Assignment Questions 1

Q1. What is the difference between Compiler and Interpreter?

ANS. The Compiler and Interpreter, both have similar works to perform. Interpreters and Compilers convert the Source Code (HLL) to Machine Code (understandable by Computer). In general, computer programs exist in High-Level Language that a human being can easily understand. But computers cannot understand the same high-level language, so for computers, we have to convert them into machine language and make them readable for computers. In this article, we are going to see the differences between them.

Compiler

The Compiler is a translator which takes input i.e., High-Level Language, and produces an output of low-level language i.e. machine or assembly language. The work of a Compiler is to transform the codes written in the programming language into machine code (format of 0s and 1s) so that computers can understand.

- A compiler is more intelligent than an assembler it checks all kinds of limits, ranges, errors, etc.
- But its program run time is more and occupies a larger part of memory. It has a slow speed because a compiler goes through the entire program and then translates the entire program into machine codes.

Interpreter

An Interpreter is a program that translates a programming language into a comprehensible language. The interpreter converts high-level language to an intermediate language. It contains pre-compiled code, source code, etc.

- It translates only one statement of the program at a time.
- Interpreters, more often than not are smaller than compilers.

Q2. What is the difference between JDK, JRE, and JVM?

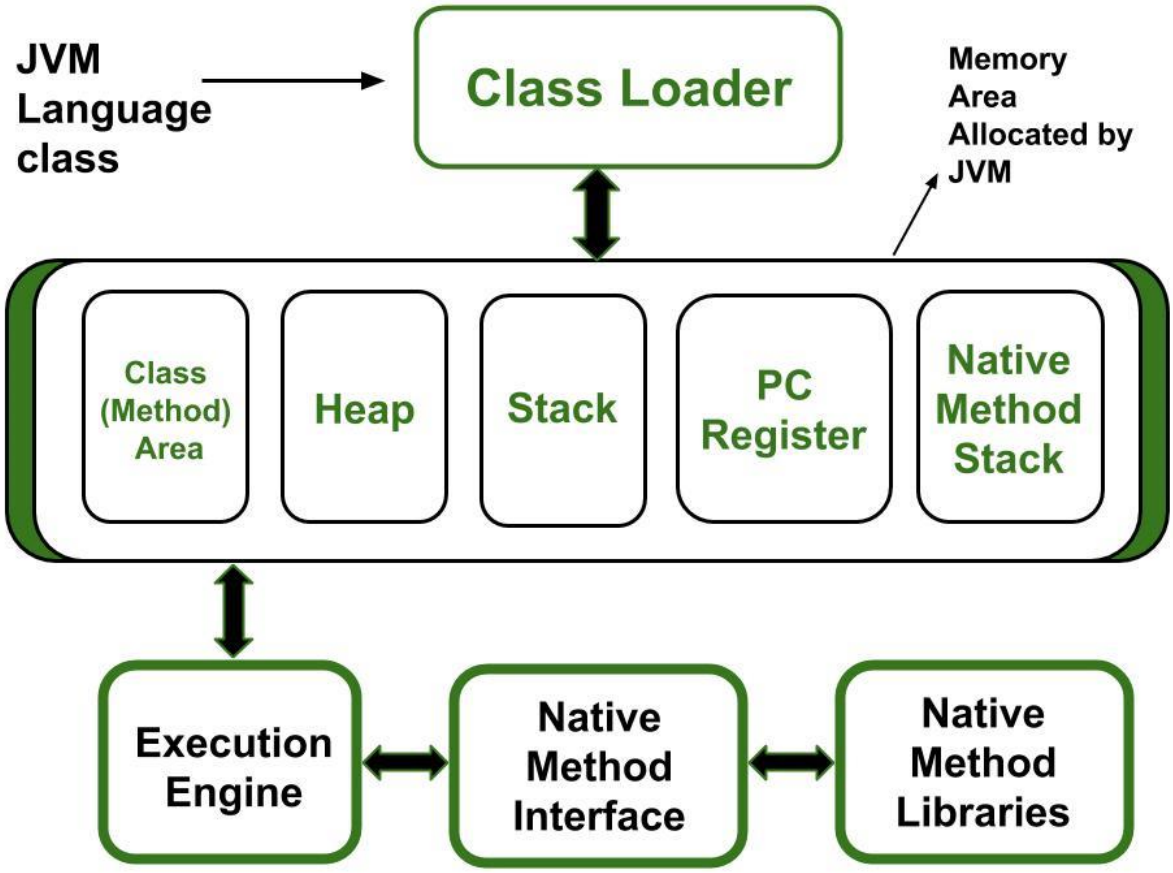
Following are the important differences between JDK, JRE and JVM

Sr. No.	Key	JDK	JRE	JVM
1	Definition	JDK (Java Development Kit) is a software development kit to develop applications in Java. In addition to JRE, JDK also contains number of development tools (compilers, JavaDoc, Java Debugger etc.).	JRE (Java Runtime Environment) is the implementation of JVM and is defined as a software package that provides Java class libraries, along with Java Virtual Machine (JVM), and other components to run applications written in Java programming.	JVM (Java Virtual Machine) is an abstract machine that is platform-dependent and has three notions as a specification, a document that describes requirement of JVM implementation, implementation, a computer program that meets JVM requirements, and instance, an implementation that executes Java byte code provides a runtime environment for executing Java byte code.
2	Prime functionality	JDK is primarily used for code execution and has prime functionality of development.	On other hand JRE is majorly responsible for creating environment for code execution.	JVM on other hand specifies all the implementations and responsible to provide these implementations to JRE.
3	Platform Independence	JDK is platform dependent i.e for different platforms different JDK required.	Like of JDK JRE is also platform dependent.	JVM is platform independent.
4	Tools	As JDK is responsible for prime development so it contains tools for developing, debugging and monitoring java application.	On other hand JRE does not contain tools such as compiler or debugger etc. Rather it contains class libraries and other supporting files that JVM requires to run the program.	JVM does not include software development tools.

Sr. No.	Key	JDK	JRE	JVM
5	Implementation	JDK = Java Runtime Environment (JRE) + Development tools	JRE = Java Virtual Machine (JVM) + Libraries to run the application	JVM = Only Runtime environment for executing the Java byte code.

Q3. How many types of memory areas are allocated by JVM?

Types of Memory Areas Allocated By the JVM:
All these functions take different forms of memory structure. The **memory in the JVM is divided into 5 different parts**:



1. Class(Method) Area
2. Heap
3. Stack
4. Program Counter Register
5. Native Method Stack

Let's see about them in brief:

1. Class (Method) Area

The class method area is the memory block that stores the class code, variable code(static variable, runtime constant), method code, and the constructor of a Java program. (Here method means the function which is written inside the class). It stores class-level data of every class such as the runtime constant pool, field and method data, the code for methods.

2. Heap

The Heap area is the memory block where objects are created or objects are stored. Heap memory allocates memory for class interfaces and arrays (an array is an object). It is used to allocate memory to objects at run time

3. Stack

Each thread has a private JVM stack, created at the same time as the thread. It is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.

Java Stack stores frames and a new frame is created each time at every invocation of the method. A frame is destroyed when its method invocation completes

4. Program Counter Register:

Each JVM thread that carries out the task of a specific method has a program counter register associated with it. The non-native method has a PC that stores the address of the available JVM instruction whereas, in a native method, the value of the program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

5. Native method Stacks:

Also called C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when it's created And it can be of a fixed or dynamic nature.

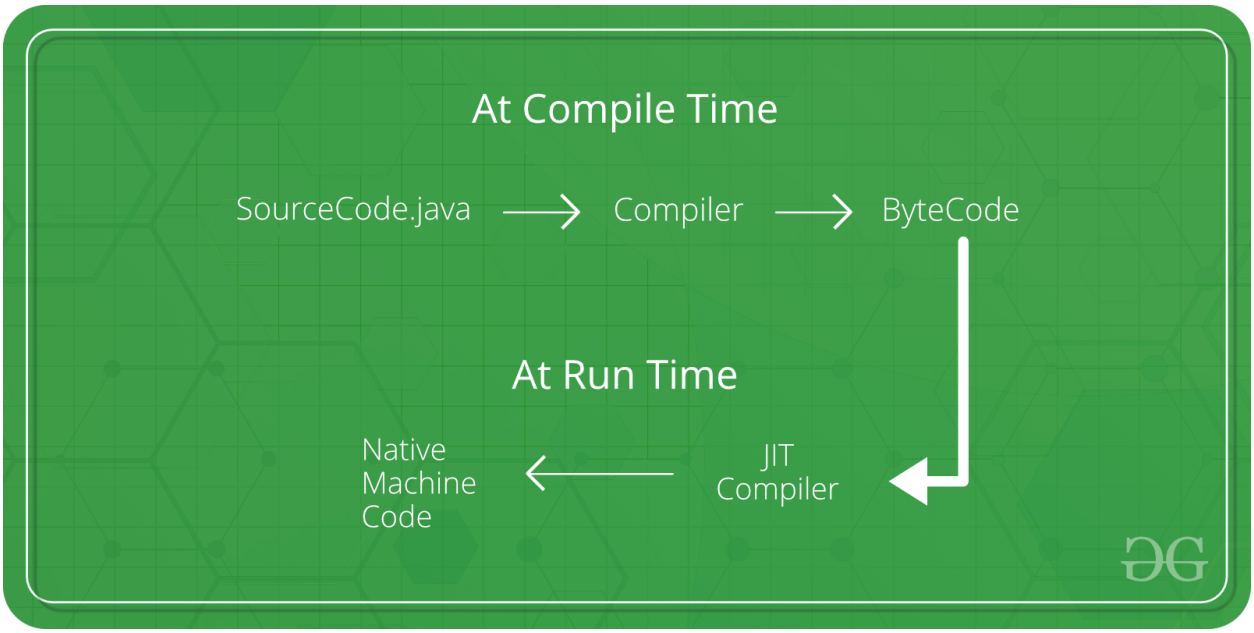
Q4. What is JIT compiler?

The JIT or Just-In-Time compiler is an essential part of the JRE (Java Runtime Environment), that is responsible for performance optimization of java based applications during run time. The compiler is one of the key aspects in deciding the performance of an application for both parties i.e. the end-user and the application developer. Let us check the Just In Time Compiler in Java in more detail.

Java JIT Compiler

Bytecode is one of the most important features of java that aids in cross-platform execution. The way of converting bytecode to native machine language for execution has a huge impact on its speed of it. These bytecodes have to be interpreted or compiled to proper machine instructions depending on the instruction set architecture. Moreover, these can be directly executed if the instruction architecture is bytecode based. Interpreting the bytecode affects the speed of execution. In order to improve performance, JIT compilers interact with the Java Virtual Machine (JVM) at run time and compile suitable bytecode sequences into native machine code. While using a JIT compiler, the hardware is able to execute the native code, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring overhead for the translation process. This subsequently leads to performance gains in the execution speed, unless the compiled methods are executed less frequently.

The JIT compiler is able to perform certain simple optimizations while compiling a series of bytecode to native machine language. Some of these optimizations performed by JIT compilers are data analysis, reduction of memory accesses by register allocation, translation from stack operations to register operations, elimination of common sub-expressions, etc. The greater the degree of optimization done, the more time a JIT compiler spends in the execution stage. Therefore it cannot afford to do all the optimizations that a static compiler is capable of, because of the extra overhead added to the execution time and moreover its view of the program is also restricted.



Working on JIT Compiler

Java follows an object-oriented approach, as a result, it consists of classes. These constitute bytecode that is platform neutral and are executed by the JVM across diversified architectures.

- At run time, the JVM loads the class files, the semantics of each are determined, and appropriate computations are performed. The additional processor and memory usage during interpretation make a Java application perform slowly as compared to a native application.
- The JIT compiler aids in improving the performance of Java programs by compiling bytecode into native machine code at run time.
- The JIT compiler is enabled throughout, while it gets activated when a method is invoked. For a compiled method, the JVM directly calls the compiled code, instead of interpreting it. Theoretically speaking, If compiling did not require any processor time or memory usage, the speed of a native compiler and that of a Java compiler would have been the same.
- JIT compilation requires processor time and memory usage. When the java virtual machine first starts up, thousands of methods are invoked. Compiling all these methods can significantly affect startup time, even if the end result is a very good performance optimization.

Q5. What are the various access specifiers in Java?

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Q6. What is a compiler in Java?

Java compilers are programs that take source code and produce class files containing platform-neutral Java bytecode that can be executed by the Java Virtual Machine (JVM).

Rather than interpret high-level Java source code, the JVM interprets low-level Java bytecode (somewhere between human-readable code and machine code that is specific to a particular computer). Bytecode is platform-neutral and, therefore, can be interpreted by any JVM running on any computer system. This is what makes compiled Java programs portable.

Most Java compilers do little to no optimization of the code, leaving that task for the JVM to do at run time. The JVM loads the bytecode and either interprets it, or just-in-time (JIT) compiles it to machine code, and then optimizes it.

Q7. Explain the types of variables in Java?

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Q8. What are the Datatypes in Java?

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

1. **Primitive Data Type:** such as boolean, char, int, short, byte, long, float, and double
2. **Non-Primitive Data Type or Object Data type:** such as String, Array, etc.

Primitive Data Types in Java

Primitive data are only single values and have no special capabilities. There are 8 primitive data types. They are depicted below in tabular format below as follows:

Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	1 bit	true, false	true, false
byte	twos-complement integer	0	8 bits	(none)	-128 to 127
char	Unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\', '\n', 'β'	characters representation of ASCII values 0 to 255
short	twos-complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos-complement intger	0	32 bits	-2,-1,0,1,2	-2,147,483,648 to 2,147,483,647
long	twos-complement integer	0	64 bits	-2L,-1L,0L,1L,2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Type	Description	Default	Size	Example Literals	Range of values
float	IEEE 754 floating point	0.0	32 bits	1.23e100f , -1.23e-100f , .3f ,3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d , -123456e-300d , 1e1d	upto 16 decimal digits

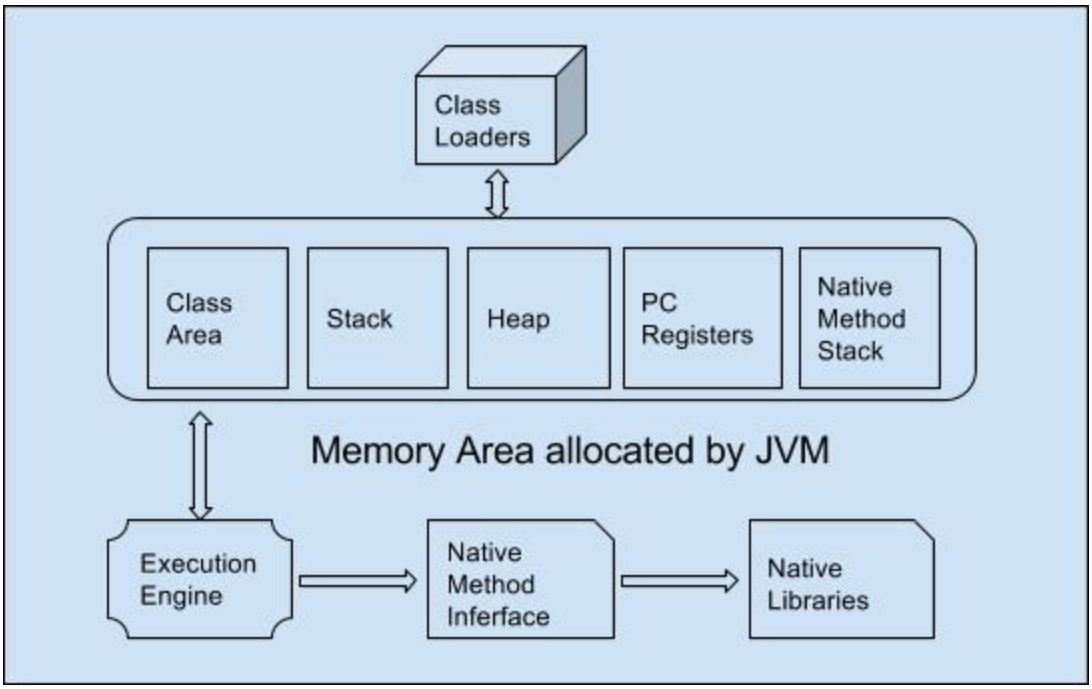
Non-Primitive Data Type or Reference Data Types

The **Reference Data Types** will contain a memory address of variable values because the reference types won't store the variable value directly in memory. They are strings, objects, arrays, etc.

Q9. What are the identifiers in java?

Identifiers in Java are symbolic names used for identification. They can be a class name, variable name, method name, package name, constant name, and more. However, In Java, There are some reserved words that can not be used as an identifier.

Q10. Explain the architecture of JVM



- **Classloader** − Loads the class file into the JVM.
- **Class Area** − Storage areas for a class elements structure like fields, method data, code of method etc.
- **Heap** − Runtime storage allocation for objects.
- **Stack** − Storage for local variables and partial results. A stack contains frames and allocates one for each thread. Once a thread gets completed, this frame also gets destroyed. It also plays roles in method invocation and returns.
- **PC Registers** − Program Counter Registers contains the address of an instruction that JVM is currently executing.
- **Execution Engine** − It has a virtual processor, interpreter to interpret bytecode instructions one by one and a JIT, just in time compiler.
- **Native method stack** − It contains all the native methods used by the application.