**Q. What is Java? Why Java is called a platform?**

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

**Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called a platform.

For eg: For running source code of C or C++, the hardware + OS are the necessary, so these make up the platform, but for java hardware + OS + JRE is the necessary, so these makes up the platform.

**Differences between the Java platform and other platforms:**

The main differences between the Java platform and other platforms are:

1. [Java is **platform independent**](https://www.geeksforgeeks.org/java-platform-independent/) because of this characteristic we can write Java code in one platform and can be read/run in/on any other platform i.e [WORA(Write Once Read Anywhere)](https://www.geeksforgeeks.org/why-is-java-write-once-and-run-anywhere/). Other languages lack this capability.
2. Java platform is a **software-only platform** that runs on the top of other hardware-based platforms, other platforms are mostly hardware software or hardware only and can be run only on hardware based.
3. Programmer can develop Java code on **any OS**. Most of the other platforms do not have this capability.
4. Java has its own run time environment known as [JRE(Java Run-time Environment)](https://www.geeksforgeeks.org/differences-jdk-jre-jvm/) and [Java Virtual Machine(JVM)](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) which converts Java code to machine code, whereas this functionality is missing in other platforms.

**Q2. What is package in Java? List down various advantages of packages?**

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Advantages of Packages:

Often, it is a hierarchical structure of storing information. It is easier to organize the related classes and subpackages in this manner. A Package also provides access protection for classes and interfaces. A package also helps in removing naming collision.

Packages are used for:

* Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
* Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
* Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
* Packages can be considered as data encapsulation (or data-hiding).

**Built-in Packages**  
These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:  
1) **java.lang:**Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.  
2) **java.io:**Contains classed for supporting input / output operations.  
3) **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.  
4) **java.applet:**Contains classes for creating Applets.  
5) **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).  
6) **java.net:**Contain classes for supporting networking operations.

**Q. JDK : Java Development Kit**

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment that offers a collection of tools and libraries used to develop java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:

* Standard Edition Java Platform
* Enterprise Edition Java Platform
* Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.

**Components of JDK**

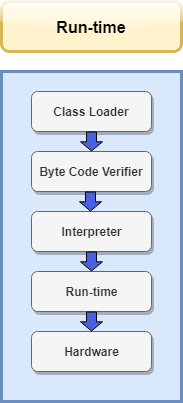
|  |  |
| --- | --- |
| **appletviewer:** | This tool is used ***to run and debug Java applets without a web browser.*** |
| **apt:** | It is an annotation-processing tool. |
| **extcheck:** | it is a utility that detects JAR file conflicts. |
| **idlj:** | An IDL-to-Java compiler. This utility generates Java bindings from a given Java IDL file. |
| **jabswitch:** | It is a Java Access Bridge. Exposes assistive technologies on Microsoft Windows systems. |
| **java:** | ***The loader*** for Java applications. This tool is ***an interpreter*** and can interpret the class files generated by the javac compiler. ***Now a single launcher is used for both development and deployment.*** The old deployment launcher, jre, no longer comes with Sun JDK, and instead it has been replaced by this new java loader. |
| **javac:** | It specifies the ***Java compiler***, which converts source code into Java bytecode. |
| **javadoc:** | The ***documentation generator***, which automatically generates documentation ***from source code comments*** |
| **jar:** | The specifies ***the archiver***, which ***packages related class libraries into a single JAR file.*** This tool also helps manage JAR files. |
| **javafxpackager:** | It is a tool to package and sign JavaFX applications. |
| **jarsigner:** | the jar signing and verification tool. |
| **javah:** | the C header and stub generator, used to write native methods. |
| **javap:** | the class file disassembler. |
| **javaws:** | the Java Web Start launcher for JNLP applications. |
| **JConsole:** | Java Monitoring and Management Console. |
| **jdb:** | ***the debugger.*** |
| **jhat:** | Java Heap Analysis Tool (experimental). |
| **jinfo:** | This utility gets configuration information from a running Java process or crash dump. |
| **jmap:** | Oracle jmap - Memory Map- This utility outputs the memory map for Java and can print shared object memory maps or heap memory details of a given process or core dump. |
| **jmc:** | Java Mission Control |
| **jps:** | Java Virtual Machine Process Status Tool lists the instrumented HotSpot Java Virtual Machines (JVMs) on the target system. |
| **jrunscript:** | Java command-line script shell. |
| **jstack:** | It is a utility that prints Java stack traces of Java threads (experimental). |
| **jstat:** | Java Virtual Machine statistics monitoring tool (experimental). |
| **jstatd:** | jstat daemon (experimental). |
| **keytool:** | It is a tool for manipulating the keystore. |
| **pack200:** | JAR compression tool. |
| **Policytool:** | It specifies the policy creation and management tool, which can determine policy for a Java runtime, specifying which permissions are available for code from various sources. |
| **VisualVM:** | It is a visual tool integrating several command-line JDK tools and lightweight [clarification needed] performance and memory profiling capabilities |
| **wsimport:** | It generates portable JAX-WS artifacts for invoking a web service. |
| **xjc:** | It is the part of the Java API for XML Binding (JAXB) API. It accepts an XML schema and generates Java classes. |

**Q.JRE : JAVA RUNTIME ENVIRONMENT**

Java Run-time Environment (JRE) is the part of the Java Development Kit (JDK). It is a freely available ***software distribution which has Java Class Library, specific tools, and a stand-alone JVM.*** **It is the most common environment available on devices to run java programs.** The source Java code gets compiled and converted to Java bytecode. If you wish to run this bytecode on any platform, you require JRE. **The JRE loads classes, verify access to memory, and retrieves the system resources. JRE acts as a layer on the top of the operating system.**

JRE has an instance of JVM with it, library classes and development tools.

Once you write this program, you have to save it with .java extension. Compile your program. The output of the Java compiler is a byte-code which is platform independent. After compiling, the compiler generates a .class file which has the bytecode. The bytecode is platform independent and runs on any device having the JRE. **From here, the work of JRE begins**. To run any Java program, you need JRE. The flow of the bytecode to run is as follows:



The following steps take place at runtime:

* **Class Loader**  
  At this step, the class loader loads various classes which are essential for running the program. The class loader dynamically loads the classes in the Java Virtual Machine.  
  When the JVM is started, three class loaders are used:
  1. Bootstrap class loader
  2. Extensions class loader
  3. System class loader
* **Byte code verifier**  
  Byte code verifier can be considered as a gatekeeper. It verifies the bytecode so that the code doesn't make any sort of disturbance for the interpreter. The code is allowed to be interpreted only when it passes the tests of the Bytecode verifier which checks the format and checks for illegal code.
* **Interpreter**  
  Once the classes get loaded and the code gets verified, then interpreter reads the assembly code line by line and does the following two functions:
  1. Execute the Byte Code
  2. Make appropriate calls to the underlying hardware

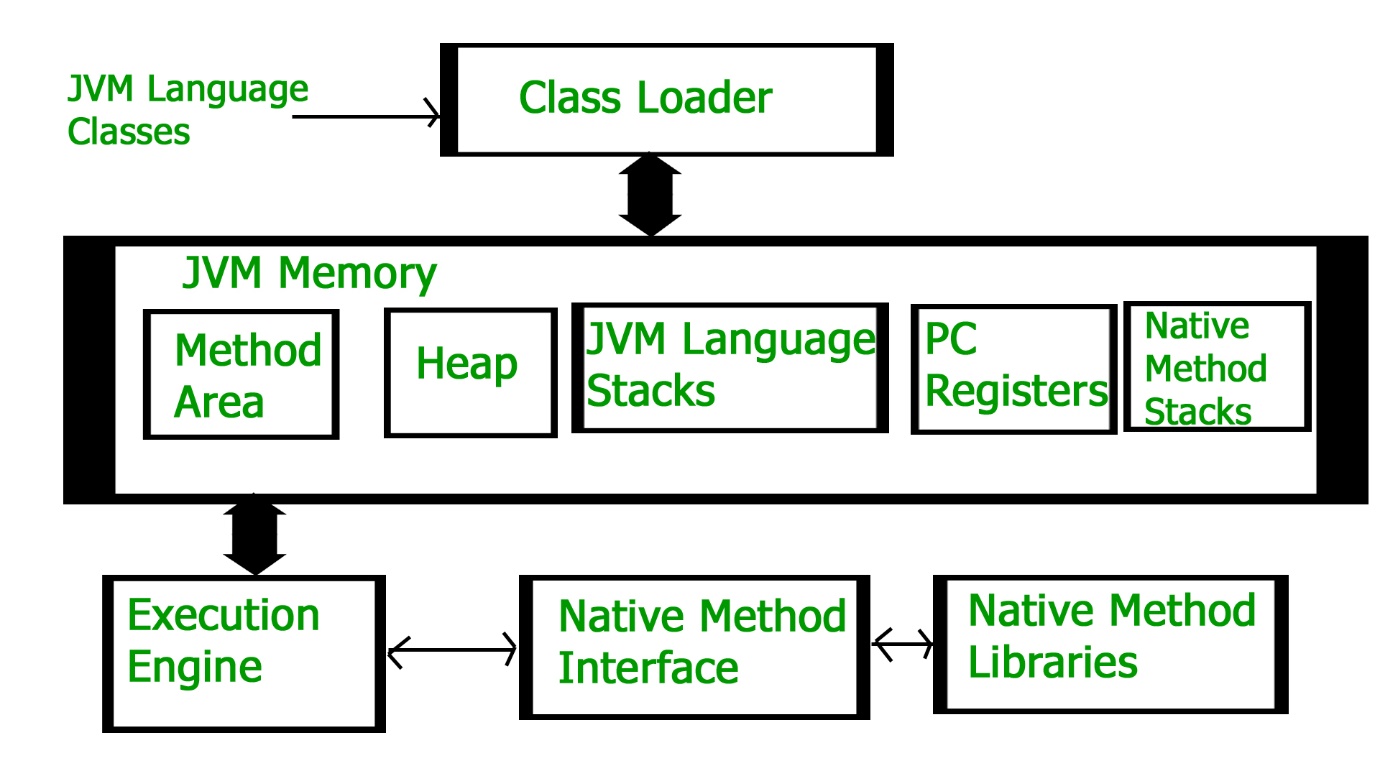
**Q.JVM**

JVM (Java Virtual Machine) is an **abstract machine. It is called a virtual machine because it doesn't physically exist**. It is a specification that provides a runtime environment in which Java bytecode can be executed. **It can also run those programs which are written in other languages and compiled to Java bytecode.**

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 a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.



**Q.JVM Architecture**

**Class Loader Subsystem**

It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

**Loading:** The Class loader reads the “.*class”* file, generate the corresponding binary data and save it in the method area.

**Linking:** Performs verification, preparation, and (optionally) resolution. 

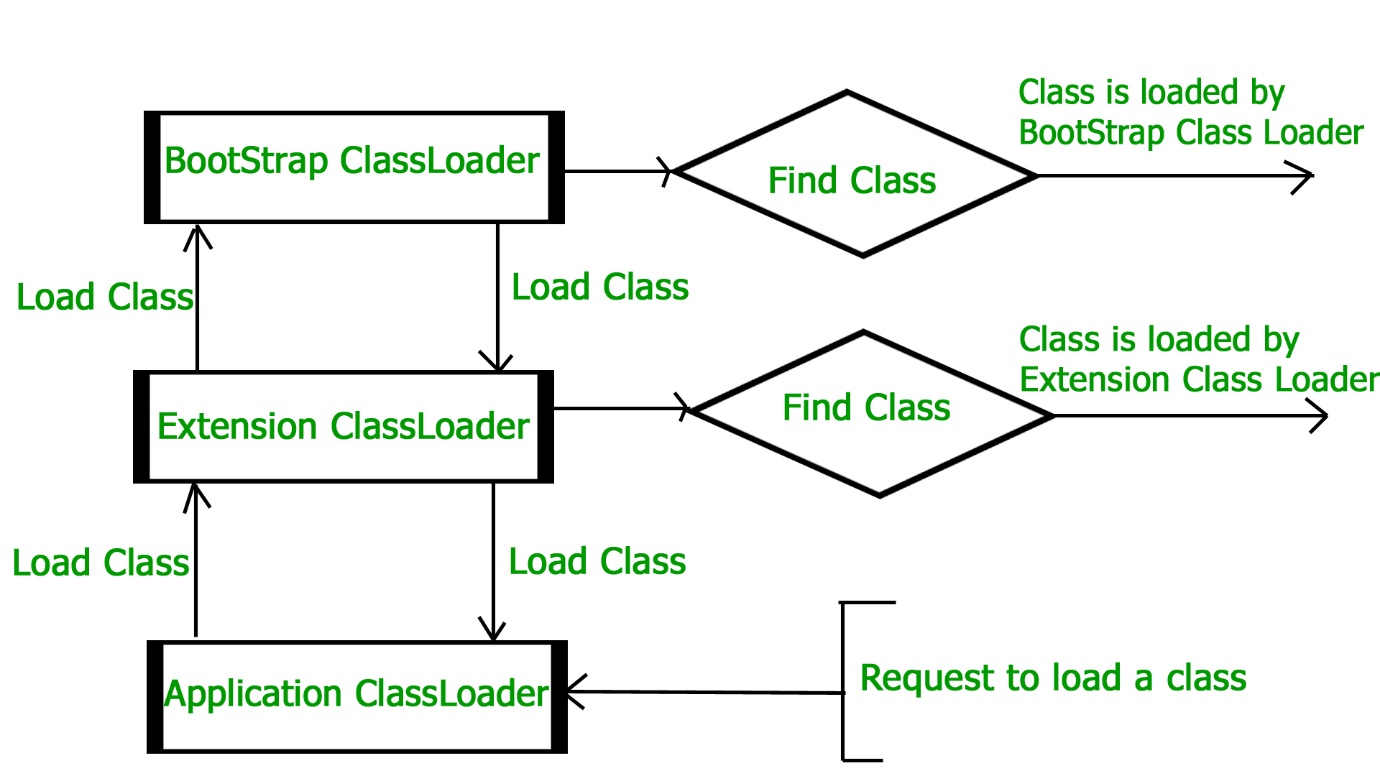
* *Verification*: It ensures the correctness of the *.class* file i.e. ***it checks whether this file is properly formatted and generated by a valid compiler or not.*** If verification fails, we get run-time exception *java.lang.VerifyError*. ***This activity is done by the component ByteCodeVerifier.*** Once this activity is completed then the class file is ready for compilation.
* *Preparation*: JVM ***allocates memory*** for class variables and initializing the memory to default values.
* *Resolution*: It is the process of ***replacing symbolic references from the type with direct references***. It is done by searching into the method area to locate the referenced entity.

**Initialization:** In this phase***, all static variables are assigned with their values*** defined in the code and static block(if any). ***This is executed from top to bottom in a class and from parent to child in the class hierarchy.***

In general, there are three class loaders : 

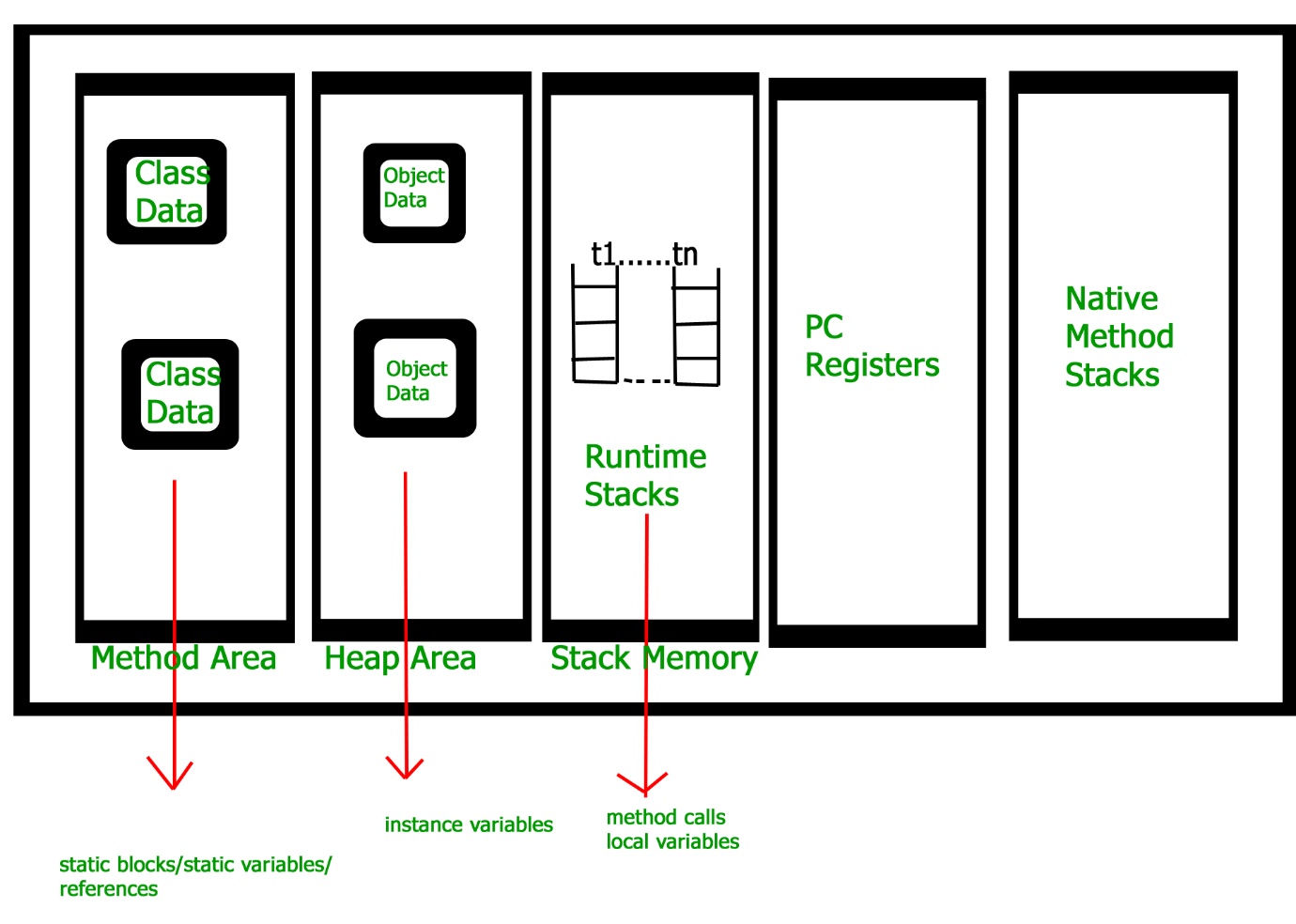
* ***Bootstrap class loader*:** Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes***. It loads core java API classes*** present in the “*JAVA\_HOME/jre/lib”* directory. ***This path is popularly known as the bootstrap path.*** It is implemented in native languages like C, C++.
* ***Extension class loader*:** It is a child of the bootstrap class loader. It ***loads the classes present in the extensions directories “JAVA\_HOME/jre/lib/ext”(Extension path)*** or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
* ***System/Application class loader*:** It is a child of the extension class loader. It is responsible to ***load classes from the application classpath***. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader* class.

**Note:**JVM follows the ***Delegation-Hierarchy principle*** to load classes. ***System class loader delegate load request to extension class loader and extension class loader delegate request to the bootstrap class loader.*** If a class found in the boot-strap path, the class is loaded otherwise request again transfers to the extension class loader and then to the system class loader. ***At last, if the system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException.***



**JVM Memory**

1. **Method area:** In the method area, all ***class level information*** like class name, immediate parent class name, methods and variables information etc. are stored***, including static variables***. ***There is only one method area per JVM, and it is a shared resource.***
2. **Heap area:** Information of ***all objects*** is stored in the heap area. ***There is also one Heap Area per JVM. It is also a shared resource.***
3. **Stack area**: ***For every thread, JVM creates one run-time stack which is stored here.*** Every block of this stack is called activation record/stack frame which ***stores methods calls. All local variables of that method are stored in their corresponding frame.*** After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
4. **PC Registers:** Store ***address of current execution instruction*** of a thread. Obviously*,* ***each thread has separate PC Registers.***
5. **Native method stacks:** For every thread, a separate native stack is created. ***It stores native method information.***



**Execution Engine**

Execution engine executes the “*.class”* (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

* *Interpreter****: It interprets the bytecode line by line and then executes***. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* *Just-In-Time Compiler (JIT)*: It is used to increase the efficiency of an interpreter. ***It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.***
* *Garbage Collector*: It ***destroys un-referenced objects***. For more on Garbage Collector, refer [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/).

**Java Native Interface (JNI) :**

It is an ***interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution.*** It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries :**

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

**Q. What are the features of Java?**

There are the following features in Java Programming Language.

**Simple:**

 Java is easy to learn. The syntax of Java is based on C++ which makes easier to write the program in it.

According to Sun Microsystem, Java language is a simple programming language because:

* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

**Object-Oriented:**

Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behaviour.

Basic concepts of OOPs are:

1. [Object](https://www.javatpoint.com/object-and-class-in-java)
2. [Class](https://www.javatpoint.com/object-and-class-in-java#class)
3. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
4. [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
5. [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
6. [Encapsulation](https://www.javatpoint.com/encapsulation)

**Platform Independent / Portability:**

Java is platform independent because it is different from other languages like [C](https://www.javatpoint.com/c-programming-language-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

**Secure :**

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

* **No explicit pointer**
* **Java Programs run inside a virtual machine sandbox**



* **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. ***It adds security by separating the package for the classes of the local file system from those that are imported from network sources.***
* **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
* **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

**Robust::**

The English mining of Robust is strong. Java is robust because:

* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

**Architectural Neutral ::**

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

**High Performance:**

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).

**Multithreaded:**

We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

**Distributed:**

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

**Dynamic:**

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

**Extras ::**

**Dynamic and Non-Dynamic Examples**  
Numerous languages fall into the dynamic category, including JavaScript, VBScript, Lisp, Perl, PHP, Python, Ruby and Smalltalk. Examples of languages that are **not dynamic** are C/C++, Java, COBOL and FORTRAN.

**Is Java a dynamic programming language?**

**Java is considered to be more dynamic than C or C++** since it is designed to adapt to an evolving environment. Java programs can carry an extensive amount of run-time information that can be used to verify and resolve accesses to objects at run-time.

**What means dynamic language?**

Dynamic programming language is a term used broadly in computer science to describe **a class of high level programming languages that execute at runtime many common behaviors that other languages might perform during compilation, if at all**.

**Is a dynamic language?**

The most on demand dynamic languages are **Python, Perl and Ruby**. Dynamic languages allow programmers to modify even change the structure while running. It is known as Run-time modification. High-level language is nothing but the language which is having higher level of abstraction.

[**Just In Time(JIT) compiler**](https://www.geeksforgeeks.org/just-in-time-compiler/)

While Java was developed as an interpreted language, in order to improve performance, there is nothing about Java that prevents [bytecode](https://www.geeksforgeeks.org/generate-byte-code-file-python/) compilation into native code on the fly***. For that reason, not long after Java’s initial release, the HotSpot JVM was released. A just-in-time (JIT) bytecode compiler is included in HotSpot.*** A Just In Time(JIT) compiler is part of the JVM **and on a piece-by-piece demand basis**, **selected portions of bytecode** are compiled into executable code in real-time. That is, **as is necessary during execution, a JIT compiler compiles code.** In addition**, not all bytecode sequences are compiled, only those that will benefit from the compilation.** The just-in-time method, however, still yields a major boost in inefficiency. The portability and safety function still exists even though **dynamic compilation is applied to bytecode** since the JVM is still in control of the execution environment.

Interpreting the bytecode, the standard implementation of the JVM slows the execution of the programs. JIT compilers interact with JVM at runtime to improve performance and compile appropriate bytecode sequences into native machine code.

**Hardware (CPU and memory is involved) is interpreting the code instead of JVM (**Java Virtual Machine). This can lead to performance gains in the speed of execution. **This can be done per-file, per-function, or maybe on any arbitrary code fragment; *the code is often compiled when it’s close to being executed (hence the name “just-in-time”), and then cached and reused later without having to be recompiled.*** It performs many optimizations: data analysis, translation from stack operations to registry operations, reduction of memory access by registry allocation, elimination of common sub-expressions.

**Working of 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.

**Difference between Interpreter and JIT compliler::**

The main difference between Interpreter and JIT compiler is that the interpreter is a **software** that **converts** the source code into native [machine code](https://pediaa.com/what-is-the-difference-between-machine-code-and-bytecode/#Machine%20Code) **line by line** while JIT compiler is a component in [JVM](https://pediaa.com/what-is-the-difference-between-jvm-and-clr/#JVM) **that improves the performance of Java programs** by **compiling**[bytecodes](https://pediaa.com/what-is-the-difference-between-machine-code-and-bytecode/#Bytecode) into native machine codes **at runtime.**

Interpreter is a program that translates the programmer written instructions or scripts into corresponding machine code that matches a particular hardware platform of a CPU. On the other hand, JIT is a compiler that translates bytecodes into machine codes at runtime. It requires CPU time and memory.

**Differences between Java Compiler,JIT Compiler,Java Interpreter**  
**Java Compiler**->Java compiler just convert .java programs into .class files,  
                   that means converting our source code into byte code.  
  
**JIT Compiler**->JUST IN TIME COMPILER is the part of the Java Virtual Machine (JVM)  
                   that is used to speed up the execution time. JIT compiles parts of the byte code  
                   that have similar functionality at the same time, and hence reduces the amount of  
                   time needed for compilation. **Here the term “compiler” refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.**  
  
**Java Interpreter**->JAVA INTERPRETER converts total Byte code into machine dependent code, if JVM use only interpreter the performance of JVM will slow, so JVM uses JIT compiler and Interpreter both.

**Difference between class path and path in java?**

Differences are like...  
  
1) Path is an *environment variable* which is **used by the operating system** **to find the executables.**  
Classpath is an environment variable which is **used by the Java compiler to find the path, of classes**.ie in J2EE we give the **path of jar files.**  
  
2)PATH is nothing but setting up an environment for operating system. Operating System will look in this PATH for executables.  
Classpath is nothing but setting up the environment for Java. Java will use to find compiled classes

it is required to Set the PATH environment variable to conveniently run the executable (javac.exe, java.exe, javadoc.exe, and so on) from any directory without having to type the full path of the command, such as:

C:\javac TestClass.java

Otherwise, you need to specify the full path every time you run it, such as:

C:\Java\jdk1.7.0\bin\javac TestClass.java

The following table demonstrates the difference between a PATH and a CLASSPATH

| **S. No.** | **PATH** | **CLASSPATH** |
| --- | --- | --- |
| 1. | An environment variable is used by the operating system to find the executable files. | An environment variable is used by the Java compiler to find the path of classes. |
| 2. | PATH setting up an environment for the operating system. Operating System will look in this PATH for executables. | Classpath setting up the environment for Java. Java will use to find compiled classes. |
| 3. | Refers to the operating system. | Refers to the Developing Environment. |
| 4. | In path variable, we must place .\bin folder path | In classpath, we must place .\lib\jar file or directory path in which .java file is available. |
| 5. | PATH is used by CMD prompt to find binary files. | CLASSPATH is used by the compiler and JVM to find library files. |

**.pointer and reference**

[C/C++ Pointers vs Java References - GeeksforGeeks](https://www.geeksforgeeks.org/is-there-any-concept-of-pointers-in-java/)

**List down the important features of Java 1.8.**

Java 8 or JDK 1.8 is the most significant expansion of the Java language yet. Java 8’s new features such as Lambda Expressions, Stream APIs, Nashorn, Compact Profiles, new Time APIs increase the expressive power of the platform and make it easier for developers to take advantage of modern, multicore processors. This article gives an overview of the new features in Java 8 with links to in-depth tutorials for the most important of them.

**New Features in Java 8**

1. [**Lambda Expressions**](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as [**Functional Interfaces**](https://www.javabrahman.com/java-8/functional-interfaces-java-8/)) more compactly.
2. [**Method references**](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/) provide easy-to-read lambda expressions for methods that already have a name. [**Constructor References**](https://www.javabrahman.com/java-8/constructor-references-java-8-simplified-tutorial/) are the equivalent forms of representation for constructors.
3. [**Default methods**](https://www.javabrahman.com/java-8/default-methods-in-java-8-with-examples/) enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
4. With default methods in Java 8, [**multiple inheritance of behavior**](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-of-behavior-from-interfaces-using-default-methods/) is now possible in Java, and it is important to understand the [**conflict resolution rules**](https://www.javabrahman.com/java-8/java-8-multiple-inheritance-conflict-resolution-rules-and-diamond-problem/) which resolve *Diamond Problem* and other conflict scenarios.
5. Java 8’s **new package** **[java.util.function](https://www.javabrahman.com/java-8/java-8-java-util-function-package-tutorial/)** provides many useful functional interfaces for the most common scenarios. The 4 most important functional interface among them are – [**Predicate**](https://www.javabrahman.com/java-8/java-8-java-util-function-predicate-tutorial-with-examples/), [**Consumer**](https://www.javabrahman.com/java-8/java-8-java-util-function-consumer-tutorial-with-examples/), [**Function**](https://www.javabrahman.com/java-8/java-8-java-util-function-function-tutorial-with-examples/) and [**Supplier**](https://www.javabrahman.com/java-8/java-8-java-util-function-supplier-tutorial-with-examples/).
6. [**Repeating Annotations**](https://www.javabrahman.com/java-8/java-8-repeating-annotations-tutorial/) provide the ability to apply the same annotation type more than once to the same declaration or type use.
7. New **java.util.stream** package provides a new [**Streams API**](https://www.javabrahman.com/java-8/java-8-streams-api-tutorial-with-examples/) to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API.
8. Java 8’s [**new Collector interface**](https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/) and its multiple predefined implementations provide an efficient way to terminate the Stream operations and collect the result in a collection.
9. **A**[**new Date-Time package**](https://www.javabrahman.com/java-8/overview-of-java-8-new-date-time-api-java-time-package-tutorial/)**– java.time** – with a new comprehensive set of date and time utilities.
10. Java 8’s **new enhanced methods in Collections API** are covered in a series of 4 tutorials – **[Iterable.forEach() & Iterator.remove()](https://www.javabrahman.com/java-8/java-8-iterable-foreach-iterator-remove-methods-tutorial-with-examples/)**,  **[Collection.removeIf()](https://www.javabrahman.com/java-8/java-8-collection-removeif-method-tutorial-with-examples/)**,  **[List.sort() & List.replaceAll()](https://www.javabrahman.com/java-8/new-features-in-java-8/www.javabrahman.com/java-8/java-8-list-sort-list-replaceall-methods-tutorial-with-examples/)**,  and new [**multi-value map**](https://www.javabrahman.com/java-8/java-8-maps-computeifabsent-computeifpresent-getordefault-methods-tutorial-with-examples/) methods.
11. Java 8 has introduced [**new internal iterators**](https://www.javabrahman.com/java-8/java-8-internal-iterators-vs-external-iterators/) based on declarative functional programming style.
12. Comparator interface has undergone a major upgrade in Java 8 with new methods leveraging Java 8’s functional programming features, comparator chaining, in-built null handling, and many more such features. [**Java 8 Comparator tutorial**](https://www.javabrahman.com/java-8/the-complete-java-8-comparator-tutorial-with-examples/) covers these new features in depth.
13. **Nashorn Javascript Engine enhanced** to provide a version of javascript which would run within the JVM
14. **Standard Encoding and Decoding Base64**,**Parallel Array Sorting** and **Unsigned Arithematic Support**.
15. **JDBC 4.2** with new features and notably the JDBC-ODBC bridge has been removed.
16. **Concurrency related important changes** which include –
    * Changes to ConcurrentHashMap to support aggregate operations based on the newly added streams facility and lambda expression.
    * Addition of classes to the java.util.concurrent.atomic package to support scalable updatable variables.
    * Support for a common pool in ForkJoinPool.
    * New **StampedLock** class to to provide a capability-based lock with three modes for controlling read/write access
17. **Type Annotations** provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
18. **Improved Type Inference** and **Method Parameter Reflection**.
19. **Compact Profiles** contain predefined subsets of the Java SE platform and enable applications that do not require the entire Platform to be deployed and run on small devices.
20. **Improved internationalization** including support for Unicode 6.2.0, new Calendar and Locale APIs, Adoption of Unicode CLDR Data and the java.locale.providers System Property and Ability to Install a Custom Resource Bundle as an Extension.

---------------------------------------------------------

**Q2.** **Differentiate between C++ & Java.**

|  |  |  |
| --- | --- | --- |
| **Comparison Index** | **C++** | **Java** |
| **Platform-independent** | C++ is platform-dependent. | Java is ***platform-independent.*** |
| **Mainly used for** | C++ is mainly used for ***system programming.*** | Java is mainly used ***for application programming.*** It is widely used in Windows-based, web-based, enterprise, and mobile applications. |
| **Goto** | C++ supports the [goto](https://www.javatpoint.com/cpp-goto-statement) statement. | Java ***doesn't support the goto statement.*** |
| **Multiple inheritance** | C++ supports multiple inheritance. | Java ***doesn't support multiple inheritance through class.*** *It can be achieved by using*[*interfaces in java*](https://www.javatpoint.com/interface-in-java)*.* |
| **Operator Overloading** | C++ supports [operator overloading](https://www.javatpoint.com/cpp-overloading). | Java ***doesn't support operator overloading.*** |
| **Pointers** | C++ supports [pointers](https://www.javatpoint.com/cpp-pointers). You can write a pointer program in C++. | ***Java supports pointer internally***. *However,* ***you can't write*** *the pointer program in java*. It means java has restricted pointer support in java. |
| **Compiler and Interpreter** | C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent. | ***Java uses both compiler and interpreter.*** Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent. |
| **Call by Value and Call by reference** | C++ supports both call by value and call by reference. | ***Java supports call by value only***. There is no call by reference in java. |
| **Structure and Union** | C++ supports structures and unions. | Java ***doesn't support structures and unions.*** |
| **Thread Support** | C++ doesn't have built-in support for threads. *It relies on third-party libraries for thread support.* | Java ***has built-in***[***thread***](https://www.javatpoint.com/multithreading-in-java)***support.*** |
| **Documentation comment** | C++ doesn't support documentation comments. | Java ***supports documentation comment (/\*\* ... \*/)*** to create documentation for java source code. |
| **Virtual Keyword** | C++ supports virtual keyword so that we can decide whether or not to override a function. | ***Java has no virtual keyword***. We can override all non-static methods by default. In other words, non-static methods are virtual by default. |
| **unsigned right shift >>>** | C++ doesn't support >>> operator. | ***Java supports unsigned right shift >>> operator*** that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator. |
| **Inheritance Tree** | C++ always creates a new inheritance tree. | Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the [inheritance](https://www.javatpoint.com/inheritance-in-java) tree in java. |
| **Hardware** | C++ is nearer to hardware. | ***Java is not so interactive with hardware.*** |
| **Object-oriented** | C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible. | Java is also an [object-oriented](https://www.javatpoint.com/java-oops-concepts) language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object. |

**Note**

* Java ***doesn't support default arguments*** like C++.
* Java ***does not support header files like C++.*** J***ava uses the import keyword to include different classes and methods.***

C vs C++ vs Java

The languages are based on each other but still, they are different in design and philosophy. The following table describes the major differences between C, C++, and Java. It will help you to select which language you have to learn.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.N.** | **Basis** | **C** | **C++** | **Java** |
| **1** | **Origin** | The C language is based on BCPL. | The C++ language is based on the C language. | The Java programming language is based on both C and C++. |
| **2** | **Programming Pattern** | It is a procedural language. | It is an object-oriented programming language. | It is a pure object-oriented programming language. |
| **3** | **Approach** | It uses the top-down approach. | It uses the bottom-up approach. | It also uses the bottom-up approach. |
| **4** | **Dynamic or Static** | It is a static programming language. | It is also a static programming language. | It is a dynamic programming language. |
| **5** | **Code Execution** | The code is executed directly. | The code is executed directly. | The code is executed by the JVM. |
| **6** | **Platform Dependency** | It is platform dependent. | It is platform dependent. | It is platform-independent because of byte code. |
| **7** | **Translator** | It uses a compiler only to translate the code into machine language. | It also uses a compiler only to translate the code into machine language. | Java uses both compiler and interpreter and it is also known as an interpreted language. |
| **8** | **File Generation** | It generates the .exe, and .bak, files. | It generates .exe file. | It generates .class file. |
| **9** | **Number of Keyword** | There are **32** keywords in the C language. | There are **60** keywords in the C++ language. | There are **52** keywords in the Java language. |
| **10** | **Source File Extension** | The source file has a .c extension. | The source file has a .cpp extension. | The source file has a .java extension. |
| **11** | **Pointer Concept** | It supports pointer. | It also supports pointer. | Java does not support the pointer concept because of security. |
| **12** | **Union and Structure Datatype** | It supports union and structure data types. | It also supports union and structure data types. | It does not support union and structure data types. |
| **13** | **Pre-processor Directives** | It uses pre-processor directives such as #include, #define, etc. | It uses pre-processor directives such as #include, #define, #header, etc. | It does not use directives but uses packages. |
| **14** | **Constructor/ Destructor** | It does not support constructor and destructor. | It supports both constructor and destructor. | It supports constructors only. |
| **15** | **Exception Handling** | It does not support exception handling. | It supports exception handling. | It also supports exception handling. |
| **16** | **Memory Management** | It uses the calloc(), malloc(), free(), and realloc() methods to manage the memory. | It uses new and delete operator to manage the memory. | It uses a garbage collector to manage the memory. |
| **17** | **Overloading** | It does not support the overloading concept. | Method and operator overloading can be achieved. | Only method overloading can be achieved. |
| **18** | **goto Statement** | It supports the goto statement. | It also supports the goto statement. | It does not support the goto statements. |
| **19** | **Used for** | It is widely used to develop drivers and operating systems. | It is widely used for system programming. | It is used to develop web applications, mobile applications, and windows applications. |
| **20** | **Array Size** | An array should be declared with size. For example, int num[10]. | An array should be declared with size. | An array can be declared without declaring the size. For example, int num[]. |

**Q. Compiler & Interpreter**

A [**compiler**](http://en.wikipedia.org/wiki/Compiler)takes entire program and converts it into object code which is typically stored in a file. The object code is also referred as binary code and can be directly executed by the machine after linking. Examples of compiled programming languages are [C](https://www.geeksforgeeks.org/c/) and [C++](https://www.geeksforgeeks.org/c-plus-plus/).

An [**Interpreter**](http://en.wikipedia.org/wiki/Interpreter_%28computing%29)directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. Examples of interpreted languages are Perl, Python and Matlab.

Following are some interesting facts about interpreters and compilers.

1) Both compilers and interpreters convert source code (text files) into tokens, both may generate a parse tree, and both may generate immediate instructions. The basic difference is that a compiler system, including a (built in or separate) linker, generates a stand alone machine code program, while an interpreter system instead performs the actions described by the high level program.

2) Once a program is compiled, its source code is not useful for running the code. For interpreted programs, the source code is needed to run the program every time.

3) In general, interpreted programs run slower than the compiled programs.

4) [Java](https://www.geeksforgeeks.org/java/)programs are first compiled to an intermediate form, then interpreted by the interpreter.

**Q. Java Versions**

Resource 1: TELUSKO

Link: https://www.youtube.com/watch?v=P46cXxgwfRM

-------------------------------------------------------

In this video features of different Java versions from 1995 to 2017 are described.

Every version has atleast one new update.

Java 1.4 does not have generics.

Java 1.6 - Lambda expression cannot be implemented if it being used with server.

Java 1.7 and 1.8 have major updates.

First version of Java was named as OAK and was released in 1994.

Public release took place in 1995 as name "Java".

In every version there were some classes which were produced or reproduced and existing classes were deprecated.

Current version in Java is version 8 (2017).

Features:

----------

Java 1.1(1997) updates - inner class, JDBC and RMI (remote method invocation)

Java 1.2(J2SE) - 1998 - Swing classes and Collection API.

Java 1.3(2000) - HotSpot JVM and JNDI (Java naming and directory interface)

Java 1.4 (2002) - Regular expression, assert keyword.

Java 1.5 or Java 5.0 - Generics , Auto boxing and Unboxing ,Varagrs, enhanced for loop and Static imports.

Java SE 6 - JDBC updates

Java 1.7 (2011) - Strings in switch statement, Automatic resource management.

Java 1.8 (2014) - Method can be defined inside a interface using default keyword, internal iteration can be used for colection Api and Lamba expression.

------------------------------------------------------

RESOURCE 2 : CodeDecode

Link: https://www.youtube.com/watch?v=oUdENE7ljjw

------------------------------------------------------