }
    }
}
```

Output:

```
1
2
3
4
5
```

7. Write a Java Program to Perform Arithmetic Operations Based on User Choice

```
import java.util.Scanner;

public class ArithmeticOperations {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter first number:");
        double num1 = scanner.nextDouble();

        System.out.println("Enter second number:");
        double num2 = scanner.nextDouble();

        System.out.println("Choose an operation: +, -, *, /");
        char operation = scanner.next().charAt(0);

        double result;

        switch (operation) {
            case '+':
                result = num1 + num2;
                System.out.println("Result: " + result);
                break;
            case '-':
                result = num1 - num2;
                System.out.println("Result: " + result);
                break;
            case '*':
                result = num1 * num2;
                System.out.println("Result: " + result);
                break;
            case '/':
                if (num2 != 0) {
                    result = num1 / num2;
                    System.out.println("Result: " + result);
                } else {
                    System.out.println("Cannot divide by zero");
                }
                break;
            default:
                System.out.println("Invalid operation");
        }

        scanner.close();
    }
}
```

8. Explain Operators with an Example

Operators are special symbols that perform operations on operands. Java has several types of operators:

1. Arithmetic Operators: Used to perform basic arithmetic operations.
Examples: +, -, *, /, %.

```
int sum = 10 + 5; // 15
```

2. Relational Operators: Used to compare two values.
Examples: ==, !=, >, <, >=, <=.

```
boolean isEqual = (10 == 5); // false
```

3. Logical Operators: Used to combine multiple boolean expressions.
Examples: &&, ||, !.
```
boolean result = (10 > 5) && (5 < 10); // true
```

4. Assignment Operators: Used to assign values to variables.
Examples: =, +=, -=, *=, /=, %=.

```
int a = 10;
a += 5; // a = 15
```

5. Unary Operators: Used with a single operand.
Examples: +, -, ++, --, !.

```
int b = 10;
b++; // b = 11
```

Example Program:

```
public class OperatorExample {
    public static void main(String[] args) {
        int a = 10, b = 5;
        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));
    }
}
```

Output:

```
a + b = 15
a - b = 5
a * b = 50
a / b = 2
a % b = 0
```