MODULE 1

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 OOP is based on four main principles: **Encapsulation, Inheritance, Polymorphism, and Abstraction.** Each principle contributes to creating modular, reusable, and flexible code.

1. Encapsulation

- **Definition:**

Encapsulation is the practice of **bundling data (fields) and methods** that operate on the data into a single unit, usually a class. It also restricts direct access to certain components to maintain control over the data.

- **Benefits:**

1. Protects an object’s internal state.
2. Hides unnecessary implementation details from the user.
3. Promotes data security and integrity.
4. Facilitates easier debugging and maintenance.

- **Example:**

```
class Car
 {
   private String model;
   public void setModel(String m) { model = m; }
   public String getModel()
       {
              return model;
       }
}
public class Test
{
   public static void main(String[] args)
       {
      Car c = new Car();
      c.setModel("Tesla");
      System.out.println(c.getModel());
       }
}
```

Output :- Tesla

2. Inheritance

- **Definition:**

Inheritance enables **a class (child class) to inherit properties and behaviors from another class (parent class).** It promotes code reuse and establishes a parent-child relationship between classes.

- **Benefits:**

1. Encourages code reusability.
2. Simplifies the addition of new features.
3. Facilitates method overriding to modify inherited behaviors.

- **Example:**

```
class Vehicle {
   void start() { System.out.println("Starting..."); }
}
class Car extends Vehicle {}
public class Test {
   public static void main(String[] args) {
      Car c = new Car();
      c.start();
   }
}
```

Op:- Starting...

3. Polymorphism

- **Definition:**

Polymorphism means **“many forms.”** It allows the same action to be performed in multiple ways, achieved via **method overloading** (compile-time polymorphism) and **method overriding** (runtime polymorphism).

- **Benefits:**

1. Enhances flexibility in code design.
2. Makes it easier to maintain and extend the application.
3. Promotes dynamic behavior in programs.

- **Example:**

```
class Animal {
   void sound() { System.out.println("Some sound"); }
}
class Dog extends Animal {
   void sound() { System.out.println("Bark"); }
}
public class Test {
   public static void main(String[] args) {
      Animal a = new Dog();
      a.sound();
   }
}
```

Op:- bark

4. Abstraction

- **Definition:**

Abstraction involves **hiding the complex implementation details of a system** and exposing only the essential features or functionality to the user. It can be achieved through **abstract classes or interfaces in Java.**

- **Benefits:**

1. Simplifies code by reducing complexity.
2. Increases code modularity and flexibility.
3. Promotes reusability and scalability.

- **Example:**

```
abstract class Vehicle {
   abstract void start();
}
class Car extends Vehicle {
   void start() { System.out.println("Car starts"); }
}
public class Test {
   public static void main(String[] args) {
      Vehicle v = new Car();
      v.start();
   }
}.
```

Op:- Car starts

Q3. Explain different lexical issues (tokens) in Java

Lexical tokens are the **smallest meaningful elements of a Java program**. Key tokens include:

1. Keywords

- **Definition:** Reserved words in Java with predefined meanings. They cannot be used for variable or method names.
- **Examples:** class, if, public, void, static, etc.

- **Code Example:**

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

//Hello, Java!

2. Identifiers

- **Definition:** Names assigned to variables, methods, classes, or objects. Must start with a letter or underscore and cannot be a keyword.

- **Code Example:**

```
class Car {
    int speed;  // speed is an identifier
    String model;  // model is an identifier
}
```

3. Literals

- **Definition:** Constant values directly used in a program. Examples: 10, "Hello", true.
- **Types:**
- Integer: int x = 5;
- Floating-point: float pi = 3.14f;
- Character: char grade = 'A';
- String: String name = "John";
- Boolean: boolean isOn = true;

4. Operators

- **Definition:** Symbols used to perform operations like addition (+), subtraction (-), multiplication (*), division (/), etc.
- Code Example:

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

// op sum = 15

5. Separators

- **Definition:** Symbols used to separate code elements, such as { }, ( ), ;, and ,.
- **Code Example:**

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

Op:- Value of x: 10

6. Comments

- **Definition:** Non-executable lines in the code that provide explanations. They are ignored by the compiler.
- **Types:**
- Single-line: // This is a comment
- Multi-line:

```
/*
This is a
multi-line comment
*/
```

Q4. Explain operators with example
If
If else
Switch
While
Do while
for

Control Statements

Control statements in Java help to control the flow of execution of the program based on certain conditions.

1. If Statement

The if statement is used to execute a block of code if a specified condition is true.

Syntax:

```
if (condition) {
   // code block to be executed if condition is true
}
```

Example

```
int x = 10;
if (x > 5) {
   System.out.println("x is greater than 5");
}  // Output: x is greater than 5
```

2. If-Else Statement

The if-else statement allows you to execute one block of code if the condition is true and another block if it is false.

Syntax:

```
if (condition) {
   // code block if condition is true
} else {
   // code block if condition is false
}
```

Example

```
int x = 3;
if (x > 5) {
   System.out.println("x is greater than 5");
} else {
   System.out.println("x is not greater than 5");  // Output: x is not greater than 5
}
```

3. Switch Statement

The switch statement evaluates an expression, matches the value of the expression with the cases, and executes the corresponding block of code.

Syntax:

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

Example

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

4. While Loop

The while loop executes a block of code repeatedly as long as the given condition is true.

Syntax:

```
while (condition) {
    // code block
}
```

Example

```
int i = 1;
while (i <= 5) {
    System.out.println(i);  // Output: 1 2 3 4 5
    i++;
}
```

5. Do-While Loop

The do-while loop executes a block of code at least once, and then repeats the execution as long as the condition is true.

Syntax:

```
do {
    // code block
} while (condition);
```

Example

```
int i = 1;
do {
    System.out.println(i);  // Output: 1 2 3 4 5
    i++;
} while (i <= 5);
```

6. For Loop

The for loop is used when the number of iterations is known beforehand. It has a counter that is initialized, a condition to check, and an increment/decrement operation.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // code block
}
```

Example

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);  // Output: 1 2 3 4 5
}
```

Summary of Control Statements

- If: Executes code if the condition is true.
- If-Else: Executes one block of code if the condition is true, another if false.
- Switch: Matches the value of an expression to one of several cases.
- While: Loops while a condition is true.
- Do-While: Loops at least once, then continues as long as the condition is true.
- For: Loops with a defined number of iterations

5. Explain the Concept of Arrays in Java with an Example
Explain the syntax and declaration of 2D arrays s in Java

An array in Java is a collection of **elements of the same type**, stored in **contiguous memory locations**. Arrays are used to store multiple values in a single variable instead of declaring separate variables for each value.

Key Features of Arrays:

1. Fixed Size: The size of the array must be specified when it is created.
2. Indexed: Array elements are accessed using an index, starting from 0.
3. Homogeneous Elements: All elements in the array must be of the same type.

Types of Arrays:

1. **1D Array:** A single row of elements.
2. **2D Array:** A table-like structure with rows and columns.

Syntax for Declaring and Initializing Arrays:

1. **Declaration:**
   - datatype[] arrayName; or datatype arrayName[];
2. **Initialization:**
   - arrayName = new datatype[size];
   - Combine declaration and initialization: datatype[] arrayName = new datatype[size];

Example: 1D Array

```
public class ArrayExample {
   public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40};  // Declaration and Initialization
      for (int i = 0; i < numbers.length; i++) {
         System.out.println("Element at index " + i + ": " + numbers[i]);
      }
   }
}
```

Output

```
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
```

2D Arrays in Java

A 2D array is an array of arrays, resembling a matrix with rows and columns.

Syntax and Declaration of 2D Arrays:

1. **Declaration:**
   - datatype[][] arrayName;
2. **Initialization:**
   - arrayName = new datatype[rows][columns];
   - Combined: datatype[][] arrayName = new datatype[rows][columns];

```
public class TwoDArrayExample {
    public static void main(String[] args) {
        // Declaration and Initialization
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Displaying 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(); // Newline for the next row
        }
    }
}
```

Output

```
1 2 3
4 5 6
7 8 9
```

Summary:

- 1D Array: Linear structure with one dimension.
- 2D Array: Tabular structure with rows and columns.
- Syntax:
- 1D: datatype[] arrayName = new datatype[size];
- 2D: datatype[][] arrayName = new datatype[rows][columns];

6. List the various operators supported by Java. Illustdate the working of >> 8 and >>> operators with an example.

Java supports the following types of operators:

1. **Arithmetic Operators**: +, -, *, /, %
   - Perform basic mathematical operations.

2. **Relational (Comparison) Operators**: ==, !=, <, >, <=, >=
   - Compare two values.

3. **Logical Operators:** &&, ||, !
   - Perform logical operations.

4. **Bitwise Operators:** &, |, ^, ~, <<, >>, >>>
   - Work on bits and perform bit-by-bit operations.

5. **Assignment Operators**: =, +=, -=, *=, /=, %=
   - Assign values to variables.

6. **Unary Operators:** +, -, ++, --, !
   - Work with a single operand.

7. **Shift Operators:** <<, >>, >>>
   - <<: Left shift, shifts bits to the left and fills with 0.
   - >>: Right shift, shifts bits to the right and fills with the sign bit (preserves the sign).
   - >>>: Unsigned right shift, shifts bits to the right and fills with 0 (ignores the sign).

8. **Ternary Operator:** ? :
   - A shorthand for if-else statements.

9. **Instanceof Operator:**
   - Tests whether an object is an instance of a specific class or subclass.

Illustration of >> and >>> Operators

>> (Signed Right Shift):

- Shifts bits to the right.
- Preserves the sign bit (**MSB**).

Example:

```
public class RightShift {
    public static void main(String[] args) {
        int num = -16; // Binary: 11111111 11111111 11111111 11110000
        int result = num >> 2; // Shift 2 bits to the right
        System.out.println("Signed Right Shift Result: " + result);
    }
}
```

Op:- Signed Right Shift Result: -4

>>> (Unsigned Right Shift):

- Shifts bits to the right.
- Fills the leftmost bits with 0, regardless of the sign.

Example:

```
public class UnsignedRightShift {
    public static void main(String[] args) {
        int num = -16; // Binary: 11111111 11111111 11111111 11110000
        int result = num >>> 2; // Shift 2 bits to the right
        System.out.println("Unsigned Right Shift Result: " + result);
    }
}
```

Op:-
Unsigned Right Shift Result: 1073741820

7. 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);

        // Input from user
        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;

        // Perform the operation based on user choice
        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");
        }

        // Close the scanner
        scanner.close();
    }
}
```

Op:-

```
Enter first number:
15
Enter second number:
3
Choose an operation: +, -, *, /
/

Result: 5.0
```

8. Develop a Java program to Celsius temperature to Fahrenheit.

Code

```
import java.util.Scanner;

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

        // Input temperature in Celsius
        System.out.println("Enter temperature in Celsius:");
        double celsius = scanner.nextDouble();

        // Convert to Fahrenheit
        double fahrenheit = (celsius * 9/5) + 32;

        // Display the result
        System.out.println("Temperature in Fahrenheit: " + fahrenheit);

        scanner.close();
    }
}
```

Example Execution:

Input:

```
Enter temperature in Celsius:
25
```

output

```
Temperature in Fahrenheit: 77.0
```

9 . Program for Matrix Addition

```
import java.util.Scanner;

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

        // Input matrix size
        System.out.println("Enter size of the matrix (n x n): ");
        int n = sc.nextInt();

        // Declare matrices
        int[][] mat1 = new int[n][n];
        int[][] mat2 = new int[n][n];
        int[][] sum = new int[n][n];

        // Input first matrix
        System.out.println("Enter elements of the first matrix:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                mat1[i][j] = sc.nextInt();
            }
        }

        // Input second matrix
        System.out.println("Enter elements of the second matrix:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                mat2[i][j] = sc.nextInt();
            }
        }

        // Add matrices and display result
        System.out.println("Resultant matrix after addition:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                sum[i][j] = mat1[i][j] + mat2[i][j];
                System.out.print(sum[i][j] + " ");
            }
            System.out.println();
        }
        sc.close();
    }
}
```

```
Ip:-
Enter size of the matrix (n x n):
2
Enter elements of the first matrix:
1 2
3 4
Enter elements of the second matrix:
5 6
7 8

Op;-
Resultant matrix after addition:
6 8
10 12
```

10. Justify the statement "Compile once and run anywhere" in Java.

This phrase means **that Java programs can be written and compiled once**, and then they can **be run on any system without changing the code.** Java achieves this because it uses bytecode and the **Java Virtual Machine (JVM).**

How It Works:

1. Java Code → Bytecode:

- When you write a Java program, you write it in human-readable code (e.g., HelloWorld.java).
- This code is compiled into bytecode using the javac compiler.
- The bytecode is not tied to any specific computer or operating system. It's stored in .class files.

2. Bytecode → JVM:

- The bytecode can be run on any system that has a JVM (Java Virtual Machine) installed.
- The JVM reads the bytecode and converts it into machine code that the computer can understand.
- The JVM does this translation based on the operating system it's running on, making the bytecode platform-independent.

Why "Compile Once, Run Anywhere"?

- No need to recompile the code for different platforms (like Windows, Linux, or macOS).
- You can transfer your Java program to any platform with a JVM and it will run the same way.

Key Points:

- Java code is compiled into bytecode.
- Bytecode can run on any platform with a JVM.
- This is why Java programs are platform-independent.

Simple Example:

1. Write Java code (e.g., HelloWorld.java).
2. Compile it using javac to get HelloWorld.class (bytecode).
3. Run the HelloWorld.class file on any system with a JVM.

11. Explain different types of if statements in JAVA

1. Simple if Statement

- **Description:** Executes a block of code if a condition is true.

- **Syntax:**

```
if (condition) {
   // Code to be executed if condition is true
}
```

- **Example:**

```
int num = 10;
if (num > 5) {
   System.out.println("Number is greater than 5");
}
```

- Output: Number is greater than 5

2. if-else Statement

- **Description:** Executes one block of code if the condition is true, and another block if the condition is false.
- **Syntax:**

```
if (condition) {
   // Code to be executed if condition is true
} else {
   // Code to be executed if condition is false
}
```

example

```
int num = 3;
if (num > 5) {
   System.out.println("Number is greater than 5");
} else {
   System.out.println("Number is not greater than 5");
}
```

- **Output:** Number is not greater than 5

3. if-else if-else Ladder

- **Description:** Used when you have multiple conditions to check. It checks the first condition, and if false, it checks the next one, and so on.

- **Syntax:**

```
if (condition1) {
    // Code to be executed if condition1 is true
} else if (condition2) {
    // Code to be executed if condition2 is true
} else {
    // Code to be executed if all conditions are false
}
```

- **Example:**

```
int num = 7;
if (num > 10) {
    System.out.println("Number is greater than 10");
} else if (num == 7) {
    System.out.println("Number is 7");
} else {
    System.out.println("Number is less than 7");
}
```

- **Output:** Number is 7

4. Nested if Statements

- **Description:** An if statement inside another if statement. This allows checking more complex conditions.
- **Syntax:**

```
if (condition1) {
    if (condition2) {
        // Code to be executed if both conditions are true
    }
}
```

Example

```
int num = 10;
if (num > 5) {
    if (num < 20) {
        System.out.println("Number is between 5 and 20");
    }
}
```

- **Output:** Number is between 5 and 20

12. 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!

13. Write a Java program to sort the elements using a for loop

```
import java.util.Scanner;

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

        // Take input for array size
        System.out.print("Enter the number of elements: ");
        int n = sc.nextInt();

        // Declare an array
        int[] arr = new int[n];

        // Input elements into the array
        System.out.println("Enter the elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

        // Sorting the array using for loop (Bubble sort method)
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap elements
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }

        // Display the sorted array
        System.out.println("Sorted Array:");
        for (int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }

        sc.close();
    }
}
```

Sample Output:

```
Enter the number of elements: 5
Enter the elements:
5 2 9 1 3
Sorted Array:
1 2 3 5 9
```