Q1.What is the difference between Compiler and Interpreter

What is a Compiler?

A compiler is a translator that produces an output of low-level language (like an assembly or machine language) by taking an input of high-level language. It is basically a computer program used to transform codes written in a programming language into machine code (human-readable code to a binary 0 and 1 bits language for a computer processor to understand). The computer then processes the machine code for performing the corresponding tasks.

- Compilers check all types of errors, limits, and ranges. Thus, it's more intelligent.
- The run time of its program is longer, and it occupies more memory.

What Is an Interpreter?

It is a program that functions for the translation of a programming language into a comprehensible one. It is a computer program used for converting high-level program statements into machine codes. It includes pre-compiled code, source code, and scripts.

- An interpreter translates only one statement at a time of the program.
- They create an exe of the programming language before the program runs.

Difference Between Compiler and Interpreter

Parameter	**Compiler**	**Interpreter**
Steps of Programming	• Creation of the program. • The Compiler analyses all the language statements and throws an error when it finds something incorrect. • If there's zero error, the compiler converts the source code to machine one. • It links various code files into a runnable program (exe). • It runs the program.	• Creation of the program. • It doesn't require the linking of files or generation of machine code. • It executes the source statements line by line during the execution.

Advantage	The code execution time is comparatively less because the program code already gets translated into machine code.	They are fairly easy to use and execute, even for a beginner.
Disadvantage	One can't change a program without getting back to the source code.	Only computers with the corresponding Interpreter can run the interpreted programs.
Machine Code	It stores the machine language on the disk in the form of machine code.	It doesn't save the machine language at all.
Running Time	The compiled codes run comparatively faster.	The interpreted codes run comparatively slower.
Model	It works on the basis of the language-translation linking-loading model.	It works on the basis of the Interpretation method.
Generation of Program	It generates an output program in the exe format. A user can run it independently from the originally intended program.	It doesn't generate an output program. Meaning, it evaluates the source program every time during individual execution.
Execution	One can separate the program execution from the compilation. Thus, you can perform it only after completing the compilation of the entire output.	Execution of the program is one of the steps of the Interpretation process. So, you can perform it line by line.
Memory Requirement	Target programs execute independently. They don't require the Compiler in the memory.	Interpreter originally exists in the memory at the time of interpretation.
Best Fitted For	You cannot port the Compiler because it stays bound to the specific target machine. The compilation model is very common in programming languages like C and C++.	They work the best in web environments- where the load time is very crucial. Compiling takes a relatively long time, even with small codes that may not run multiple times due to the exhaustive analysis. Interpretations are better in such cases.

Optimization of Code	A compiler is capable of seeing the entire code upfront. Thus, it makes the codes run faster by performing plenty of optimizations.	An interpreter sees a code line by line. The optimization is, thus, not very robust when compared to Compilers.
Dynamic Typing	Compilers are very difficult to implement because they can't predict anything that happens during the turn time.	The Interpreted language supports Dynamic Typing.
Use	It works best for the Production Environment.	It works the best for the programming and development environment.
Execution of Error	A Compiler displays every error and warning while compiling. So, you can't run this program unless you fix the errors.	An Interpreter reads every statement, then displays the errors, if any. A user must resolve these errors in order to interpret the next line.
Input	A Compiler takes a program as a whole.	An Interpreter takes single lines of a code.
Output	The Compilers generate intermediate machine codes.	The Interpreters never generate any intermediate machine codes.
Errors	This translator displays all the errors after compiling- together at the same time.	It displays the errors of every single line one by one.
Programming Languages	Java, Scala, C#, C, C++ use Compilers.	Perl, Ruby, PHP use Interpreters.

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

What is JDK?

JDK is an abbreviation for Java Development Kit. It is an environment of software development used for developing applets and Java applications. JDK has a physical existence, and it contains JRE + development tools. One can easily install more than one version of JDK on the same computer. The Java developers can make use of it on macOS, Windows, Linux, and Solaris. JDK assists them in coding and running the Java programs.

It is an implementation of any of the given Java Platforms that the Oracle Corporation released:

- Micro Edition
- Enterprise Edition
- Standard Edition

The JDK consists of a private JVM (Java Virtual Machine) along with a few other resources, java (a loader/interpreter), like javac (a compiler), Javadoc (a documentation generator), jar (an archiver), etc., for completing the process of Java application development.

What is JRE?

JRE stands for Java Runtime Environment- also written as Java RTE. It is a set of software tools designed for running other software. It is an implementation of JVM, and JRE provides a runtime environment. In short, a user needs JRE to run any Java program. If not a programmer, the user doesn't need to install the JDK- JRE alone will help run the Java programs.

All the versions of JDK come bundled up with the JRE (Java Runtime Environment). This way, a user doesn't have to download and install JRE on their PC separately. The JRE also exists physically. It consists of a library set + a few more files that the JVM (Java Virtual Machine) deploys at the runtime.

What is JVM?

JVM stands for Java Virtual Machine. It provides a runtime environment for driving Java applications or code. JVM is an abstract machine that converts the Java bytecode into a machine language. It is also capable of running the programs written by programmers in other languages (compiled to the Java bytecode). The JVM is also known as a virtual machine as it does not exist physically.

JVM is essentially a part of the JRE (Java Run Environment). You cannot separately download and install it. You first need to install the JRE to install the JVM. It is available for many software and hardware platforms. In various distinct programming languages, the compiler functions to produce machine code for specific systems. However, only the Java compiler produces code for a virtual machine- also known as JVM.

All three, JDK, JRE, and JVM, are dependent. It is because each Operating System's (OS) condition is different from one another. But Java is independent of the platform. The JVM has three notions: *implementation*, *instance,* and *specification*.

JVM primarily performs the following tasks:

- Provides runtime environment
- Verifies code
- Loads code
- Executes code

Difference Between JDK, JRE, and JVM

Parameter	**JDK**	**JRE**	**JVM**
Full-Form	The JDK is an abbreviation for Java Development Kit.	The JRE is an abbreviation for Java Runtime Environment.	The JVM is an abbreviation for Java Virtual Machine.
Definition	The JDK (Java Development Kit) is a software development kit that develops applications in Java. Along with JRE, the JDK also consists of various development tools (Java Debugger, JavaDoc, compilers, etc.)	The Java Runtime Environment (JRE) is an implementation of JVM. It is a type of software package that provides class libraries of Java, JVM, and various other components for running the applications written in Java programming.	The Java Virtual Machine (JVM) is a platform-independent abstract machine that has three notions in the form of specifications. This document describes the requirement of JVM implementation.
Functionality	The JDK primarily assists in executing codes. It primarily functions in development.	JRE has a major responsibility for creating an environment for the execution of code.	JVM specifies all of the implementations. It is responsible for providing all of these implementations to the JRE.
Platform Dependency	The JDK is platform-dependent. It means that for every different platform, you require	JRE, just like JDK, is also platform-dependent. It means that for every different platform, you	The JVM is platform-independent. It means that you won’t require a different JVM for

	a different JDK.	require a different JRE.	every different platform.
Tools	Since JDK is primarily responsible for the development, it consists of various tools for debugging, monitoring, and developing java applications.	JRE, on the other hand, does not consist of any tool- like a debugger, compiler, etc. It rather contains various supporting files for JVM, and the class libraries that help JVM in running the program.	JVM does not consist of any tools for software development.
Implementation	**JDK** = Development Tools + JRE (Java Runtime Environment)	**JRE** = Libraries for running the application + JVM (Java Virtual Machine)	**JVM** = Only the runtime environment that helps in executing the Java bytecode.
Why Use It?	Why use JDK? Some crucial reasons to use JDK are: • It consists of various tools required for writing Java programs. • JDK also contains JRE for executing Java programs. • It includes an Appletviewer, Java application launcher, compiler, etc. • The compiler	Why use JRE? Some crucial reasons to use JRE are: • If a user wants to run the Java applets, then they must install JRE on their system. • The JRE consists of class libraries along with JVM and its supporting files. It has no other tools like a compiler or a debugger for Java development.	Why use JVM? Some crucial reasons to use JVM are: • It provides its users with a platform-independent way for executing the Java source code. • JVM consists of various tools, libraries, and multiple frameworks. • The JVM also comes with a Just-in-Time (JIT) compiler

	helps in converting the code written in Java into bytecodes. • The Java application launcher helps in opening a JRE. It then loads all of the necessary details and then executes all of its main methods.	• JRE uses crucial package classes like util, math, awt, lang, and various runtime libraries.	for converting the Java source code into a low-level machine language. Thus, it ultimately runs faster than any regular application. • Once you run the Java program, you can run JVM on any given platform to save your time.
Features	Features of JDK • Here are a few crucial features of JDK: • It has all the features that JRE does. • JDK enables a user to handle multiple extensions in only one catch block. • It basically provides an environment for developing and executing the Java source code. • It has various development	Features of JRE • Here are a few crucial features of JRE: • It is a set of tools that actually helps the JVM to run. • The JRE also consists of deployment technology. It includes Java Plug-in and Java Web Start as well. • A developer can easily run a source code in JRE. But it does not allow them to write and	Features of JVM Here are a few crucial features of JVM: • The JVM enables a user to run applications on their device or in a cloud environment. • It helps in converting the bytecode into machine-specific code. • JVM also provides some basic Java functions, such as garbage collection, security,

	tools like the debugger, compiler, etc. • One can use the Diamond operator to specify a generic interface in place of writing the exact one. • Any user can easily install JDK on Unix, Mac, and Windows OS (Operating Systems).	compile the concerned Java program. • JRE also contains various integration libraries like the JDBC (Java Database Connectivity), JNDI (Java Naming and Directory Interface), RMI (Remote Method Invocation), and many more. • It consists of the JVM and virtual machine client for Java HotSpot.	memory management, and many more. • It uses a library along with the files given by JRE (Java Runtime Environment) for running the program. • Both JRE and JDK contain JVM. • It is easily customizable. For instance, a user can feasibly allocate a maximum and minimum memory to it. • JVM can also execute a Java program line by line. It is thus also known as an interpreter. • JVM is also independent of the OS and hardware. It means that once a user writes a Java program, they can easily run it anywhere.

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

JVM (Java Virtual Machine) is an abstract machine, In other words, it is a program/software which takes Java bytecode and converts the byte code (line by line) into machine understandable code.

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in Java code. JVM is a part of the JRE(Java Runtime Environment).
JVM perform some particular **types of operations**:
1. Loading of code
2. Verification of code
3. Executing the code
4. It provides a run-time environment to the users

ClassLoader
It is a subsystem of JVM which is used to load class files. It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

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?

What are Access Modifiers?

Access modifiers are keywords that can be used to control the visibility of fields, methods, and constructors in a class. The four access modifiers in Java are public, protected, default, and private.

Four Types of Access Modifiers

- **Private**: We can access the **private modifier** only within the same class and not from outside the class.

- **Default:** We can access the **default modifier** only within the same package and not from outside the package. And also, if we do not specify any access modifier it will automatically consider it as default.
- Protected: We can access the protected modifier within the same package and also from outside the package with the help of the child class. If we do not make the child class, we cannot access it from outside the package. So inheritance is a must for accessing it from outside the package.
- Public: We can access the public modifier from anywhere. We can access public modifiers from within the class as well as from outside the class and also within the package and outside the package.

Q6.What is a compiler in Java?

A Java compiler is a program that takes the text file work of a developer and compiles it into a platform-independent Java file. Java compilers include the Java Programming Language Compiler (javac), the GNU Compiler for Java (GCJ), the Eclipse Compiler for Java (ECJ) and Jikes.

Programmers typically write language statements in a given programming language one line at a time using a code editor or an integrated development environment (IDE). The resulting file contains what are called the source statements. The programmer then runs a compiler for the appropriate language, specifying the name of the file that contains the source statements.

At run time, the compiler first parses (analyzes) all of the language statements syntactically and then, in one or more successive stages or "passes," builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

Generally, Java compilers are run and pointed to a programmer's code in a text file to produce a class file for use by the Java virtual machine (JVM) on different platforms. Jikes, for example, is an open source compiler that works in this way.

A just-in-time (JIT) compiler comes along with the Java VM. Its use is optional, and it is run on the platform-independent code. The JIT compiler then translates the code into the machine code for different hardware so that it is optimized for different architectures. Once the code has been (re-)compiled by the JIT compiler, it will usually run more quickly than the Java code that can only be executed one instruction at a time.

Q7.Explain the types of variables in Java?

Java Variables

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.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Q8.What are the Datatypes in Java?

What are Data Types 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

Type	Description	Default	Size	Example Literals	Range of 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
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

Let us discuss and implement each one of the following data types that are as follows:

1. Boolean Data Type

Boolean data type represents only one bit of information **either true or false** which is intended to represent the two truth values of logic and Boolean algebra, but the size of the boolean data type is **virtual machine-dependent**. Values of type boolean are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.

Syntax:

boolean booleanVar;

Size: Virtual machine dependent

2. Byte Data Type

The byte data type is an 8-bit signed two's complement integer. The byte data type is useful for saving memory in large arrays.

Syntax:
byte byteVar;

Size: 1 byte (8 bits)

3. Short Data Type

The short data type is a 16-bit signed two's complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.

Syntax:
short shortVar;

Size: 2 bytes (16 bits)

4. Integer Data Type

It is a 32-bit signed two's complement integer.

Syntax:
int intVar;

Size: 4 bytes (32 bits)

Remember: *In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has a value in the range [0, 2^{32}-1]. Use the Integer class to use the int data type as an unsigned integer.*

5. Long Data Type

The range of a long is quite large. The long data type is a 64-bit two's complement integer and is useful for those occasions where an int type is not large enough to hold the desired value. The size of the Long Datatype is 8 bytes (64 bits).

Syntax:
long longVar;

Remember: *In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2^{64}-1. The Long class also contains methods like comparing Unsigned, divide Unsigned, etc to support arithmetic operations for unsigned long.*

6. Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).

Syntax:
float floatVar;

7. Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The size of the double data type is 8 bytes or 64 bits.

Syntax:
double doubleVar;

Note: *Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable. If accuracy is the most prior concern then, it is recommended not to use these data types and use BigDecimal class instead.*
It is recommended to go through *rounding off errors in java.*

8. Char Data Type

The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).

Syntax:
char charVar;

Q9.What are the identifiers in java?

Identifiers in Java are names that are helpful in **uniquely recognizing** a class, a method, a package name, a constant, or a variable. Some words in Java are reserved and cannot be used as identifiers. Certain rules must be followed in Java while we are defining an identifier, or the compiler will throw an error.

See Also: Complete Java Tutorial

Introduction to Identifiers in Java

In general terms, an identifier is a name given to an entity for identification. Java identifiers are any attribute/property of an entity that is utilized for identification. For example, a person's name, an employee's employee number, or an individual's social security number.

Identifiers in Java are names that distinguish between different Java entities, such as classes, methods, variables, and packages. Identifiers include the names of classes, methods, variables, packages, constants, etc. These identifiers are each specified using a specific syntax and naming scheme.

The syntax would depend on the type of identifier. For example, an integer variable in Java can be declared as below,

int <variable name(identifier)>;

The variable name is the name given by the programmer to uniquely identify the int variable. Later the programmer can use that variable in his program. Hence, here the variable name is the identifier. These Identifiers follow a certain naming convention/rules when they are named, which we shall look at in the next section of this article.

Example:

Identifiers: Below is the list of identifiers that are present in the above sample code.

- MainClass (Class name)
- main (Method name)
- String (Predefined Class name)
- args (String variable name)
- var1 (integer variable name)
- var2 (double variable name)
- System(Predefined Class name)
- out(Variable name)
- println (Method name)

Rules for Identifiers in Java

Below are the rules that need to be followed when defining an identifier in Java. A compile-time error will be produced if any of the following rules are broken.

Rule 1:

Identifiers can only contain alphanumeric characters [a-z] [A-Z] [0-9] , dollar sign ($) and underscore (_). No other character is allowed. **Valid Examples:**

Identifier	**Explanation**
NomansLand90	This is valid as it contains only alphanumerics
$Wink1NTomBoy	This is valid as characters are alphanumerics, and $

Invalid Examples:

Identifier	**Explanation**
wiseWizard#1994	This is invalid as # character is not allowed

Rule 2:

Identifiers cannot start with a numeric value [0-9]. The starting character should be alphabet [A-Z] [a-z], dollar ($) or underscore (_).

Valid Example:

Identifier	**Explanation**
ScalerAcademy	This is valid as the name starts with an alphabet.

Invalid Example:

Identifier	**Explanation**
1ScalerAcademy	This is invalid as the name starts with a numeric value 1.

Rule 3:

Identifiers should not contain spaces in their name.

Valid Example:

Identifier	**Explanation**
Four_HorseMen$2021	This is valid as characters are alphanumerics, $

Invalid Example:

Identifier	**Explanation**
Four HorseMen$2021	This is invalid as there is space after Four

Rule 4:

Java identifiers are case-sensitive.

Eg: ‘Vendetta’ and ‘vendetta’ are considered as two different identifiers.

Rule 5:

The standard convention sets the size of the Java identifiers between 4 – 15, and it is advised to follow that length when defining an identifier. However, there is no upper limit on the length of an identifier.

Rule 6:

Reserve Keywords cannot be used as Identifiers. Java defines a total of 53 reserved keywords that are predefined. In the following part, we'll examine what a reserved keyword is and a list of predefined terms.

Example: int, float, public, static, etc.

Java Reserved Keywords

Reserved keywords are keywords that are used by java syntax for a **particular functionality**. Since each of these reserved keywords' functionality is predefined, these keywords cannot be used for any other purpose.

Java predefines a set of 53 reserved keywords that cannot to used as Identifiers. So these keywords cannot be used as Class names, Method names, Package names, or Variable names. We have some of these keywords in our sample program in the introduction. int, double, public, static, etc., are all examples of reserved keywords.

Example 1: int Var1; → Here, int is a reserved keyword that is used to define an integer variable. The variable name is Var1, which is an identifier for an int variable.

Example 2: int break; → Here, int is a reserved keyword, and the variable name is break, which is invalid as break itself is a reserved keyword with its predefined functionality in java. Hence this statement will throw an error.

Below is the table of references for all the 53 keywords in Java:

Keywords

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient

Keywords

byte	enum	int	return	true
catch	extends	interface	short	try
char	false	instanceof	static	void
class	final	long	strictfp	volatile
const	finally	native	super	while
continue	float	new	switch	
case	for	null	synchronized	

Valid Identifiers in Java

Below is an example list of valid Identifiers that can be used in java

Identifier	Explanation
Employee	alphabets
EMP12	alphanumerics
$Manager1	alphanumerics and $
_AngryMonk404	alphanumerics and _
Student36Pro9	alphanumerics
A	alphabet uppercase A
i	alphabet lowercase i
$	Symbol $
final_result_value	alphabets and _
SevenUp___7	alphanumerics and _

Invalid Identifiers in Java

Below is an example list of Identifiers that are invalid in java.

Identifier	**Explanation**
@Employee	contains invalid character @
12EMP	Starts with numerics
&Manager1	contains invalid character &
^AngryMonk404	contains invalid character ^
36-StudentPro9	Starts with numerics and contains invalid character –
5	Is a numeric
final result value	Contains spaces

Q10.Explain the architecture of JVM

JVM (Java Virtual Machine) Architecture

1. Java Virtual Machine
2. Internal Architecture of JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

The JVM performs following operation:

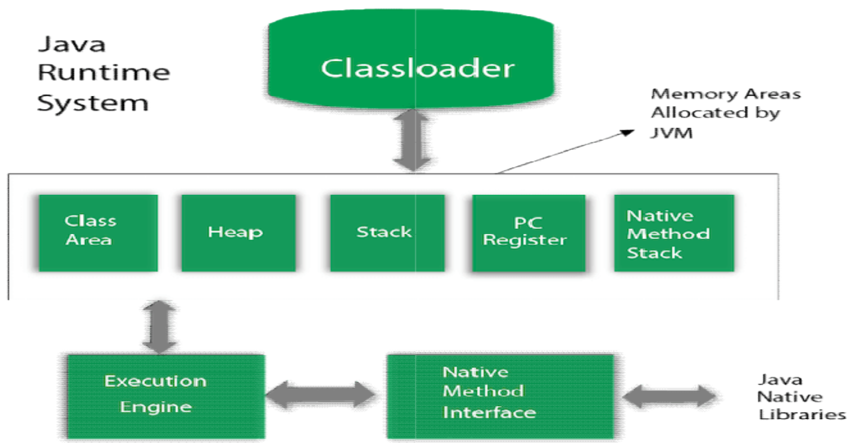
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.