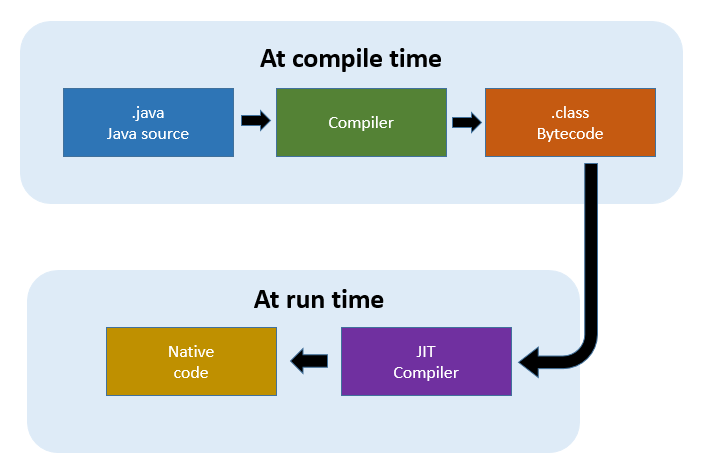
**How does it work ?!**

The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time. Java programs consists of classes, which contain platform neutral bytecode that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time.   


The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.   
JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

**Different compilers for different applications**

The JIT compiler comes in two flavors, and the choice of which compiler to use is often the only compiler tuning that needs to be made when running an application. In fact, knowing which compiler you want to choose is something that must be considered even before Java is installed, since different Java binaries contain different compilers.

**Client-side compilers**

A well-known optimizing compiler is C1, the compiler that is enabled through the -client JVM startup option. As its startup name suggests, C1 is a client-side compiler. It is designed for client-side applications that have fewer resources available and are, in many cases, sensitive to application startup time. C1 use performance counters for code profiling to enable simple, relatively unintrusive optimizations.

**Server-side compilers**

For long-running applications such as server-side enterprise Java applications, a client-side compiler might not be enough. A server-side compiler like C2 could be used instead. C2 is usually enabled by adding the JVM startup option -server to your startup command-line. Since most server-side programs are expected to run for a long time, enabling C2 means that you will be able to gather more profiling data than you would with a short-running light-weight client application. So you'll be able to apply more advanced optimization techniques and algorithms.

**Tiered compilation**

Tiered compilation combines client-side and server-side compilation. Tiered compilation takes advantage of both client and server compiler advantages in your JVM. The client compiler is most active during application startup and handles optimizations triggered by lower performance-counter thresholds. The client-side compiler also inserts performance counters and prepares instruction sets for more advanced optimizations, which will be addressed at a later stage by the server-side compiler. Tiered compilation is a very resource-efficient way of profiling because the compiler is able to collect data during low-impact compiler activity, which can be used for more advanced optimizations later. This approach also yields more information than you'll get from using interpreted code profile counters alone.

**Code optimization**

When a method is chosen for compilation, the JVM feeds its bytecode to the Just-In-Time compiler (JIT). The JIT needs to understand the semantics and syntax of the bytecode before it can compile the method correctly. To help the JIT compiler analyze the method, its bytecode are first reformulated in an internal representation called [trees](https://en.wikipedia.org/wiki/Trace_tree), which resembles machine code more closely than bytecode. Analysis and optimizations are then performed on the trees of the method. At the end, the trees are translated into native code. The JIT compiler can use more than one compilation thread to perform JIT compilation tasks. Using multiple threads can potentially help Java applications to start faster. In practice, multiple JIT compilation threads show performance improvements only where there are unused processing cores in the system. The default number of compilation threads is identified by the JVM, and is dependent on the system configuration. If the resulting   
number of threads is not optimum, you can override the JVM decision by using the XcompilationThreads option. For information on using this option, see JIT and AOT command-line options

The compilation consists of the following phases:

**Inlining**

Inlining is the process by which the trees of smaller methods are merged, or "inlined", into the trees of their callers. This speeds up frequently executed method calls.

**Local optimizations**

Local optimizations analyze and improve a small section of the code at a time. Many local optimizations implement tried and tested techniques used in classic static compilers.

**Control flow optimizations**

Control flow optimizations analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency.

**Global optimizations**

Global optimizations work on the entire method at once. They are more "expensive", requiring larger amounts of compilation time, but can provide a great increase in performance.

**Native code generation**

Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions; some small optimizations are performed according to architecture character.

# **Differences between JDK, JRE and JVM**

**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.

**JAVA RUNTIME ENVIRONMENT**

**JRE** stands for **“Java Runtime Environment”** and may also be written as **“Java RTE.”** The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

[**JAVA VIRTUAL MACHINE**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)

It is:

* 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 Sun and other companies.
* An **implementation** is a computer program that meets the requirements of the JVM specification
* **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

**Difference betweem JDK, JRE and JVM**

To understand the difference between these three, let us consider the following diagram.  


* **JDK** – **Java Development Kit** (in short JDK) is Kit which provides the environment to **develop and execute(run)**the Java program. JDK is a kit(or package) which includes two things
  + 1. Development Tools(to provide an environment to develop your java programs)
    2. JRE (to execute your java program).

**Note :**JDK is only used by Java Developers.

* **JRE** – **Java Runtime Environment** (to say JRE) is an installation package which provides environment to **only run(not develop)** the java program(or application)onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
* **JVM** – **Java Virtual machine**(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

**How does JRE and JDK works?**

**What does JRE consists of?**  
JRE consists of the following components:

* **Deployment technologies**, including deployment, Java Web Start and Java Plug-in.
* **User interface toolkits**, including Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag and drop (DnD) and input methods.
* **Integration libraries**, including Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) and scripting.
* **Other base libraries**, including international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization and Java for XML Processing (XML JAXP).
* **Lang and util base libraries**, including lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects and Regular Expressions.
* **Java Virtual Machine (JVM)**, including Java HotSpot Client and Server Virtual Machines.

**How does JRE works?**  
To understand how the JRE works let us consider a Java source file saved as Example.java. The file is compiled into a set of Byte Code that is stored in a “.class” file. Here it will be “Example.class“.  
  
  
The following diagram depicts what is done at compile time.  
  
The following actions occur at runtime.

* **Class Loader**

The Class Loader loads all necessary classes needed for the execution of a program. It provides security by separating the namespaces of the local file system from that imported through the network. These files are loaded either from a hard disk, a network or from other sources.

* **Byte Code Verifier**

The JVM puts the code through the Byte Code Verifier that checks the format and checks for an illegal code. Illegal code, for example, is code that violates access rights on objects or violates the implementation of pointers.

The Byte Code verifier ensures that the code adheres to the JVM specification and does not violate system integrity.  


* **Intrepreter**

At runtime the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:

* + Execute the Byte Code
  + Make appropriate calls to the underlying hardware

Both operations can be shown as:  
  
To understand the interactions between JDK and JRE consider the following diagram.  


**How does JVM works?**

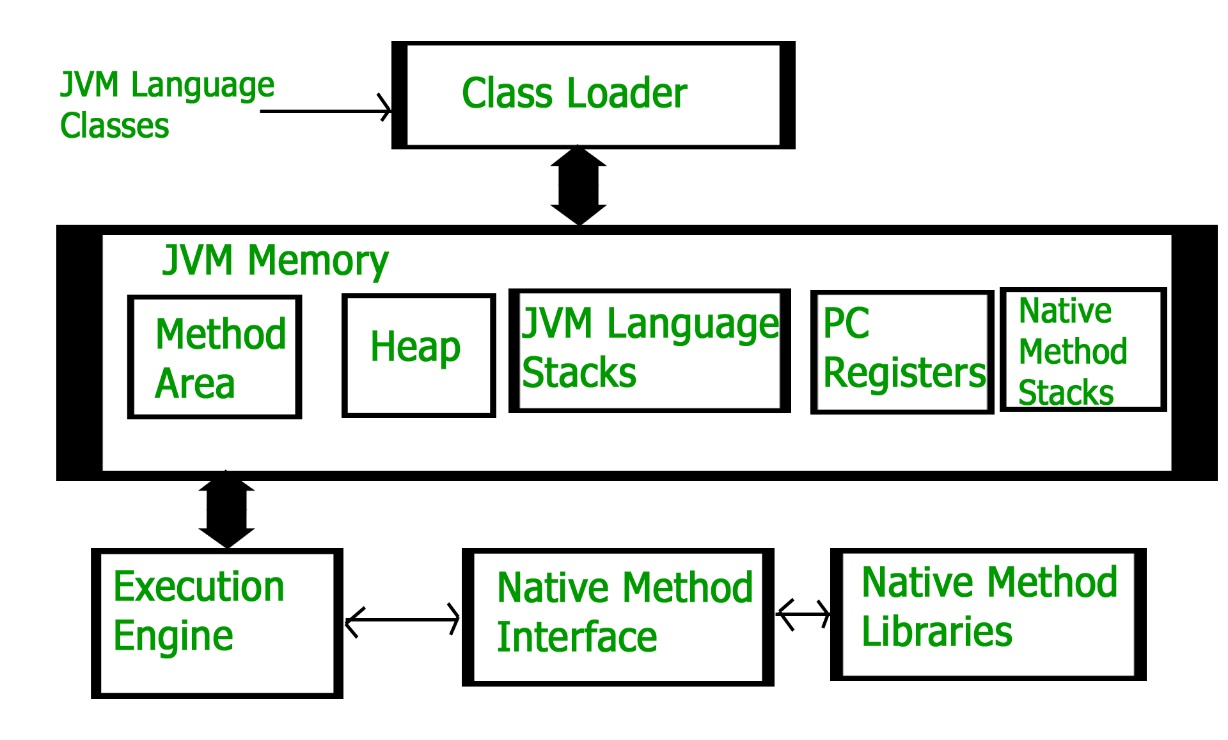
JVM becomes an instance of JRE at runtime of a Java program. It is widely known as a runtime interpreter.JVM largely helps in the abstraction of inner implementation from the programmers who make use of libraries for their programmes from JDK

# **How JVM Works – JVM Architecture?**

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.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/jvm-3.jpg)

**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. Please note that this object is of type Class predefined in java.lang package. 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. To get this object reference we can use getClass() method of [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // A Java program to demonstrate working of a Class type  // object created by JVM to represent .class file in  // memory.  import java.lang.reflect.Field;  import java.lang.reflect.Method;    // Java code to demonstrate use of Class object  // created by JVM  public class Test  {      public static void main(String[] args)      {          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());      }  }    // A sample class whose information is fetched above using  // its Class object.  class Student  {      private String name;      private int roll\_No;        public String getName()  {  return name;   }      public void setName(String name) { this.name = name; }      public int getRoll\_no()  { return roll\_No;  }      public void setRoll\_no(int roll\_no) {          this.roll\_No = roll\_no;      }  } |

Output:

Student

getName

setName

getRoll\_no

setRoll\_no

name

roll\_No

**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.

filter\_none

edit

play\_arrow

brightness\_4

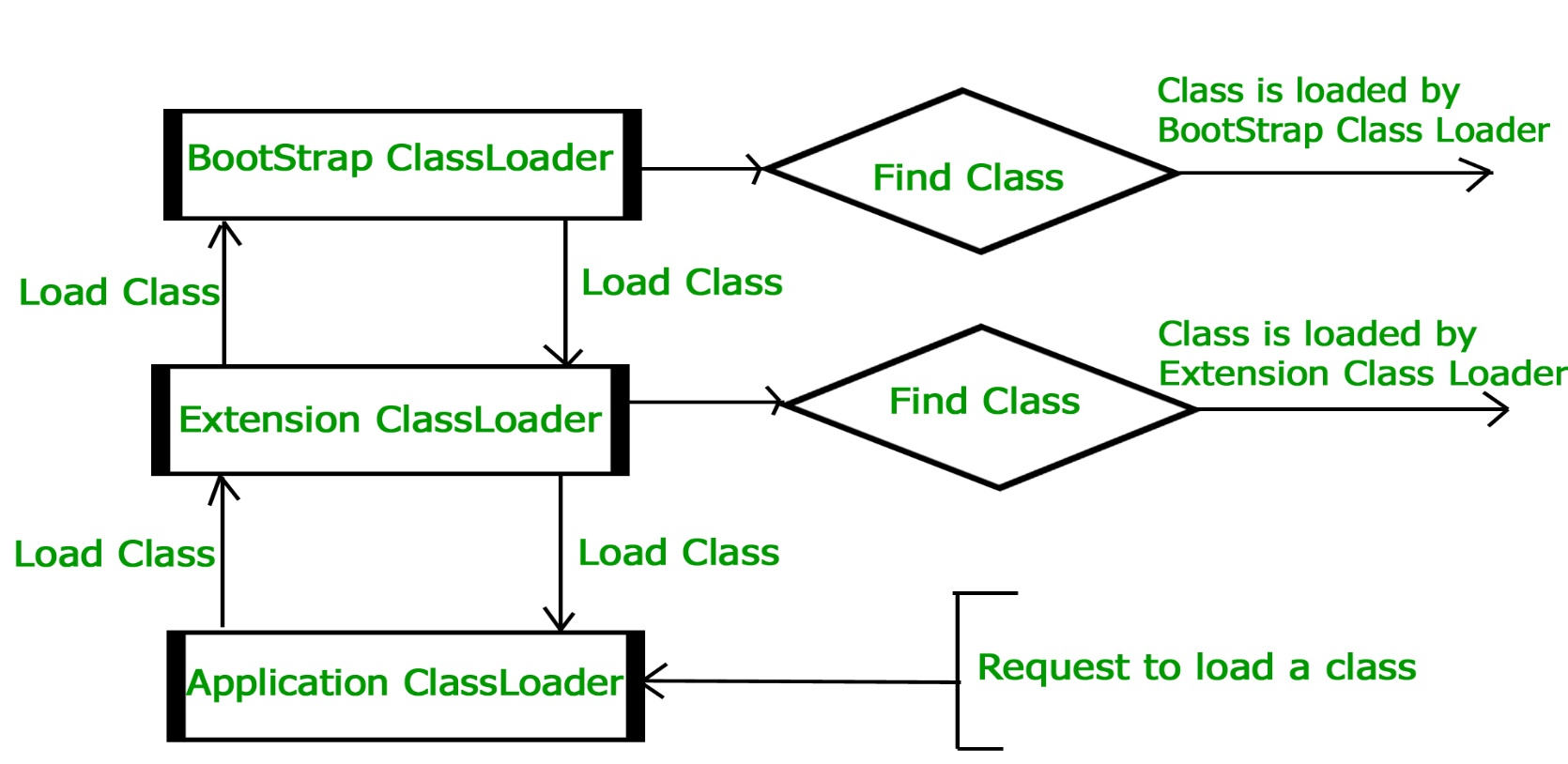
|  |
| --- |
| // Java code to demonstrate Class Loader subsystem  public class Test  {      public static void main(String[] args)      {          // String class is loaded by bootstrap loader, and          // bootstrap loader is not Java object, hence null          System.out.println(String.class.getClassLoader());            // Test class is loaded by Application loader          System.out.println(Test.class.getClassLoader());      }  } |

Output:

null

sun.misc.Launcher$AppClassLoader@73d16e93

**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.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/jvmclassloader.jpg)

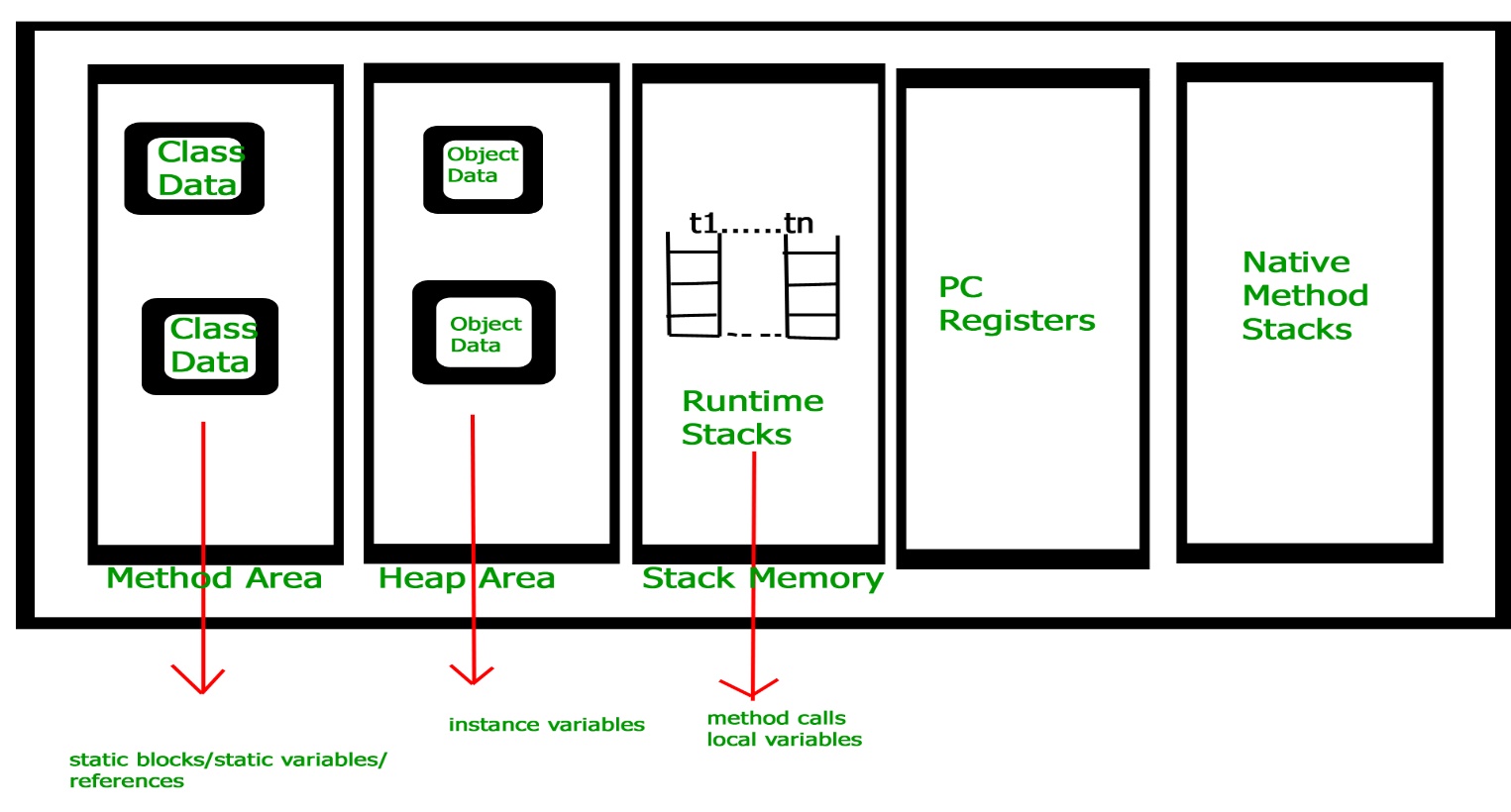
**JVM Memory**  
**Method area :**In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

**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.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/jvm-memory-2.jpg)

**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.For more on Garbage Collector,refer [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/).

**Java Native Interface (JNI) :**  
It is a 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.