**OOPs** (Object-Oriented Programming System)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts: **Inheritance, Polymorphism, Abstraction, Encapsulation**

Apart from these concepts, there are some other terms which are used in Object-Oriented design: **Coupling, Cohesion, Association, Aggregation, Composition**  
  
**Object**: Any entity that has state and behavior is known as an object. An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.

**Class**: Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.  
  
**Inheritance**: When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.  
  
**Polymorphism**: If one task is performed in different ways, it is known as polymorphism. In Java, we use method overloading and method overriding to achieve polymorphism.  
  
**Abstraction**: Hiding internal details and showing functionality is known as abstraction. In Java, we use abstract class and interface to achieve abstraction.  
  
**Encapsulation**: Binding (or wrapping) code and data together into a single unit are known as encapsulation. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.  
  
**Coupling**: Coupling refers to the knowledge or information or dependency of another class. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

**Cohesion**: Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

**Association**: Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects: ***One to One, One to Many, Many to One, Many to Many*.** Association can be unidirectional or bidirectional.

**Aggregation**: Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a **has-a** relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

**Composition**: The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

**What is Java Architecture?**

* In Java, there is a process of compilation and interpretation.
* The code written in [Java](https://www.edureka.co/blog/java-tutorial/), is converted into byte codes which is done by the Java Compiler.
* The byte codes, then are converted into machine code by the JVM.
* The Machine code is executed directly by the machine.

This diagram illustrates the internal working of a Java code, or precisely, Java Architecture!  


**Components of Java Architecture** There are three main components of Java language: JVM, JRE, and JDK.  
 **Differences between JDK, JRE and JVM.**

* **JDK** – **Java Development 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** 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** 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.

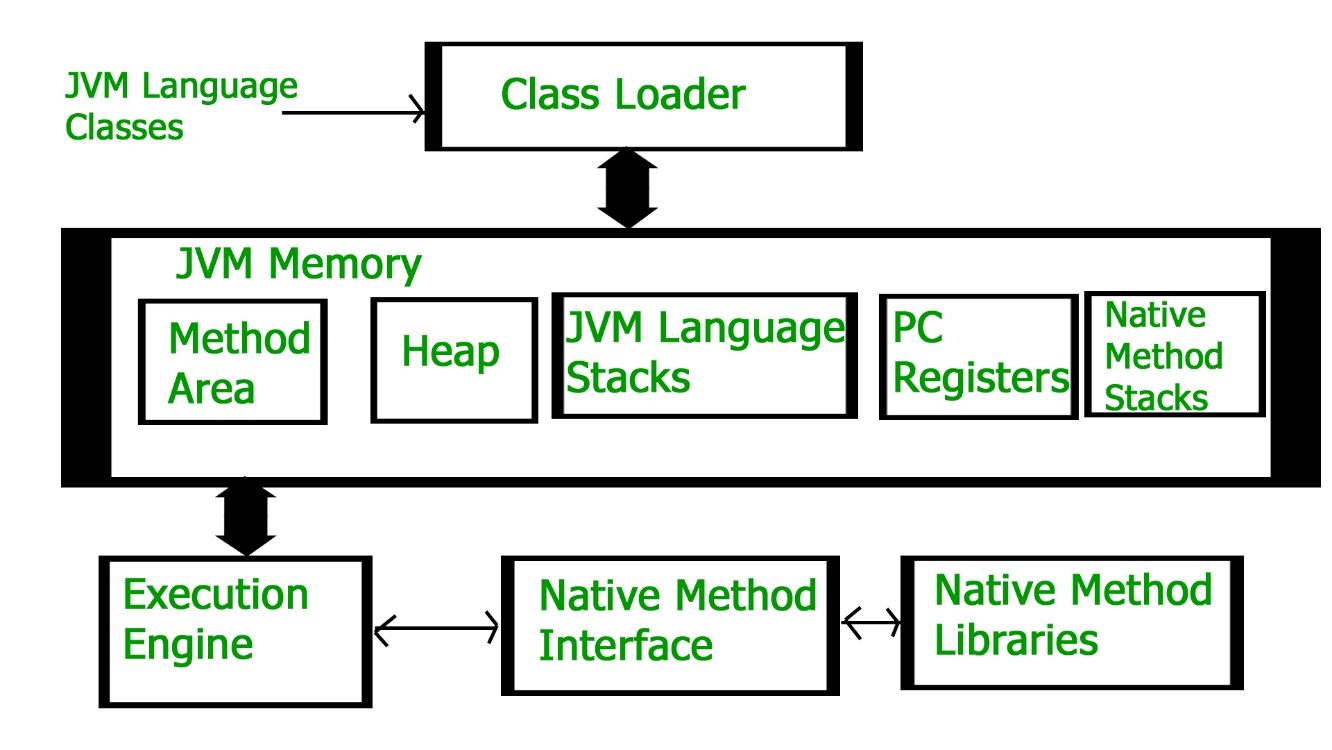
**How does JRE works?**  
To understand how the JRE works let us consider a Java source file saved as *Example.java*. The file is compiled into a set of Byte Code that is stored in a “*.class*” file. Here it will be “*Example.class*“.  
  
  
  
The following diagram depicts what is done at compile time.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

* **Class Loader** 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.  
  


* **Interpreter** 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 JVM Works – JVM Architecture?**JVM 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.  
  
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 the method area. For each “.class” file.  
  
**Linking**: Performs verification, preparation, and (optionally) resolution.

1. **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.
2. **Preparation**: JVM allocates memory for class variables and initializing the memory to default values.
3. **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.   
  
There are three class loaders: ***Bootstrap, Extension, System***

1. **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++.
2. **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$ExtClassLoader class.
3. **System/Application** class loader: It is a child of the extension class loader. It is responsible to load classes from the application classpath. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$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.  


 **1) ClassLoader** The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

**2) Method Area** JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.

**3) Heap** All the [Objects](https://www.guru99.com/java-oops-class-objects.html), their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

**4) JVM language Stacks** Java language Stacks store local variables, and it’s partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.

**5)  PC Registers** PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

**6) Native Method Stacks** Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.

**7) Execution Engine** It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

**8) Native Method interface** The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.

**9) Native Method Libraries** Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine.

**JVM Memory Java (JVM) Memory Model**

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.

  
  
  
  
  
<https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java>



**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.

**Software Code Compilation & Execution process** In order to write and execute a software program, you need the following  
**1) Editor**– To type your program into, a notepad could be used for this  
**2) Compiler**– To convert your high language program into native machine code  
**3) Linker**– To combine different program files reference in your main program together.  
**4) Loader**– To load the files from your secondary storage device like Hard Disk, Flash Drive, CD into RAM for execution. The loading is automatically done when you execute your code.  
**5) Execution** – Actual execution of the code which is handled by your OS & processor