[Q.1]

Q.1 Define following Terms:

Ans. A. Object:- An object is a fundamental unit of the program that represents a real-world entity or concept.
It's an instance of a class, which is a blueprint defining the object's structure and behavior. Object's can have attributes (variables) and methods (functions) that allow them to interact with each other and perform actions.

B. Class:- A class is a blueprint or template that defines the structure and behavior of objects. It serves as blueprint for creating objects, specifying what attributes (variables) an object will have and what methods (functions) it can perform.

C. Abstraction:- Abstraction in Java is a fundamental concept in Object-Oriented Programming that focuses on hiding complex implementation details while exposing only the essential features of an object.

D. Encapsulation :-
Encapsulation in Java is a key concept of Object-oriented Programming that involves bundling an object's data (attributes) and the methods (functions) that operate on that data into a single unit, known as a class.
It also, involves controlling access to an object's internal state by using access modifiers like Public, Private, and Protected.

E. Inheritance :-
Inheritance in Java is a mechanism that allows a new class (subclass or derived class) to inherit attributes and methods from a existing class (superclass or base class).
It promotes code reusability and establishes a hierarchical relationship between classes.

F. Polymorphism:-
Polymorphism in Java is a fundamental concept that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface or method name to represent different behaviors based on the specific objects involved.

G. Byte code:-
Byte code in Java is an intermediate representation of a Java source code program. When you compile a Java source code file, it gets converted into bytecode, which is not machine-specific and can execute on any platform with a Java-virtual machine (JVM).

H. JVM:-
The JVM, or Java virtual machine, is a critical component of the Java runtime Environment. It's a software based virtual machine that executes Java bytecode, which is compiled from the Java source code.

[Q.2]

2. Explain the features of Java.

→ Java is a versatile and widely-used programming language known for its platform independence and robust features. Here are some of its key features:

1. Platform Independence:
   Java is designed to be platform-independent, thanks to the "Write Once, Run Anywhere" (WORA) principle. It achieves this through the use of the Java Virtual Machine (JVM), which allows Java programs to run on any platform that has a compatible JVM.

2. Object-Oriented:
   Java is an object-oriented language, which means it encourages modular, reusable code through the use of classes and objects.

3. Simple and Familiar:
Java's syntax is based on C and C++, making it relatively easy to learn for those familiar with these languages.
It avoids complex features like pointers.

4. Robust:
Java incorporates strong memory management and exception handling, reducing the likelihood of system crashes due to errors. This robustness is a key reason why Java is popular for building large-scale applications.

5. Multi-threading:
Java supports multi-threading, allowing programs to efficiently execute multiple tasks concurrently. This is crucial for developing responsive and scalable applications.

6. Security:
Java has built-in security features, such as the ability to run applets in a restricted environment and a robust access control mechanism.

7. Portability:
Java's platform independence extends to its portability. Code written in Java can be easily moved from one system to another without modification.

8. Rich Standard Library:
Java offers a comprehensive standard library (Java Standard Library or Java API) that provides pre-built classes and methods for various tasks, such as networking, file handling, and data structures.

9. Performance:
Java's performance has improved significantly over the years, thanks to advancements like Just-In-Time (JIT) compilation, which translates Java bytecode into native machine code at runtime for faster execution.

[Q.3]

3. What is JDK? Also, explain the use of its components.

→ JDK stands for Java Development kit, and it's a critical software package in the Java ecosystem. The JDK is primarily used for developing, compiling, and running Java applications and applets. It includes a set of tools and components that are essential for Java development. Here are the main components of the JDK and their uses:

1. Java Compiler (Javac):
- Use: This component is responsible for compiling Java source code files (.java) into bytecode files (.class) that can be executed on the Java Virtual Machine (JVM).

2. Java Virtual Machine (JVM):
   - USE: The JVM is an integral part of the JDK. It's responsible for executing Java bytecode. It provides platform independence, allowing Java applications to run on different operating systems without modification.

3. Java Runtime Environment (JRE):
   - USE: The JRE is a subset of the JDK. It includes the JVM and essential runtime libraries needed to run Java applications. End-Users typically install the JRE to run Java applications without needing the full JDK for development.

4. Java Standard Library (Java API):
   - USE: The JDK comes with an extensive library of pre-built classes and methods known as the Java Standard Library or Java API.

5. Development Tools:
   - USE: The JDK provides several development tools, including:
     - Javac: The Java compiler for converting source code to bytecode.
     - Java: The Java interpreter for running Java applications.
     - Javadoc: A tool for generating documentation from Java source code comments.
     - Jar: A utility for packaging Java classes and resources into JAR (Java Archive) files.
     - Jdb: The Java Debugger for debugging Java programs.
     - Javap: A class files disassembler for examining bytecode.

6. Debugger Interface (JDI):
   - USE: The JDK includes the Java Debugger Interface, which allows developers to create debugging tools that can interact with the JVM for debugging Java applications.

[Q.47]

4. Discuss the Command-line Arguments in Java.

→ In Java, command-line arguments are parameters that you can pass to a Java application when you run it from the command line. These arguments are passed as strings and can be accessed within your Java program using the args parameter of the main method. Here's explanation with an example:

```
Public class CommandLineArguments
Example {

// The 'args' parameter is an
array of string that contains
command-line arguments.

// Check if any arguments were
provided
System.out.println("No
command-line arguments
provided.");
}
```

```
else {
        System.out.println
        ("List of command-line
        arguments:");

// Loop through the 'args' array
   to access and print each
   argument.

   for (int i=0; i<args.length; i++) {
      System.out.println
      ("Argument" + (i+1) + ": "
         + args[i]);
      }
   }
}
```

In this example, we have a Java program called CommandLineArgumentsExample. It defines a main method that takes an array of String args, as a parameter.

Here's how you can run this program and pass command-line arguments:

1. Open a command prompt or terminal.

2. Navigate to the directory where your Java source file (CommandLineArgumentsExample.java) is located.
3. Compile the Java program using the javac command:
   javac CommandLineArgumentsExample.java
4. Run the compiled Java program with command-line arguments.
   java CommandLineArgumentExample arg1 arg2 arg3

In the example above, we passed three command-line arguments (arg1, arg2, arg3) to the Java program.

When you run the program, it will check if any command-line arguments were provided. If arguments are present, it will print each argument along with its position in the args array.

For example, if you run the program as shown above, the output will be:

List of Command-line arguments:
Argument 1: arg 1
Argument 2: arg 2
Argument 3: arg 3

You can use command-line arguments to customize the behavior of your Java programs based on user input or configuration options provided at runtime.

[Q.5]

5. State the difference between Instance variables and Static variables.

→ In Java, instance variables and static variables are two types of variables used in classes, and they have distinct characteristics:

1. Instance Variables:
   - Also known as non-static variables.
   - Each instance (object) of a class has its own copy of instance variables.

- These variables are declared within a class but outside any method, usually at the top of the class.
- They are associated with the instance of the class and are used to store unique data for each object.
- Instance variables are created when an object is instantiated and destroyed when the object is garbage collected.
- They are accessed using object references (e.g., objectName.variableName).

Example:

```
Public class MyClass {
    // Instance variables
    int instanceVar1;
    String instanceVar2;
}
```

- Instance variables are used to represent the state of individual objects and can have different instances of the class.

2. Static Variables:
- Also known as Class Variables.
- There is only one copy of a static variable per class, regardless of how many objects (instances) of that class are created.
- Static variables are declared with the static keyword within a class, usually at the top of the class.
- They are associated with the class itself rather than with any specific instance of the class.
- Static variables are created when the class is loaded into memory and persist as long as the class remains loaded.
- They are accessed using the class name itself (e.g., ClassName.staticVariableName) or through an object reference (though this is discouraged for clarity).

Example:

```
Public class Myclass{
  // Static Variable
  static int staticVar = 0;
}
```

[Q.6]

6. How to create an array ? Explain it with example.

→ In Java, you can create an array by specifying its type, size and optionally initializing its elements.
Here's how you can create an array in Java with an example:

Syntax for array declaration and Initialization:

```
// Declare and create an array of a specific type and size
dataType[] arrayName = new dataType[arraySize];

// Initialize the array elements
arrayName[index] = value;
```

Here's an example of creating and initializing an array of integers:

```
Public class ArrayExample {
    Public static void main(String[] args) {
// Declare and create an array of integers with a size of 5

    int[] numbers = new int[5];

// Initialize the array elements
    numbers[0] = 10;
    Numbers[1] = 20;
    numbers[2] = 30;
    numbers[3] = 40;
    numbers[4] = 50;

// Access and Print array elements
    System.out.println
    ("Element at index 0: " + numbers[0]);
    System.out.println
    ("Element at index 1: " + numbers[1]);
    System.out.println
    ("Element at index 2:" + numbers[2]);
    System.out.println
    ("Element at index 3: " + numbers[3]);
    System.out.println
    ("Element at index 4: " + numbers[4]);
    }
}
```

O/T:-

Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50

[Q. 7]

7. State the difference between break and Continue Statements with suitable Example.

→ In Java, both the break and continue Statements are used to control the flow of loops, but they serve different purposes!

(1) break Statement!
- The break Statement is used to exit a loop prematurely, terminating the loop's execution and continuing with the code immediately after the loop.
- It is commonly used to exit a loop based on a condition without completing all iterations.

Example: Using break to exit a loop early

```
for (int i=1; i<=5; i++) {
    if (i==3) {
        break; // Exit the loop when i is 3
    }
    System.out.println("Iteration" + i);
}
```

O/T:- Interation 1
Interation 2

2. Continue Statement:

- The continue statement is used to skip the current iteration of a loop and proceed to the next iteration.
- It is commonly used when you want to skip specific iterations based on a condition but continue the loop.

Example: Using continue to skip an iteration

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue; // Skip iteration when i is 3
    }
    System.out.println("Iteration " + i);
}
```

O/P:- Iteration 1
Iteration 2
Iteration 4
Iteration 5