1. **Few Facts of Java-**

* Java is Object Oriented. However, it is not considered as pure object oriented as it provides support for primitive data types (like int, char, etc)
* The Java codes are first compiled into byte code (machine independent code). Then the byte code is run on Java Virtual Machine (JVM) regardless of the underlying architecture.
* Java syntax is similar to C/C++. But Java does not provide low level programming functionalities like pointers. Also, Java codes are always written in the form of classes and objects.

1. **JDK, JRE and JVM?**

Ans- JVM, JRE and JDK all three are platform dependent because configuration of each Operating System is different. But Java is platform independent.

* **JDK (Java Development Kit)** : The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.

It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

* **JRE (Java Runtime Environment) :** JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK.

It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.

JRE is part of the Java Development Kit (JDK) but can be downloaded separately. JRE was originally developed by Sun Microsystems Inc., a wholly-owned subsidiary of Oracle Corporation

* **JVM**: JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed line by line. JVMs are available for many hardware and software platforms.

1. **Important Features of Java**

Ans-

* **Simple**
* **Platform Independent**
* **Architectural Neutral:** A Language or Technology is said to be Architectural neutral which can run on any available processors in the real world without considering their development and compilation.
* **Portable:** If any language supports platform independent and architectural neutral feature known as portable.
* **Multi-Threading:** A flow of control is known as a thread. When any Language executes multiple thread at a time that language is known as multithreaded e. It is multithreaded.
* **Distributed:** Using this language we can create distributed applications. In distributed application multiple client system depends on multiple server systems so that even problem occurred in one server will never be reflected on any client system.
* **Networked:** It is mainly designed for web-based applications, J2EE is used for developing network-based applications.
* **Robust:** Simply means of Robust are strong. It is robust or strong Programming Language because of its capability to handle Run-time Error, automatic garbage collection
* **Dynamic:** It supports Dynamic memory allocation due to this memory wastage is reduce and improve performance of the application.
* **Secured:** It is a more secure language compared to other language; In this language, all code is covered in byte code after compilation which is not readable by human.
* **Object Oriented:** It supports OOP's concepts because of this it is most secure language

1. **Naming conventions in Java?**

Ans- They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constants.

Classes and Interfaces- First letter of each word should be capitalized.

Methods- first letter should be in lowercase and first letter of each word capitalized.

Variables- Should not start with “$” and “\_”. One-character variable names should be avoided for temporary variables.

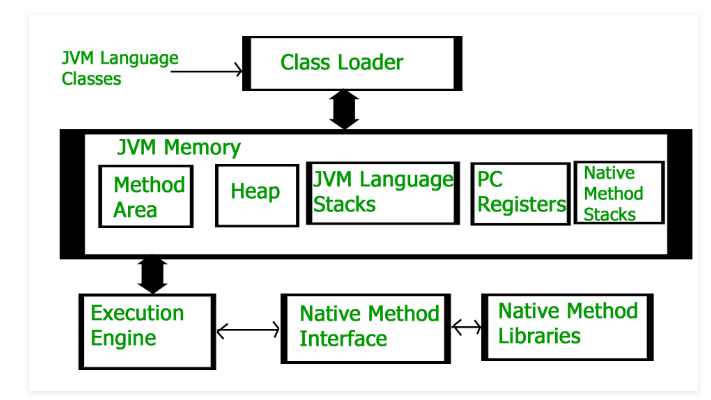
Constant- Should be all uppercase and separated by “\_”

Package- should be in lowercase.

1. **How JVM works?**

Ans- 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. 

**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 method area. For each *.class* file, JVM stores following information in method area.

* Fully qualified name of the loaded class and its immediate parent class.
* Whether *.class* file is related to Class or Interface or Enum
* Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory.

This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc

***Student s1 = new Student();***

***// Getting hold of Class object created by JVM.***

***Class c1 = s1.getClass();***

***// Printing type of object using c1.***

***System.out.println(c1.getName());***

***// getting all methods in an array***

***Method m[] = c1.getDeclaredMethods();***

***for (Method method : m)***

***System.out.println(method.getName());***

***// getting all fields in an array***

***Field f[] = c1.getDeclaredFields();***

***for (Field field : f)***

***System.out.println(field.getName());***

**Note:** For every loaded *.class* file, only **one** object of Class is created.

Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true

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

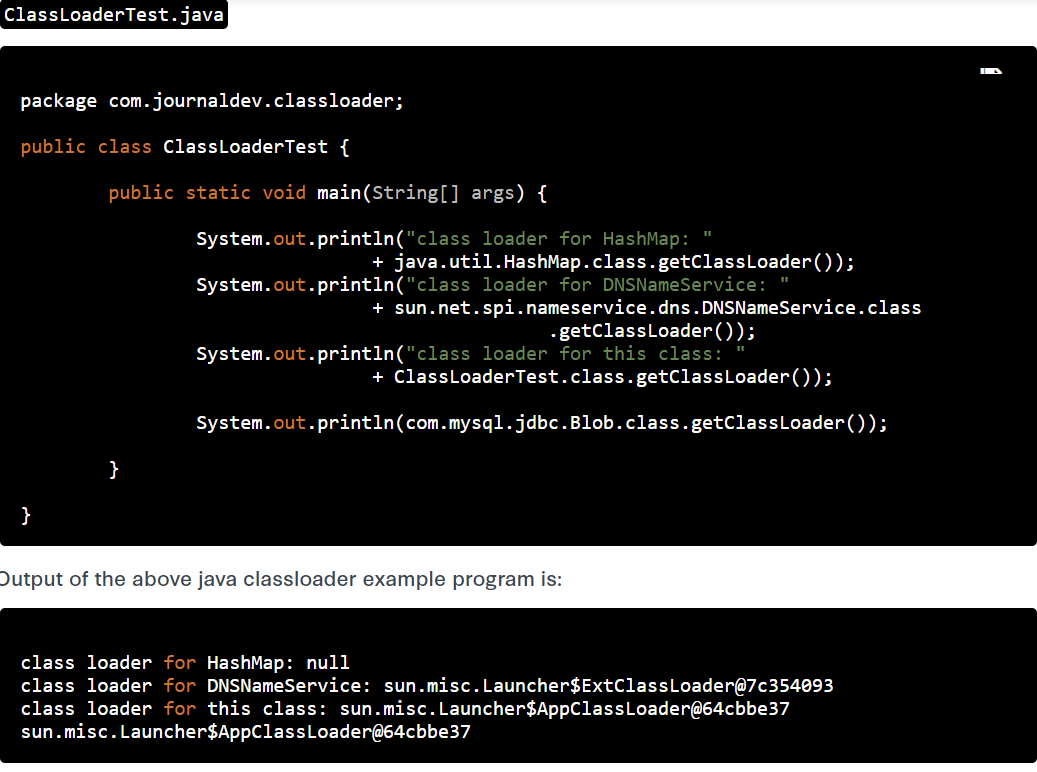
* *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
* *Preparation* : JVM allocates memory for class variables and initializing 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 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 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 *JAVA\_HOME/jre/lib* directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
* ***Extension class loader*** : It is child of 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$ExtClassLoader* class.
* ***System/Application class loader***: It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader* class.

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

Java ClassLoader is hierarchical and whenever a request is raised to load a class, it delegates it to its parent and in this way uniqueness is maintained in the runtime environment. If the parent class loader doesn’t find the class then the class loader itself tries to load the class.



As you can see that [java.util.HashMap](https://www.journaldev.com/11560/java-hashmap" \t "_blank) ClassLoader is coming as null that reflects Bootstrap ClassLoader whereas DNSNameService ClassLoader is ExtClassLoader. Since the class itself is in CLASSPATH, System ClassLoader loads it.

When we are trying to load [HashMap](https://www.journaldev.com/11560/java-hashmap), our System ClassLoader delegates it to the Extension ClassLoader, which in turns delegates it to Bootstrap ClassLoader that found the class and load it in JVM.

The same process is followed for DNSNameService class but Bootstrap ClassLoader is not able to locate it since it’s in $JAVA\_HOME/lib/ext/dnsns.jar and hence gets loaded by Extensions Classloader

**JVM Memory**  
**Method area: In** 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.

**Heap area: Information** of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

**Stack area: For** every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.

**PC Registers: Store** address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

**Native method stacks :**For every thread, separate native stack is created. It stores native method information.

**Execution Engine**  
Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in 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 efficiency of interpreter. It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls, JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.
* ***Garbage Collector***: It destroy un-referenced objects.

**Java Native Interface (JNI):**  
It is an interface which 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.

1. **JVM Shutdown Hook in Java?**

Ans- Shutdown Hooks are a special construct that allows developers to plug in a piece of code to be executed when the JVM is shutting down. This comes in handy in cases where we need to do special clean up operations in case the VM is shutting down.

***public class ShutDownHook***

***{***

***public static void main(String[] args)***

***{***

***Runtime.getRuntime().addShutdownHook(new Thread()  {***

***public void run()       {***

***System.out.println("Shutdown Hook is running !");***

***}***

***});***

***System.out.println("Application Terminating ...");***

***}***

***}***

Output:

Application Terminating ...

Shutdown Hook is running !

**Note: *Shutdown hooks are called when the application terminates normally*** *(when all threads finish, or when System.exit(0) is called). Also, when the JVM is shutting down due to external causes such as a user requesting a termination (Ctrl+C), a SIGTERM being issued by O/S (normal kill command, without -9), or when the operating system is shutting down.*

***It is not guaranteed that shutdown hooks will always run.*** *If the JVM crashes due to some internal error, then it might crash down without having a chance to execute a single instruction. Also, if the O/S gives a SIGKILL (http://en.wikipedia.org/wiki/SIGKILL) signal (kill -9 in Unix/Linux) or TerminateProcess (Windows), then the application is required to terminate immediately without doing even waiting for any cleanup activities.*

***We can have more than one Shutdown Hooks,*** *The JVM can execute shutdown hooks in any arbitrary order. Moreover, the JVM might execute all these hooks concurrently.*

**Once shutdown sequence starts, it can be stopped by Runtime.halt() only.**

1. **If a .java file has more than one class then each class will compile into a separate class files.**
2. **Does JVM create an object of class Main?**

Ans- The answer is “No”. We have studied that the reason for main() static in Java is to make sure that the main() can be called without any instance. To justify the same, we can see that the following program compiles and runs fine.

|  |
| --- |
| Not Main is abstract  abstract class Main {      public static void main(String args[])  {          System.out.println("Hello");      } } |

Output:

Hello

Since we can’t create object of [abstract classes in Java](https://www.geeksforgeeks.org/abstract-classes-in-java/), it is guaranteed that object of class with main() is not created by JVM.

1. **What makes it JAVA as platform independent language?**

Ans-  A compiler is a program that translates the source code for another program from a programming language into executable code.  
This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the **Java Byte Code.**

**Step by step Execution of Java Program:**

* Whenever, a program is written in JAVA, the javac compiles it.
* The result of the JAVA compiler is the **.class file or the bytecode** and not the machine native code (unlike C compiler).
* The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
* And finally, program runs to give the desired output.
* So here bytecode make JAVA language as platform independent and also portable.

In case of C or C++ (language that are not platform independent), the compiler generates an .exe file which is OS dependent. When we try to run this .exe file on another OS it does not run, since it is OS dependent and hence is not compatible with the other OS.

**Java is platform independent but JVM is platform dependent.**

1. **JDBC?**

Ans-It is an application programming interface (API) for the programming language Java, which defines how a client may access any kind of tabular data, especially relational database.

t acts as a middle layer interface between java applications and database.

The JDBC classes are contained in the Java Package **java.sql** and **javax.sql**.  
JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver: This driver interacts directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

* Does not require any native library and Middleware server, so no client-side or server-side installation.
* It is fully written in Java language, hence they are portable drivers.

**Which Driver to use When?**

* If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
* If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
* Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
* The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

1. **Is main method compulsory in Java?**

Ans- Prior to JDK 7, the main method was not mandatory in a java program. You could write your full code under [static block](https://www.geeksforgeeks.org/g-fact-79/) and it ran normally.

* The static block is first executed as soon as the class is loaded before the main();
* It will run static block first and then it will see no main() is there. Therefore, it will give **“exception”**, as exception comes while execution. However, if we don’t want an exception, we can terminate the program by System.exit(0);
* From JDK7 main method is mandatory. The compiler will verify first, whether main() is present or not. If your program doesn’t contain the main method, then you will get an **error** “main method not found in the class”. It will give an error (byte code verification error because in it’s byte code, main is not there) not an exception because the program has not run yet.

1. **In java file name and class name should be the same if the class is declared as public.**
2. **In Java, Using predefined class name as Class or Variable name is allowed but you cannot use a keyword as name of a class, name of a variable nor the name of a folder used for package**.

**// Number is predefined class name in java.lang package**

// Note : java.lang package is included in every java program by default

public class Number

{

    public static void main (String[] args)     {

        System.out.println("It works"); }}

***Output: It works***

**Using String as User Defined Class:**

|  |
| --- |
| // String is predefined class name in java.lang package  // Note : java.lang package is included in every java program by default  public class String  {      public static void main (String[] args)   {          System.out.println("I got confused");  } } |

However, in this case you will get run-time error like this:  [Main thread](https://www.geeksforgeeks.org/main-thread-java/)is looking for main method() with predefined **String class** array argument. But here, it got main method() with user defined String class. Whenever Main thread will see a class name, it tries to search that class scope by scope. First it will see in your program, then in your package.If not found, then [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) follows delegation hierarchy principle to load that class.Hence you will get run-time error. To fix, write this -**public static void main (java.lang.String[] args)**

1. Identifiers are used for identification purpose. In Java, an identifier can be a class name, method name, variable name or a label. It should be alphanumeric character or “$” or “\_” and should not start with the digit, are case-sensitive and reserved words can’t be used as an identifier.
2. values of type boolean are not converted implicitly or explicitly (with casts) to any other type



**By default, fraction value is double in java and use byte and short if memory is a constraint.**

1. **Enum in java?**

Ans-Enumerations serve the purpose of representing a group of named constants like the planets, days of the week, colors, directions, etc.

* Enums are used when we know all possible values at **compile time**.
* It is not necessary that the set of constants in an enum type stay **fixed** for all time.
* In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types).
* Enum declaration can be done outside a Class or inside a Class but not inside a Method.

enum Color

{

    RED, GREEN, BLUE;

}

public class Test {

    public static void main(String[] args) {

        Color c1 = Color.RED;

        System.out.println(c1);

} }

**Output:RED**

* First line inside enum should be list of constants and then other things like methods, variables and constructor.
* Every enum internally implemented by using Class.

/\* internally above enum Color is converted to

class Color

{

public static final Color RED = new Color();

public static final Color BLUE = new Color();

public static final Color GREEN = new Color();

}\*/

* Every enum constant represents an **object** of type enum.
* Enum type can be passed as an argument to **switch** statement.
* Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using enum Name. Since it is **final**, we can’t create child enums.
* We can declare **main() method** inside enum. Hence we can invoke enum directly from the Command Prompt.
* All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
* **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name but enum can implement many interfaces.
* **Enum methods**- These methods are present inside **java.lang.Enum**.
* **values() method** can be used to return all values present inside enum.
* Order is important in enums.By using **ordinal() method**, each enum constant index can be found, just like array index.
* **valueOf() method** returns the enum constant of the specified string value, if exists

enum Color

{

    RED, GREEN, BLUE;

}

public class Test

{

    public static void main(String[] args)

    {

        // Calling values()

        Color arr[] = Color.values();

        // enum with loop

        for (Color col : arr)

        {

            // Calling ordinal() to find index

            // of color.

            System.out.println(col + " at index "

                             + col.ordinal());

        }

        // Using valueOf(). Returns an object of Color with given constant.

        // Uncommenting second line causes exception IllegalArgumentException

        System.out.println(Color.valueOf("RED"));

        // System.out.println(Color.valueOf("WHITE"));

    }

}

Output: RED at index 0

GREEN at index 1

BLUE at index 2

RED

**enum and constructor :**

* enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
* We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

**enum and methods :**

* enum can contain **concrete** methods only i.e. no any **abstract** method.

|  |
| --- |
| // Java program to demonstrate that enums can have constructor  // and concrete methods.    // An enum (Note enum keyword inplace of class keyword)  enum Color  {      RED, GREEN, BLUE;        // enum constructor called separately for each constant      private Color()      {          System.out.println("Constructor called for : " + this.toString());      }        // Only concrete (not abstract) methods allowed      public void colorInfo()      {          System.out.println("Universal Color");      }  }    public class Test  {       public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);          c1.colorInfo();      }  } |

Output:

Constructor called for : RED

Constructor called for : GREEN

Constructor called for : BLUE

RED

Universal Color