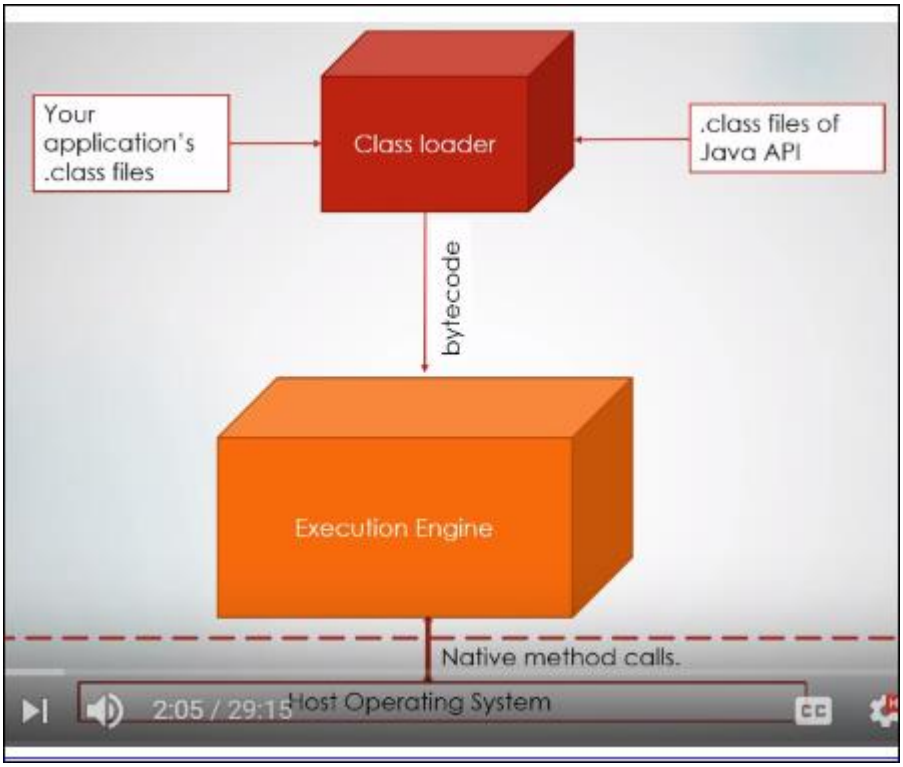
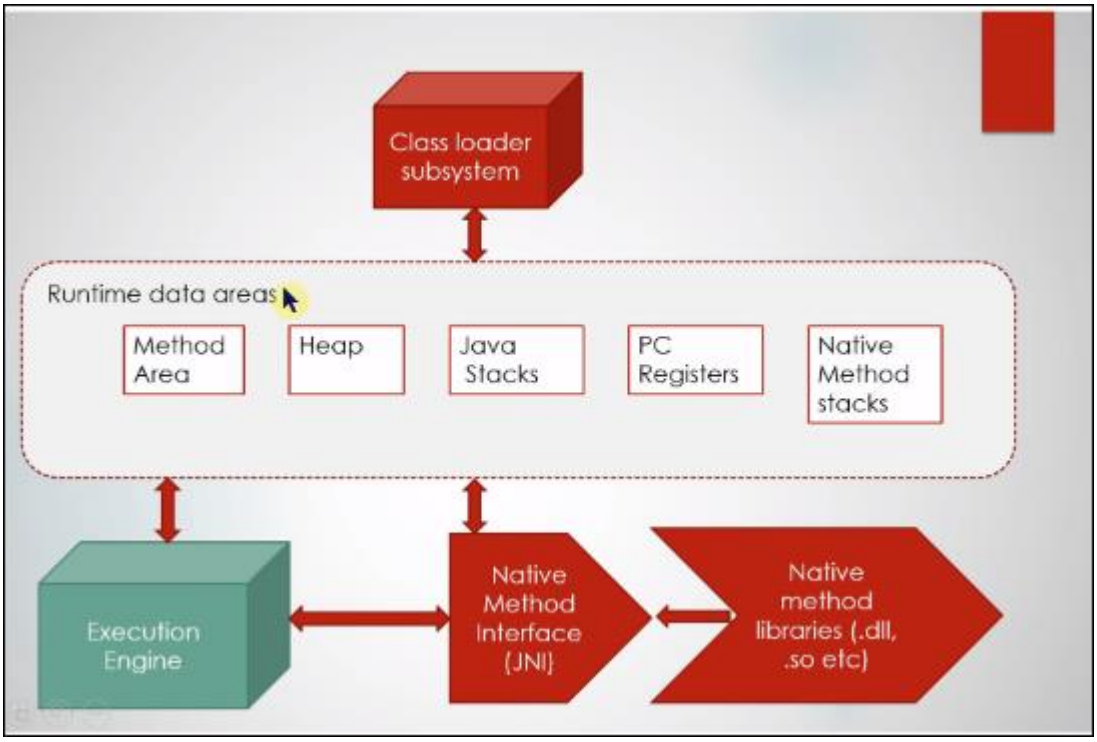
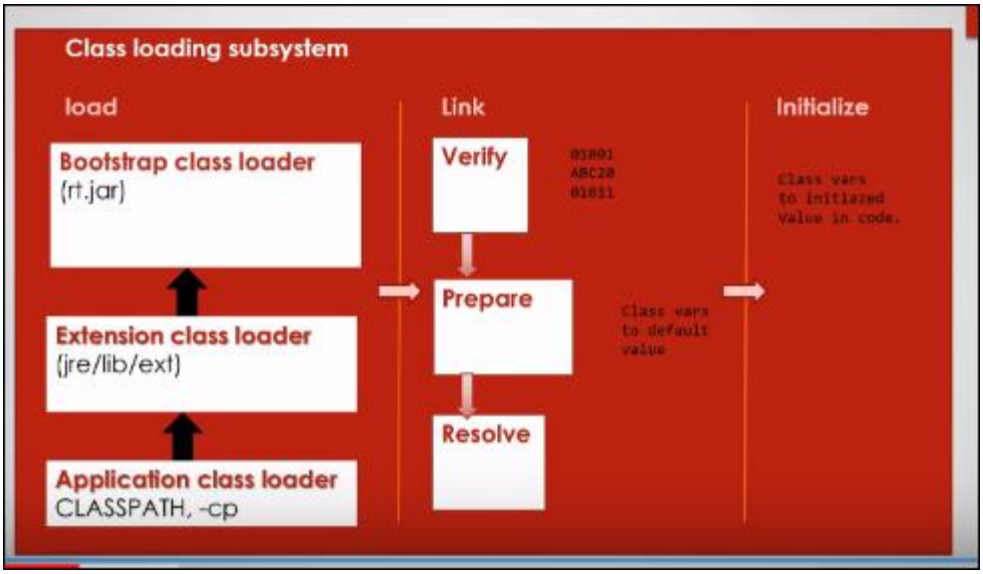
 Contact: 8087883669

JVM Architecture

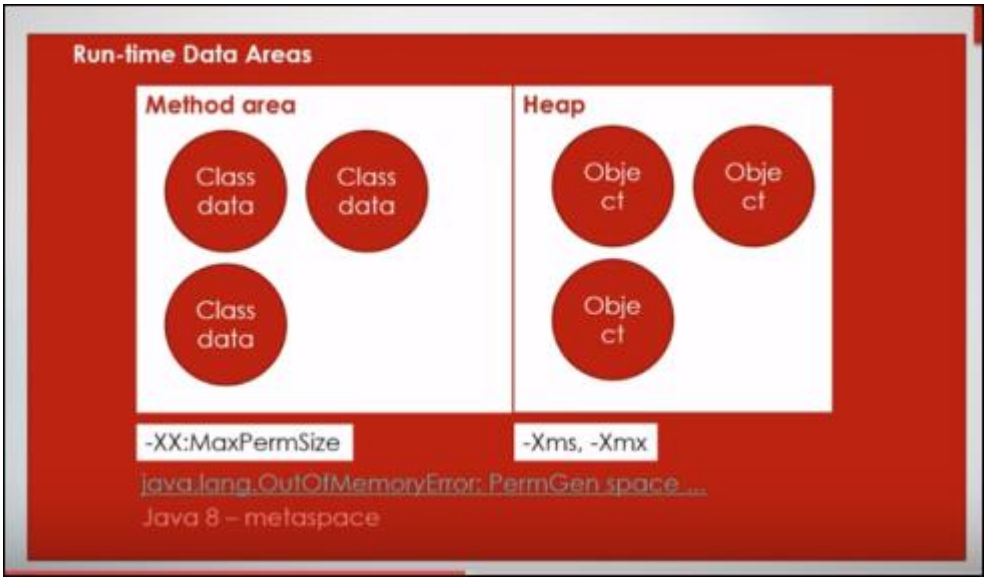


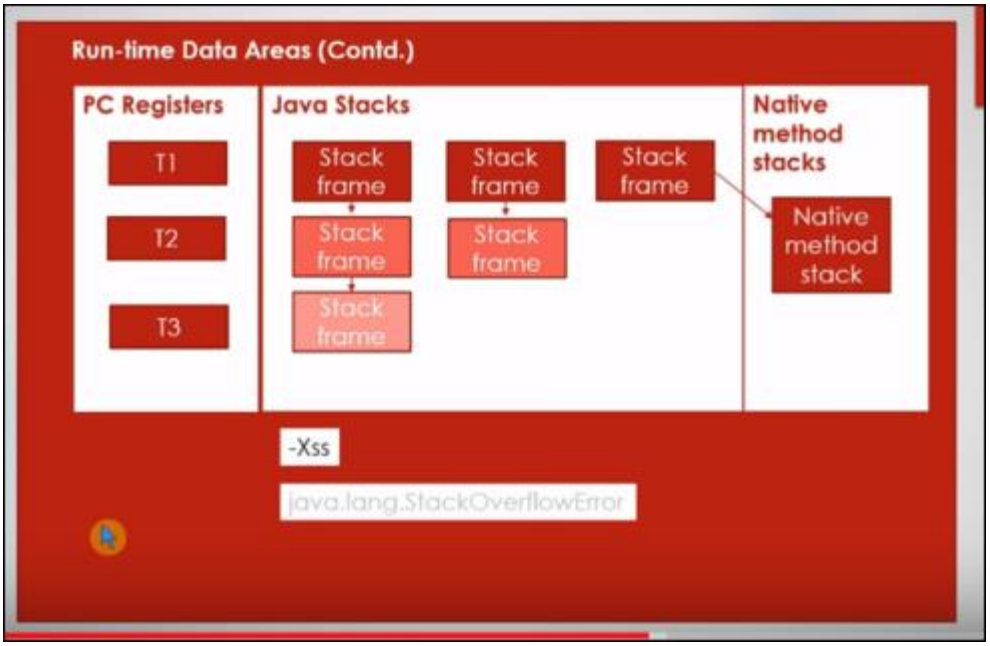
 Contact: 8087883669

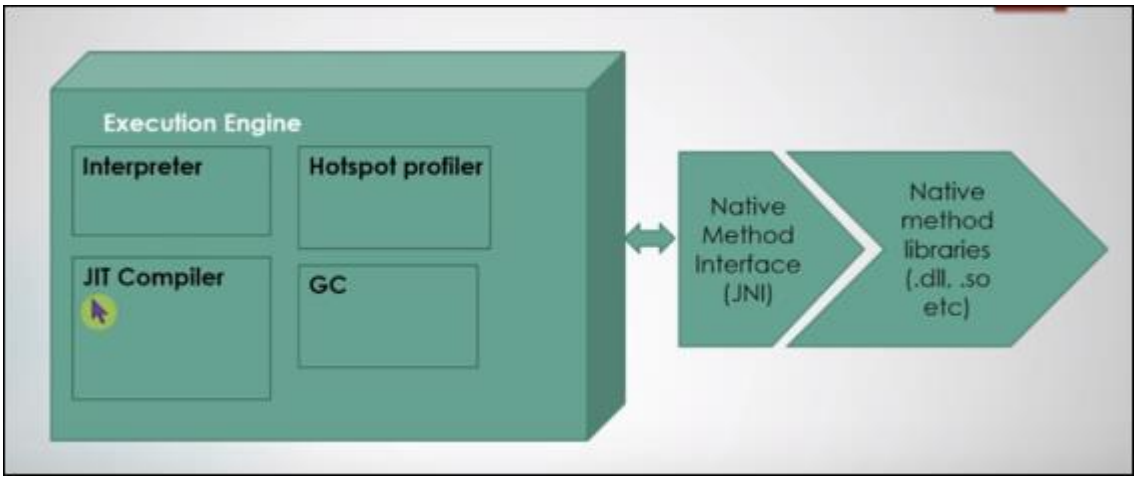




 Contact: 8087883669







The JVM (Java Virtual Machine)

is a crucial component of the Java platform. It is responsible for executing Java bytecode and providing a runtime environment for Java applications to run on various operating systems and hardware architectures. Here's an overview of the JVM architecture:

Classloader: The JVM's classloader subsystem is responsible for loading Java class files into memory. It performs tasks such as locating and loading the necessary class files from the file system or network, verifying their bytecode for security, and preparing them for execution.

Runtime Data Areas: The JVM divides memory into several runtime data areas, each serving a specific purpose:

Method Area: It stores the class-level data, including the bytecode, constants, field and method data, and static variables. Each JVM instance has one method area shared among all threads.

```
Class<? extends Class> aClass = Player.class.getClass();
Method[] methods = aClass.getMethods();
aClass.getDeclaredFields();
```

Heap: The heap is the runtime data area where objects are allocated. It is shared among all threads but provides memory isolation for each individual object. The garbage collector operates in this area to reclaim unused objects.

Java Stacks: Each thread in a JVM has a corresponding Java stack that contains method frames. A method frame holds the state of a method during its execution, including local variables, parameters, and intermediate results. It also manages method invocations and returns.

```
public class StackOverFlowExcepionTest {

    public static void main(String[] args) {
        StackOverFlowExcepionTest.test();
    }
    public static void test(){
        test();//(StackOverFlowExcepionTest.java:12)
    }
}
```

PC Registers: Each thread has its own program counter (PC) register, which keeps track of the current executed instruction.

Native Method Stacks: It is similar to Java stacks but specifically used for executing native methods (methods written in languages other than Java).

Execution Engine: The JVM's execution engine executes bytecode instructions. There are different approaches to implementing the execution engine:

Interpreter: It interprets bytecode instructions one by one and executes them. This approach is portable but less efficient compared to other approaches.

Just-In-Time (JIT) Compiler: Some JVM implementations use a JIT compiler to dynamically translate bytecode into native machine code. The JIT compiler optimizes frequently executed parts of the bytecode, providing performance improvements.

Ahead-of-Time (AOT) Compilation: In recent JVM implementations, AOT compilation is gaining popularity. It translates the entire bytecode into native machine code before execution, eliminating the need for interpretation or JIT compilation. This approach can offer faster startup times and reduced memory footprint.

Native Method Interface (JNI): The JNI allows Java code to interact with native code written in languages such as C or C++. It provides a mechanism for calling native methods and accessing native libraries from within the Java program.

Garbage Collector (GC): The JVM's garbage collector is responsible for automatic memory management. It periodically identifies and reclaims unused objects, freeing up memory and preventing memory leaks.

Overall, the JVM architecture provides a layer of abstraction between the Java code and the underlying hardware and operating system. It enables platform independence, memory management, and runtime execution of Java applications.

Class loader

In the JVM (Java Virtual Machine), there are three main types of class loaders that are responsible for loading classes into memory:

Bootstrap Class Loader: Also known as the primordial class loader, it is the first class loader that gets invoked when the JVM starts. It is responsible for loading essential Java runtime classes from the bootstrap classpath, which includes core libraries (such as java.lang) provided by the JVM implementation. The bootstrap class loader is typically written in native code and is not written in Java itself.

Extensions Class Loader: The extensions class loader is a child of the bootstrap class loader. It is responsible for loading classes from the extension classpath, which typically includes the JAR files in the "jre/lib/ext" directory. These classes provide additional functionality and extensions to the Java runtime environment. If a class is not found by the bootstrap class loader, the extensions class loader is consulted next.

System Class Loader (or Application Class Loader): Also known as the application class loader, it is a child of the extensions class loader. It loads classes from the application classpath, which consists of the directories and JAR files specified by the CLASSPATH environment variable or the "-classpath" command-line option. The system class loader is responsible for loading the classes that make up the application being executed.

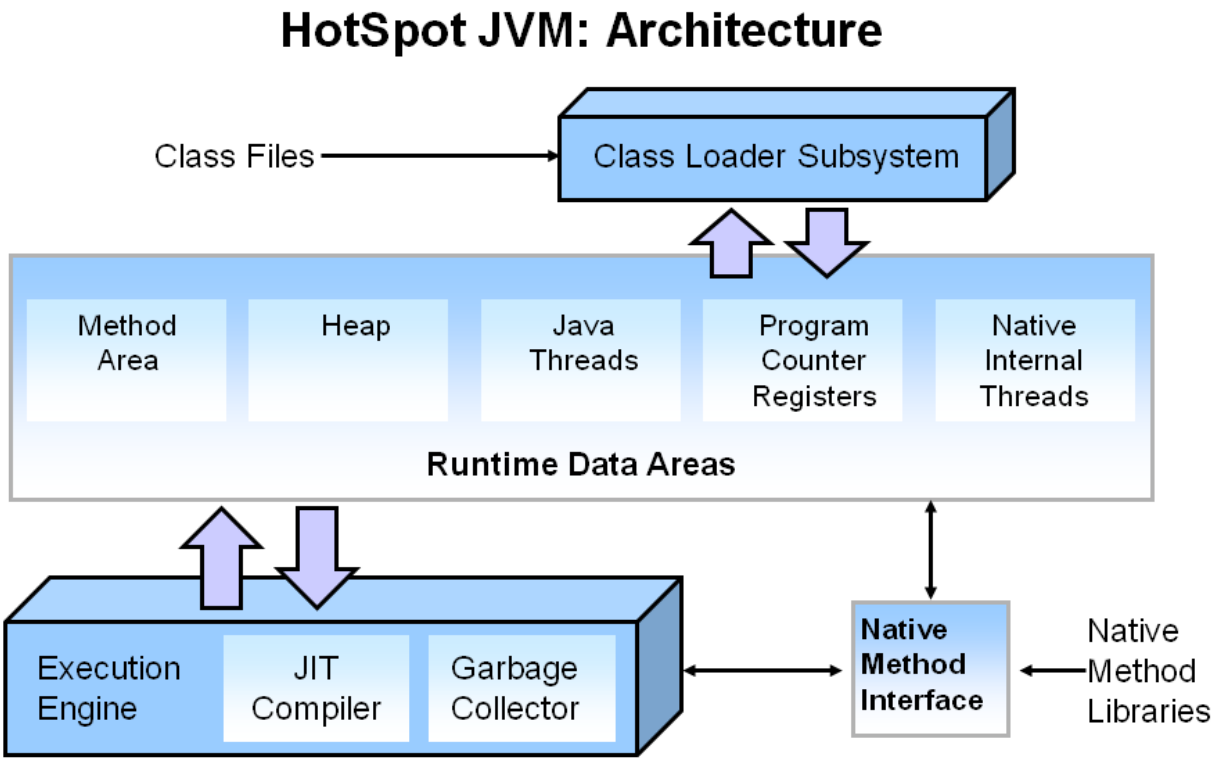
Apart from these three primary class loaders, it is also possible to define custom class loaders by extending the java.lang.ClassLoader class. These custom class loaders can be used to load classes from alternative sources such as databases, network locations, or dynamically generated bytecode. Custom

class loaders can provide additional features like class versioning, class reloading, or security restrictions based on specific application requirements.

The class loaders in the JVM work together in a hierarchical manner. When a class needs to be loaded, the class loaders follow a delegation model, where each class loader first checks if it can load the class and only delegates to its parent class loader if it cannot. This delegation model allows for class reuse and ensures that classes are loaded in a controlled and organized manner.

Hotspot Architecture

The HotSpot JVM possesses an architecture that supports a strong foundation of features and capabilities and supports the ability to realize high performance and massive scalability. For example, the HotSpot JVM JIT compilers generate dynamic optimizations. In other words, they make optimization decisions while the Java application is running and generate high-performing native machine instructions targeted for the underlying system architecture. In addition, through the maturing evolution and continuous engineering of its runtime environment and multithreaded garbage collector, the HotSpot JVM yields high scalability on even the largest available computer systems.



The main components of the JVM include the classloader, the runtime data areas, and the execution engine.

Major GC cleans up the old generation. The task of Major GC is as same as the minor GC, but the only difference is minor GC reclaims the memory of the young generation whereas major GC reclaims the memory of the old generation. It is also said that many major GCs are triggered by minor GCs.

When major GC is triggered?

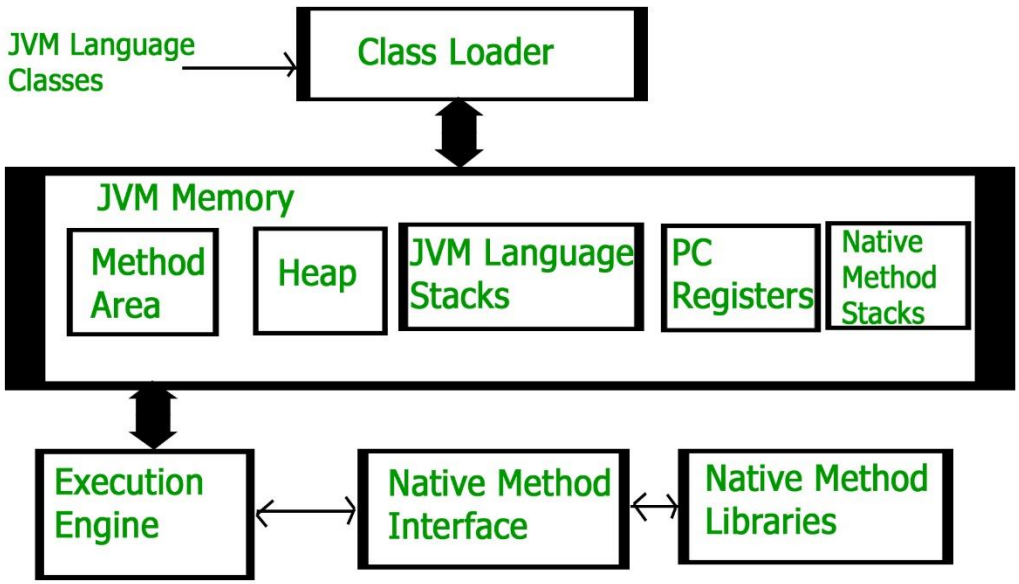
JVM will trigger full GC(both minor+major GC) when the heap is full.13-

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.



Class Loader Subsystem

It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

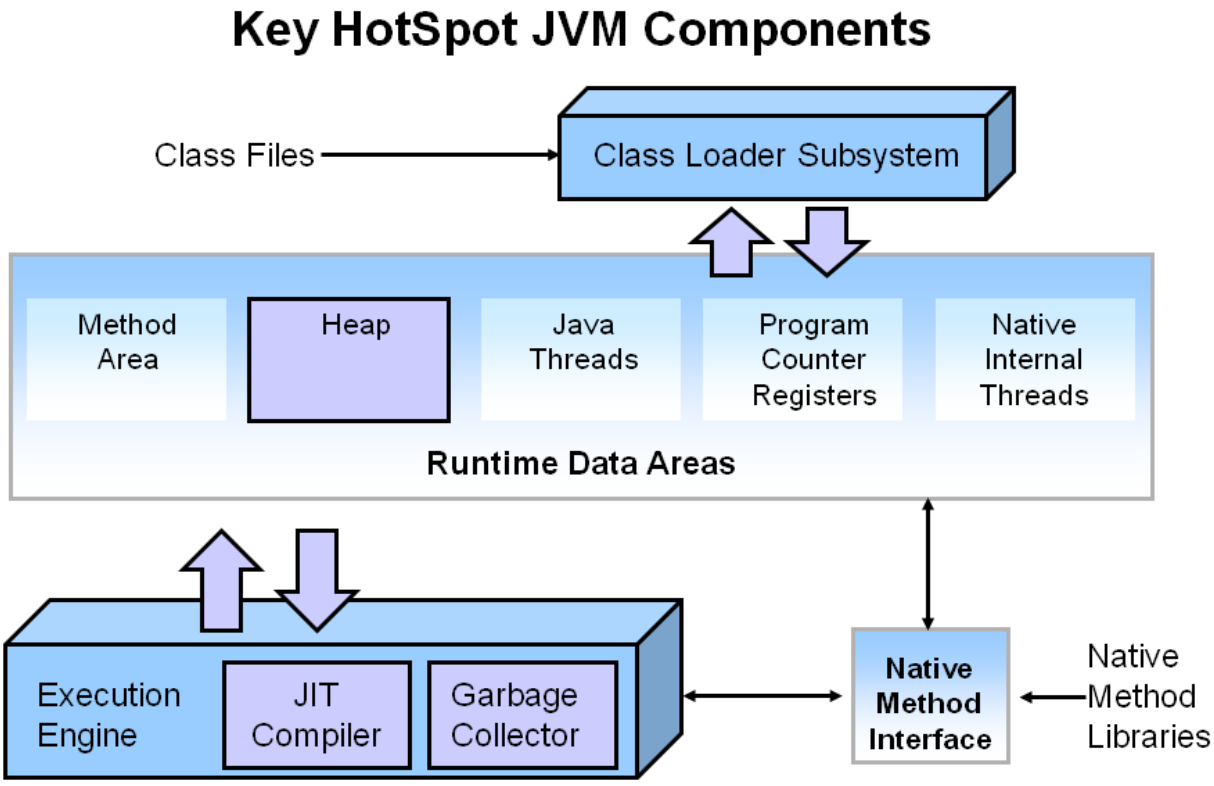
Loading : The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class*file, JVM stores following information in method area.

- 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 class

Key Hotspot Components

The key components of the JVM that relate to performance are highlighted in the following image



There are three components of the JVM that are focused on when tuning performance.
The *heap* is where your object data is stored. This area is then managed by the garbage collector

selected at startup. Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation. The JIT compiler also has a big impact on performance but rarely requires tuning with the newer versions of the JVM.

Performance Basics

Typically, when tuning a Java application, the focus is on one of two main goals: responsiveness or throughput. We will refer back to these concepts as the tutorial progresses.

Responsiveness

Responsiveness refers to how quickly an application or system responds with a requested piece of data. Examples include:

- How quickly a desktop UI responds to an event
- How fast a website returns a page
- How fast a database query is returned

For applications that focus on responsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

Throughput

Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include:

- The number of transactions completed in a given time.
- The number of jobs that a batch program can complete in an hour.
- The number of database queries that can be completed in an hour.

High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick response time is not a consideration.