Session-2 Interview Theory Questions
(The Structure of Java Programs)

1. What is procedural programming?

Procedural programming is a programming paradigm that is based on the concept of procedures or subroutines. It involves organizing a program into one or more procedures or subroutines, each of which performs a specific task or set of tasks. These procedures or subroutines are called from the main program in a sequential manner, and they can be reused across the program as needed.

In procedural programming, the focus is on solving a problem by breaking it down into smaller, more manageable sub-problems or procedures. Each procedure is designed to perform a specific task or set of tasks, and it may take inputs and produce outputs as necessary.

Procedural programming languages typically include features such as variables, conditional statements (such as if-else statements), loops (such as for and while loops), and functions or procedures.

Some examples of procedural programming languages include C, Fortran, and Pascal. While procedural programming can be a powerful approach for certain types of problems, it can become difficult to maintain and scale as programs become larger and more complex. As a result, object-oriented programming has become a more popular paradigm for software development in recent years.

2. What is the advantage of Object Oriented Language over procedural programming?

Object-Oriented Programming (OOP) offers several advantages over procedural programming, which is a traditional programming paradigm. Here are some of the key advantages of OOP:

- Modularity and Code Reusability
- Encapsulation and Data Hiding (Will be explained later)
- Flexibility
- Simplification:
- Easier Maintenance and Testing:
- Better Problem Solving and Real-World Modeling

OOP allows you to model real-world entities and interactions more accurately, making problem-solving more intuitive and closer to the real-world scenarios. Overall, Object-Oriented Programming offers a more structured, organized, and maintainable

approach to software development compared to procedural programming. It helps create code that is easier to understand, modify, and extend, making it a popular and widely adopted paradigm in modern software development.

3. What is the advantage of using enum in Java?

Using enums in Java provides several advantages that make code more robust, readable, and maintainable. Enums, short for enumerations, are a special data type that allows us to define a set of named constants representing distinct values. Here are the advantages of using enums in Java:

- Readability and Self-Documenting Code:
  Enums allow us to give meaningful names to a set of related constants. By using descriptive names, code becomes more self-documenting, making it easier for developers to understand the purpose and meaning of each constant.
- The use of enums makes the code more expressive and less error-prone. ● Type-Safety - Enums are a type-safe alternative to using integer or string constants. This reduces the risk of runtime errors due to incorrect constant values.
- Compile-Time Checking
- Switch Statements: (Will be described later)
  Enums are particularly useful in switch statements. Using enums in switch statements makes the code more concise and less error-prone, as all enum constants are explicitly accounted for.

4. What are the primitive data types in Java?

Java has eight primitive data types, which are used to store simple values. They are:

- **byte**: Represents a signed 8-bit integer. Range: -128 to 127.
- **short**: Represents a signed 16-bit integer. Range: -32,768 to 32,767. ● **int**: Represents a signed 32-bit integer. Range: -2,147,483,648 to 2,147,483,647. ● **long**: Represents a signed 64-bit integer. Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- **float**: Represents a single-precision 32-bit floating-point number. ● **double**: Represents a double-precision 64-bit floating-point number. ● **boolean**: Represents a boolean value (true or false).
- **char**: Represents a single Unicode character. Range: '\u0000' (0) to '\uffff' (65,535).

5. What is the purpose of JVM in java?

JVM stands for "Java Virtual Machine," and it plays a crucial role in the execution of Java programs. It is a part of the Java Runtime Environment (JRE) and is responsible for converting Java bytecode into machine code that can be understood and executed by the underlying operating system and hardware.

The purpose of JVM (Java Virtual Machine) is to enable the execution of Java bytecode on different platforms in a safe, secure, and efficient manner. Here are the primary uses of JVM in Java:

Platform Independence: One of the most significant advantages of Java is its platform independence. Java programs are compiled into bytecode, which is an intermediate representation that is independent of the underlying hardware and operating system. JVM takes this bytecode and translates it into machine code specific to the host system, enabling Java programs to run on any platform that has a compatible JVM installed.

Memory Management: JVM handles memory allocation and management for Java programs. It automatically takes care of allocating memory for objects, freeing up memory when objects are no longer in use through garbage collection, and managing the stack and heap memory.

Security: JVM provides a secure environment for running Java applications. It uses a security manager that allows fine-grained control over the resources a Java program can access. This sandboxing approach prevents unauthorized access to critical system resources and helps in maintaining the integrity and safety of the host system.

Performance Optimization: JVM includes a Just-In-Time (JIT) compiler that translates bytecode into native machine code on the fly during runtime. This compilation process optimizes the performance of Java programs, as the compiled machine code can be directly executed by the CPU, reducing interpretation overhead.

Exception Handling: JVM provides built-in exception handling mechanisms. It catches and manages exceptions that occur during program execution, ensuring that programs can gracefully handle errors and recover from unexpected situations. (Exception handling will be explained in more detail later.)

Multithreading and Synchronization: JVM supports multithreading, allowing Java programs to execute multiple threads concurrently. It manages thread creation,

synchronization, and context switching, making it easier for developers to write multithreaded applications. (Multithreading will be explained in more detail later.)
Class Loading and Reflection: JVM is responsible for dynamically loading Java classes at runtime through a process called class loading. It also provides reflection capabilities, allowing Java programs to inspect and manipulate classes, methods, and fields dynamically.

Overall, JVM acts as an intermediary layer between the Java code and the host operating system, providing an abstract and consistent environment for Java programs to run efficiently and securely across different platforms.

Following diagram will explain the relationship between JVM, JDK and JRE: