Chapter 11

JVM Architecture

- In this chapter, You will learn
  - What is VM, JVM, and Types of JVMs, HotSpot technology
  - What will happen when we run java command?
  - **5** phases in a program development, compilation & execution
  - Why Java compiled code is named bytecode?
  - .class file structure
  - JVM: specification, implementation, and instance
  - JVM Architecture Block diagram, JVM Components
  - Explanation on *Class Loader subsystem* (types & algorithm)
  - Class Life Cycle: *Loading, Linking, Initialization*
  - Explanation on *custom ClassLoader* creation
  - Explanation on *Five Runtime Data Areas*
  - Explanation on *Thread and Stack frame* architecture
  - Explanation on *Method Calls* execution flow with PC Register
  - Explanation on *Java Native Interface* (JNI, native method)
  - Explanation on *Execution Engine* (interpreter & JIT Compiler)
  - Draw JVM Architecture with programs execution flow
    - Memory distribution for all 4 types of variables
    - Methods execution control flow with Stack frames
    - Program execution with multiple classes with IS-A & HAS-A relations.
  - *Changing* JVM max & min memory sizes
  - Finding JVM Memory Statistics-
    - Total memory, Free memory, Used memory
  - Stand-alone application lifecycle phases & execution flow
  - JVM Architecture Final Diagram with complete internal details
- By the end of this chapter- you will learn all Components of JVM, Runtime Data Areas, and their functionalities with complete execution flow of static and non-static members of a class.

JVM Architecture

In this chapter we will learn the internals of JVM, different components of JVM, ClassLoader subsystem, classes loading mechanism, runtime areas of JVM, execution engine, JNI, different activities performed inside JVM when class is loaded and object created, memory distribution for all 4 types of variables, methods execution procedure, etc…

You, being a programmer, no need to learn JVM internals. Instead if you can gain some basic knowledge on what is happening inside JVM during our class execution then this knowledge will help you understanding Java program execution easily. This will also help you finding correct output of an unknown program in interview written test in addition to an enhancement of skill to develop high performance applications.

If you compare, Knowing about JVM internals is like knowing about Human anatomy in deep like hands, mouth, throat, stomach, blood and other internal parts which are helping the digestive system to perform properly to generate energy from the food we eat. Of course, here we, no need to know about complete functionality of these parts as we are not doctors and also not going to do any surgery. But by just knowing outer functionality, we can take complete care in getting energy generated by eating proper food, at proper time in a proper way to live more years healthy and happy. By knowing body part functionality we can perform first aid to save life.

So if you compare JVM with your body parts, **Food** is nothing but the **Program** you are given to JVM, then JVM's internal components are like *hand, mouth, throat, stomach* will execute (digest) the given program and gives the output(energy).

As I said in first paragraph, in this chapter we only learn the essential parts of JVM and their functionality. If you are interested to learn complete details of JVM, you can refer "JVM *specification given by Oracle (previously Sun)*" or refer a text book "*Inside the JVM*, by Bill Venners".

Let us learn JVM Architecture starts with definition

Definition of Java Virtual Machine:

- ➔ JVM stands for ***J**ava **V**irtual **M**achine*. It is a Process Based Virtual Machine. JVM is a *Java platform* it is responsible to run Java bytecode by translating them into current OS machine language.JVM provides runtime environment with an equally partitioned identical memory areas for executing Java bytecode in any OS.

The main reason behind the invention of JVM is to achieve platform independency, means JVM is invented for running Java compiled code, bytecode, in all available OS irrespective of the OS in which it is compiled. For this purpose, Oracle Corporation, previously SUN Microsystems, developed separate JVM for every OS for running Java compiled code by translating it into current OS specific machine language.

Virtual and **Virtual Machine**

The concept called **V**irtual **M**achine is not newly invented for Java. The VM concept is already existed in OS and other languages**.** So Let us first get some idea on the word Virtual and Virtual Machine, and then we will learn more about Java VM.

Well, what is Virtual?

- ✓ This real-world is physical. You and your friends in this real-world are physical.
- ✓ Your dream world is virtual. You and your friends in dream world are virtual.

The meaning of **Virtual** is *not physical.* It is a logical representation of one physical thing.

In computer world the meaning of Virtual is:

➔ A logical representation of a physical thing created in a computer through software.

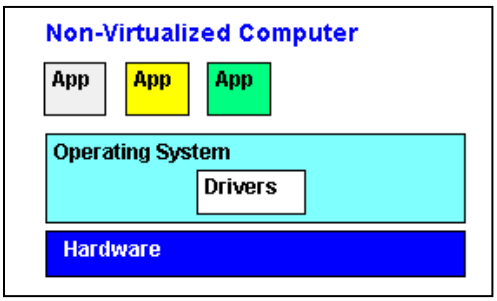
For example:

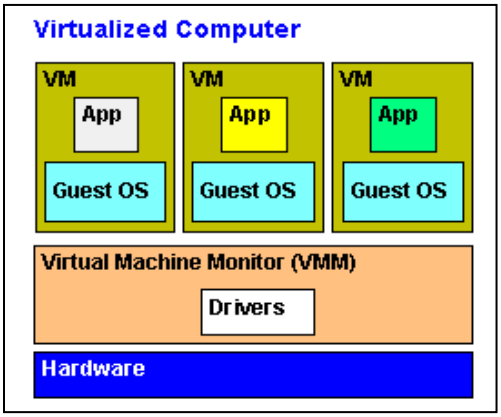
1) A photo album in a computer is a Virtual family.
2) In a Bike Racing game the Bike is a virtual Bike.
3) In a Temple Run Game the person running and jumping is a virtual human being

In above examples- the family, the bike and the person are created virtually using software in computer. They are not physically existed inside computer.

Definition of Virtual Machine

1) A ***Virtual Machine*** is a **software implementation** of a computer machine. It creates an isolated duplication of real computer and allows us to perform operations as we perform operations in a real computer.





Q2) Well, What is the need of Virtual Machine software, when we can perform operations directly in computer?

If we perform operations directly in a computer we will have below problems as shown above

1. In one computer we can install only one OS
2. We can run only current OS specific compiled code in a platform dependent way.
3. We cannot use hardware resources effectively

If we perform operations through Virtual machine software will have below benefits

1. In a single computer we can install one OS in another OS
2. We can run any OS compiled program in current OS in a platform independent way
3. We can use hardware resources effectively

Summary: Purposes of Virtual Machine

Basically Virtual machines are used for below two purposes

1) For installing one OS in another OS in the same computer, installing Linux in Windows.
2) For proving runtime environment for executing a language application in a platform independent way.
3) We can use hardware resources effectively

Types of Virtual Machines

Virtual machines are separated into two major classes, based on their use and degree of correspondence to a real machine:

1. A **System Virtual Machine** (also called as Hardware Virtual Machine)
2. A **Process Virtual Machine** (also called as Language Virtual Machine or an Application Virtual Machine, or Managed Runtime Environment)

System Virtual Machine

1) System virtual machine is a software **emulation** of a computer system.
2) It means System Virtual Machine mimics the entire computer.
3) It provides a platform/runtime enviroment for the execution of a complete operating system.
4) It will create *number of different isolated identical execution environments* in the single computer by partitioning computer memory to install and execute different OS at time.
5) Then allows us to install applications in each OS, run application in this OS as if we work in real computer. *For example* we can install Windows XP/Linux Ubuntu in Windows 7/8 OS with the help of Virtual Machine.

Examples of System Virtual Machine software

1. VMware
2. VirtualBox
3. Parallels
4. QEMU
5. Citrix Xen
6. Windows Virtual PC

Process virtual machine

1) **Process virtual machine** is a software **simulation** of a computer system.
2) It means Process Virtual Machine does not mimics the entire computer.
3) Process VMs provide runtime environment to execute a single program, and supports a single process.
4) The purpose of a Process VM is to provide a platform-independent programming environment that abstracts the details of the underlying hardware or operating system, and allows a program to execute in the same way on any platform.
5) Process VMs are implemented using an interpreter; for improving performance these VMs will use just-in-time compilers internally.

Examples of Process VMs

1. **PVM** - **P**arrot **V**irtual **M**achine for Perl Language
2. **JVM** - **J**ava **V**irtual **M**achine for Java language
3. **CLR** - **C**ommon **L**anguage **R**untime for .NET Framework
4. **PVM** - **P**ython **V**irtual **M**achine for Python language

Types of JVMs:
The Java 2 SDK, Standard Edition, contains two implementations of the Java virtual machine

- Java HotSpot Client VM
- Java HotSpot Server VM

Java HotSpot Client VM:
The Java HotSpot Client VM is the default virtual machine. By default java program is executed with client VM. As its name implies, it is tuned for best performance when running applications in a client environment by reducing application start-up time. The Client version is tuned for quick loading. It makes use of interpretation.

Java HotSpot Server VM:
The Java HotSpot Server VM is designed for maximum program execution speed for applications running in a server environment. The Server version loads more slowly, putting more effort into producing highly optimized JIT compilations, that yield higher performance.

The JVM launcher, *java,* by default starts client VM, if we want to start the Java HotSpot Server VM we must use the java command-line option *-server* when launching an application, as shown below:

```
java Test          -> class Test is executed by client VM
java -server Test  -> class Test is executed by server VM
```

History of HotSpot
HotSpot is a technology it improves performance of application execution. This technology provides techniques such as *just-in-time* compilation and *adaptive* optimization designed to improve performance.

The Java HotSpot Performance Engine, first released in April 27, 1999, built on technologies from the Strongtalk implementation of the Smalltalk programming language originally developed by Longview Technologies, which traded as "Animorphic". Animorphic's virtual-machine technology had earlier been successfully used in a Sun research project, the Self programming language. In 1997 Sun Microsystems purchased Animorphic.

Shortly after acquiring Animorphic, Sun intended to write a new just-in-time (JIT) compiler for the newly developed virtual machine. This new compiler would give rise to the name "HotSpot", which derives from the fact that, as the software runs Java bytecode, it continually analyzes the program's performance for "hot spots" which are frequently or repeatedly executed. These are then targeted for optimization, leading to high-performance execution with a minimum of overhead for less performance-critical code. In one report, the JVM beat some C++ or C code in some benchmarks.

Initially available as an add-on for Java 1.2, HotSpot became the default Sun JVM in Java 1.3.

Features of HotSpot technology

Some of the features of Java HotSpot technology, common to both VM implementations, are the following.

Adaptive compiler

- Applications are launched using a standard interpreter, but the code is then analyzed as it runs to detect performance bottlenecks, or "hot spots".
- The Java HotSpot VMs compile those performance-critical portions of the code for a boost in performance, while avoiding unnecessary compilation of seldom-used code (most of the program).
- The Java HotSpot VMs also uses the adaptive compiler to decide, on the fly, how best to optimize compiled code with techniques such as in-lining.
- The runtime analysis performed by the compiler allows it to eliminate guesswork in determining which optimizations will yield the largest performance benefit.

Rapid memory allocation and garbage collection:

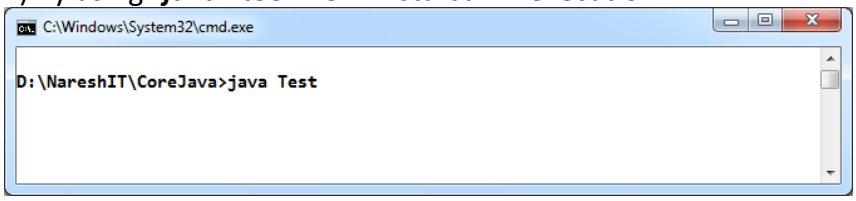
- Java HotSpot technology provides for rapid memory allocation for objects, and it has a fast, efficient, state-of-the-art garbage collector.

Thread synchronization:

- The Java programming language allows for use of multiple, concurrent paths of program execution (called "threads").
- Java HotSpot technology provides a thread-handling capability that is designed to scale readily for use in large, shared-memory multiprocessor servers.

Q) How can we start JVM execution?

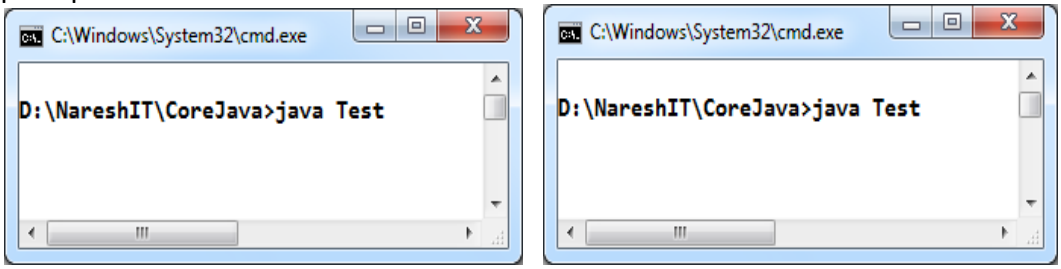
A) By using "***java***" tool we will start JVM execution.



With the above command, JVM process will be launched by occupying some memory from RAM, Test class bytecode will be loaded into JVM memory and executed, output is displayed.

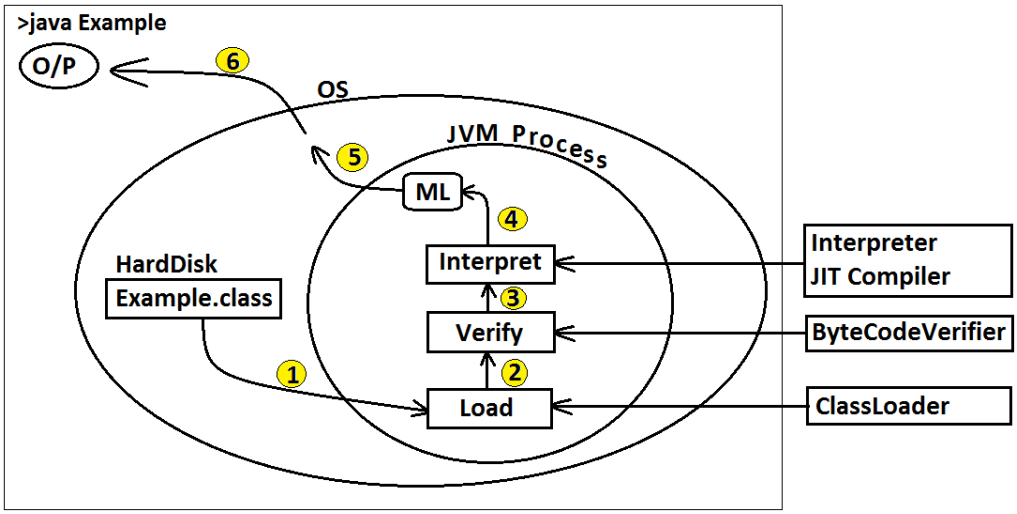
How can we start multiple JVMs at a time?

A command prompt window can hold only one JVM at a time. Then, if we want to start multiple instances of JVM at a time, we must run *java* command from different command prompts at a time as shown below.



What will happen when we run java command?

When we run java command, JVM is created as a process in OS by occupying some memory from RAM. For executing the given class, JVM performs internally below three phases as shown in below diagram:



With respect to above diagram:

1. **Class Loading phase:**

In class loading phase ClassLoader will load given class bytecode in JVM.

Once JVM process is created, JVM will search the given class bytecode in its memory. If the given class is not loaded into JVM, JVM will request ClassLoader to load the given class bytecode. Then ClassLoader will load the given class bytecode for the hard disk folder which is configured in *classpath* environment variable.

If the given class file is not found, ClassLoader will throw exception *java.lang.ClassNotFoundException/java.lang.NoClassDefFoundError* (both are same).

From Java 8 onwards exception is displaying as

Error: Could not find or load main class

2. **Bytecode verification phase:**

In Bytecode verification phase loaded bytecode will be verified.

After class loading phase, BytecodeVerifier verifies the loaded bytecode's internal format. If the bytecode is in JVM understandable format, it allows interpreter to execute loaded bytecode. If loaded bytecode is not in the JVM understandable format, it will terminate execution process by throwing exception "*java.lang.ClassFormatError*".

3. **Execution / interpretation phase:**

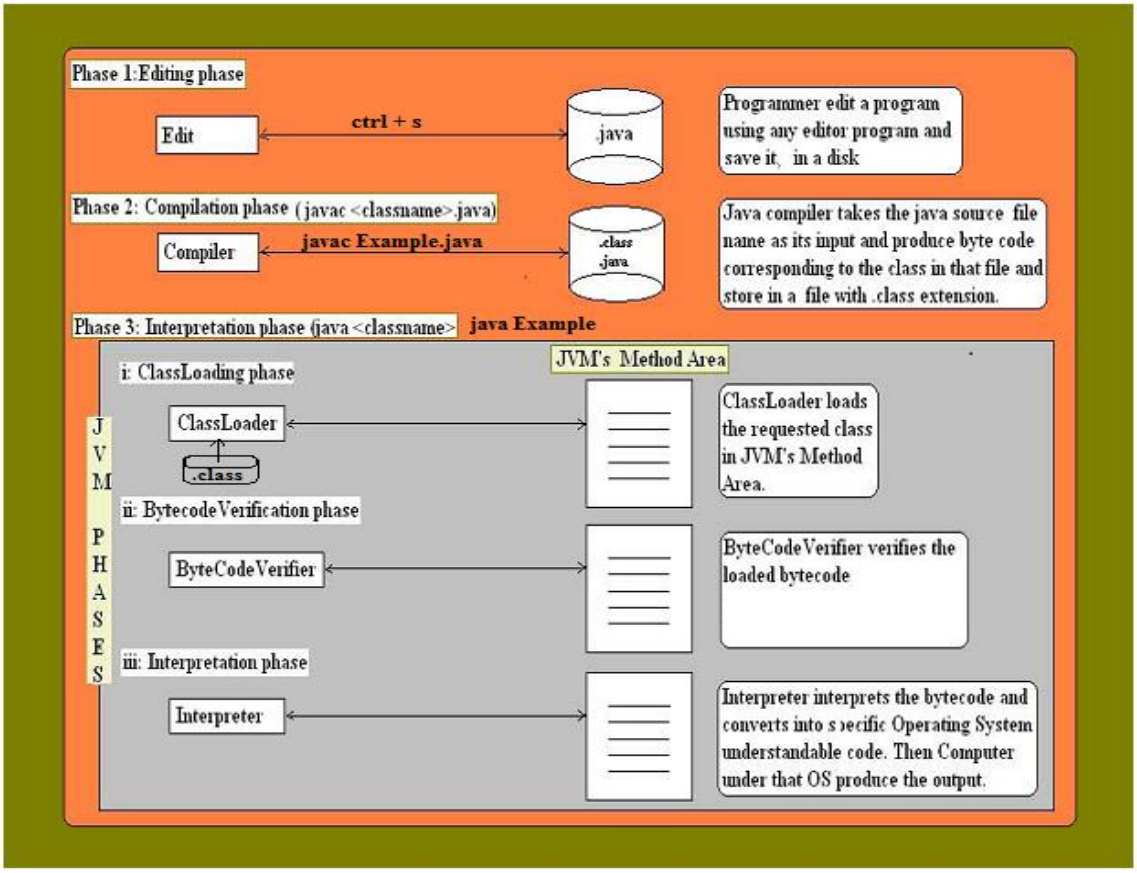
In the interpretation phase loaded bytecode will be executed and displays output.

After byte code verification completed, interpreter will executed loaded bytecode by converting them into current OS machine language. While executing bytecode if any logical mistakes found, JVM will throw associated exception.

Five phases diagram

Java program development, compilation and execution contains five phases, they are

1. Editing phase
2. Compilation phase
3. ClassLoading
4. BytecodeVerification
5. Execution



Q) What is bytecode?

Java bytecode is the *instruction set* of the Java virtual machine. It is an intermediate code of all operating systems. The basic need of bytecode is to achieve platform independency through JVM. Compiler software will convert Java Source code into bytecode, and then JVM will convert bytecode into machine language code of current OS in which it is running.

Q) Why Java compiled code is named as bytecode?

The word bytecode is derived from instruction sets which have one-byte opcodes followed by optional parameters. It means, the Java compiled code is named as bytecode based on what a .class file contains. Actually the .class file is made up of opcode and operands. Each opcode is of size "1 byte", and in the most of the cases the instructions in the .class file are only the opcodes. So the java compiled code is named as *byteopcode,* later they are called as *bytecode*.

The Java bytecode instruction set is made up of 256 opcodes. As of 2015, 198 are in use, 54 are reserved for future use, and 3 instructions are set aside as permanently unimplemented.

JVM: a specification, implementation, and instance

JVM is

1. a Specification
2. a Concrete implementation
3. a Runtime instance

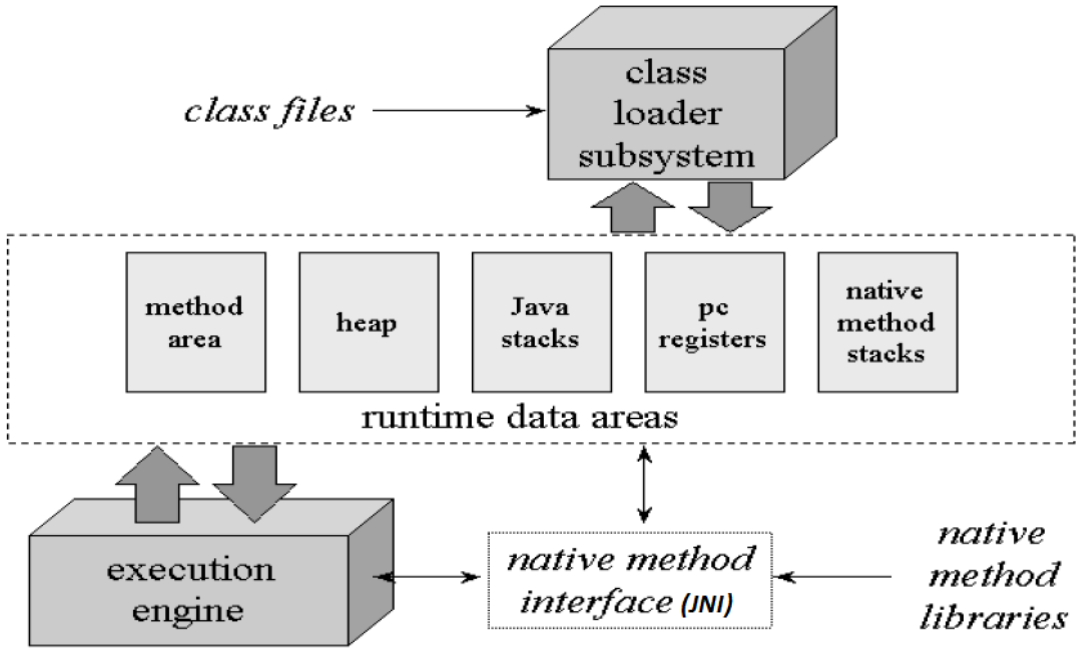
JVM is a specification means it is an abstract documentation it provides set of rules and guidelines for implementing JVM functionality.

JVM is a concrete implementation means it is a software implementation of JVM specification given by different vendors in the world such as SUN, Oracle, IBM, etc... JVM software implementation is available for all verities of OS like windows, Linux, Solaris, Mac OS for running java byte code in all OS in a platform independent way.

JVM is a runtime instance means when we run java command one separate instance of JVM is created for running given Java application.

JVM Architecture block diagram

When we run *java* command for executing an application, one instance of JVM will start as shown below by occupying some memory from RAM. Below diagram will give you pictorial representation of JVM internal architecture with all its internal components:



JVM Components:

As shown in above diagram JVM is a combination of 4 components

1. ClassLoader subsystem
2. Runtime Data Areas
3. Execution Engine
4. Java Native Interface(JNI)

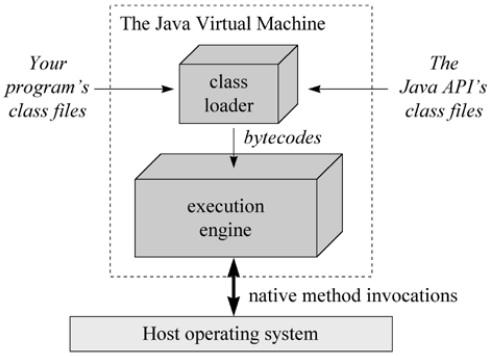
Runtime Data Areas are responsible to provide memory to store byte code, objects, parameters, local variables, return values and intermediate results of computations.

ClassLoader subsystem is responsible to load and store given class byte code in to JVM from both the program and the Java API and also it is responsible to link the loaded class to runtime application of JVM and initialize the loaded class memory.

Execution engine is responsible to execute the loaded byte code with the help of interpreter and JIT compiler.

Java Native Interface (JNI)**:** It is responsible to link Java method to native method, executes native method from a Java program and returns its result to Java program.

Below diagram will show you **the interaction between** *ClassLoader, Execution engine, and JNI***.**

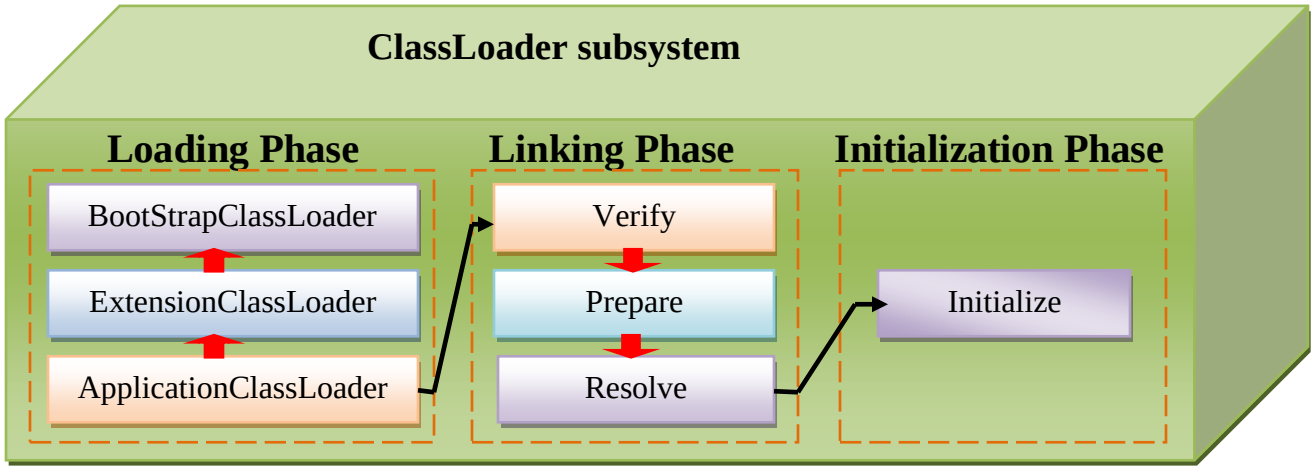


Explanation on ClassLoader subsystem

ClassLoader is a class it is responsible to perform below three activities

➔ Load, Link and Initialize the given class byte code

Below diagram will given you outlook of a class loader subsystem



In **Class Loading phase** ClassLoader will load & store given class bytecode into JVM's Method Area. ClassLoader subsystem contains three types of class loaders for loading core Java API classes and userdefined classes. They are

1. BootstrapClassLoader/PrimordialClassLoader
2. ExtensionClassLoader
3. ApplicationClassLoader/SystemClassLoader

BootStrapClassLoader

1) It is responsible to load Java library classes from BootStrapClasspath i.e.; from jdk\jre\lib\rt.jar file.
2) It is part of the Java Virtual Machine implementation.
3) It is implemented in native API, so it does not have class name.
4) If we try to display its name, null will be displayed.

ExtensionClassLoader

1) It is responsible to load user defined classes or Java library classes from ExtensionClasspath, i.e.; from jdk\jre\lib\ext folder.
2) It is part of Java API implementation.
3) This class loader is implemented in Java with class name ***ExtClassLoader***.
4) This class is defined as inner class in ***sun.misc.Launcher*** class.
5) So, when we try to display its name it is displayed as ***sun.misc.Launcher$ExtClassLoader***.

ApplicationClassLoader

1) It is responsible to load user defined classes from ApplicationClasspath.
2) It will load classes from current working directory or from the folders configured in *classpath* environment variable.
3) This class loader is also implemented in Java with class name ***AppClassLoader***.
4) This class is also defined as inner class in ***sun.misc.Launcher*** class.
5) So, when we try to display its name it is displayed as ***sun.misc.Launcher$AppClassLoader***.

ClassLoader Working procedure:

ClassLoader will work on Parent-delegation algorithm. When ClassLoader gets request to load a class, it will first delegates this class loading request to its parent. The parent class loader, in turn, delegates this request to its parent. This chain of delegation continues through to the bootstrap class loader. In this chain of hierarchy the bootstrap class loader is at the top.

1) When JVM come across a type, it checks for that class bytecodes in method area. If it is already loaded, it makes use of the loaded class bytecode type. If the class is not yet loaded, JVM requests class loader subsystem to load this class.
2) Then ClassLoader subsystem, first handovers this request to *ApplicationClassLoader*.
3) As per the Parent-delegation principle, the application loader, without searching for this class in the folders configured in *Classpath* environment variable, delegates the request to its parent ClassLoader ExtensionClassLoader.
4) Then ExtensionClassLoader, in turn delegates request to its parent ClassLoader BootStrapClassLoader.
5) BootStrapClassLoader is the final parent ClassLoader, then it will search the given class in BootStrapClasspath(jdk\jre\lib\rt.jar and other jar files available in lib folder)
6) If class is found, it will load the class into JVM. If class is not found, BootStrapClassLoader will send request back to ExtensionClassLoader.
7) Then *ExtensionClassLoader* will search this class in *ExtensionClasspath(*jdk\jre\lib\ext folder jar files*)*.

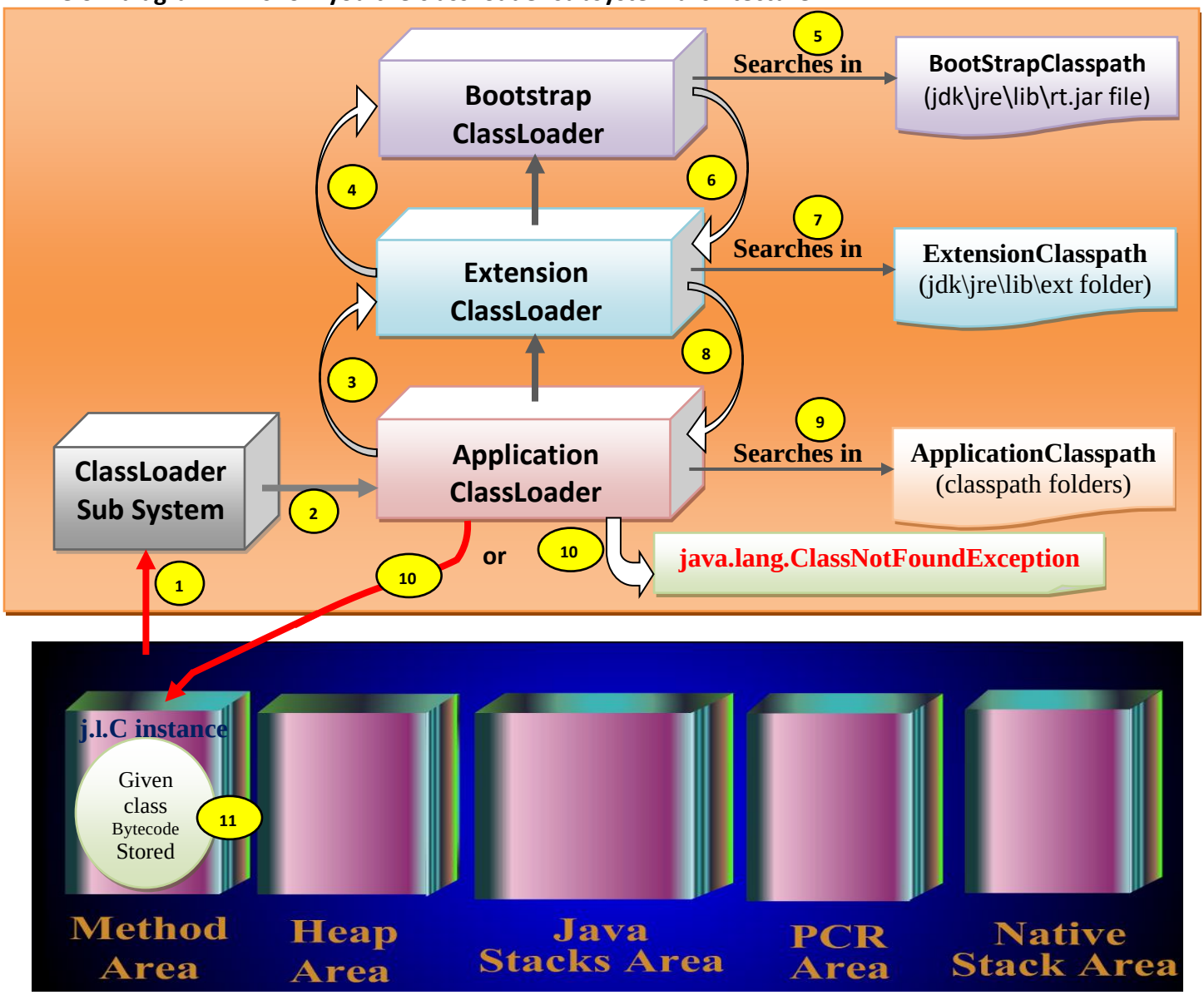
8) If class is found in ext folder jar files, it will load the class into JVM. If class is not found, ExtensionClassLoader sends request back to ApplicationClassLoader.
9) Then *ApplicationClassLoader* will search this class in *ApplicationClasspath(*classpath folders*)*.

10) If class is found in application classpath it will load the class into JVM. If here also class is not found, then we get an exception "*java.lang.NoClassDefFoundError*" or "*java.lang.ClassNotFoundException*".

11) If class is found in any one of the classpaths, the respective ClassLoader will load this class into JVM's method area by creating *java.lang.Class* instance.

12) Then JVM will use the loaded bytecodes to complete the execution of this class.

Below diagram will show you the class loader subsystem architecture



Note: AppClassLoader will load class only if class is not found by Bootstap and ExtClassLoader

Q) If the given class is found in Extension directory and alos in local directory, from which directory this class is loaded and by which class loader?

A) class will be loaded from extension directory by ExtClassLoader

As per parent-delegation algorithm, the extension classpath will be given first priority. Then if class is given class is found in both extension folder and in current working directory, this class will loaded from extension folder by ExtClassLoader.

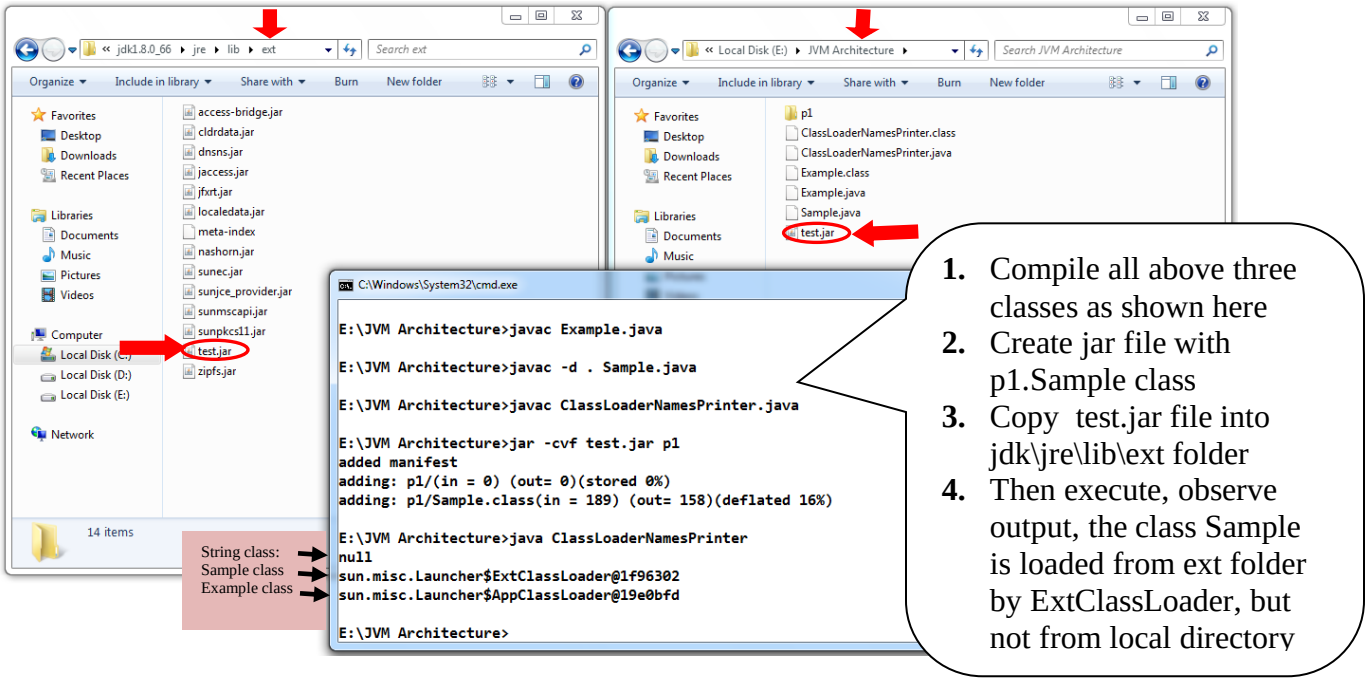
Create and Run below classes to find the class loader through which the given class is loaded

```
//Example.java
class Example{

}
```

```
//Sample.java
package p1;
public class Example{

}
```

```
//ClassLoaderNamePrinter.java
class ClassLoaderNamePrinter{
        public static void main(String[] args) {

                System.out.println( java.lang.String.class.getClassLoader());

                System.out.println( p1.Sample.class.getClassLoader());

                System.out.println( Example.class.getClassLoader());
        }
}
```

Save Example.class in only local folder and Sample.class in both ext folder and local folder, then Compile and Execute above main method class as shown in below diagram



The Class Life Time: Loading, Linking, and Initialization

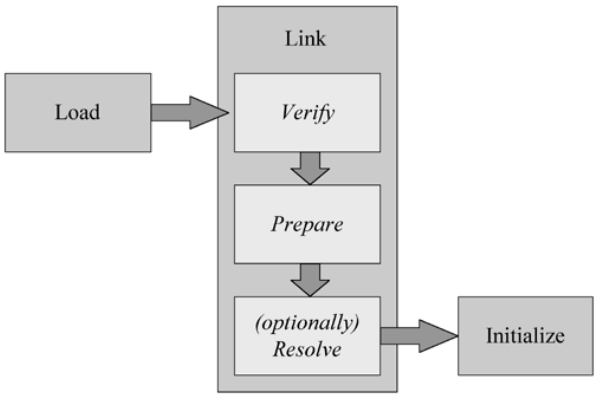
In this section we will discuss the lifetime of a type (class or interface) from its initial entrance into the virtual machine to its ultimate exit. Every class execution will contain below life cycle phases:

1. Loading
2. Linking
3. Initialization
4. Execution
5. Instantiation
6. Garbage Collection
7. Finalization
8. Unloading

➔ Loading, linking, and initialization will occur at the beginning of a class's lifetime
➔ Execution, instantiation (object creation), garbage collection, and finalization will occur in the middle of a class's lifetime and
➔ The finalization and unloading of types will occur at the end of a class's lifetime.

Loading, Linking and initialization

The Class loader subsystem will make types available to the JVM running program through a process of loading, linking, and initialization. It means, the class loader subsystem is not only responsible for loading the binary data for classes in to JVM. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. All these activities are performed in the order strictly as shown in the below diagram:



1) **Loading** is the process of finding the given class, copy its bytecode from .class file and storing its binary form in JVM's method area.
2) **Linking** is the process of incorporating the binary type data into the runtime state of the virtual machine**.** Linking is divided into three sub-steps:
   - **Verification, Preparation, and Resolution.**
   1) **In Verification phase** ClassLoader will the type is properly formed and fit for use by the JVM.
   2) **In Preparation phase** static variables memory is allocated with default values based on their data type**.**
   3) **In Resolution phase** symbolic references in the constant pool are transferred into their direct references**.**
3) **In initialization phase** static block is executed and the static variables are initialized with their initial values given in their declaration**.**
4) **In execution phase class** execution is stated with main method
5) **Once main method execution is completed the** class is unloaded from the JVM

Explanation on Runtime Data Areas

Runtime data areas provide memory to store bytecode, objects, method parameters, local variables, return values, and intermediate results of computations. The Java Virtual Machine organizes the memory it needs to execute a program into several runtime data areas.

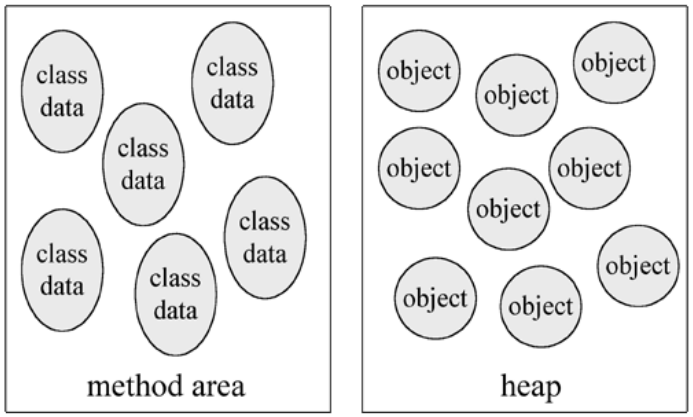
Whenever we run java command for executing a class, the Java launcher, **java**, starts the Java runtime environment as a process in OS by occupying some memory from RAM, and further the entire Java Virtual Machine setup is divided into various run-time data areas that are used during execution of a program. Some of these data areas are created on Java Virtual Machine start-up and are destroyed only when the Java Virtual Machine exits. Other data areas are created per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

JVM totally contains five runtime areas

1. Method Area
2. Heap Area
3. Java Stacks Area
4. Program Counter Registers Area
5. Native Methods Stacks area

About Method Area and Heap Area

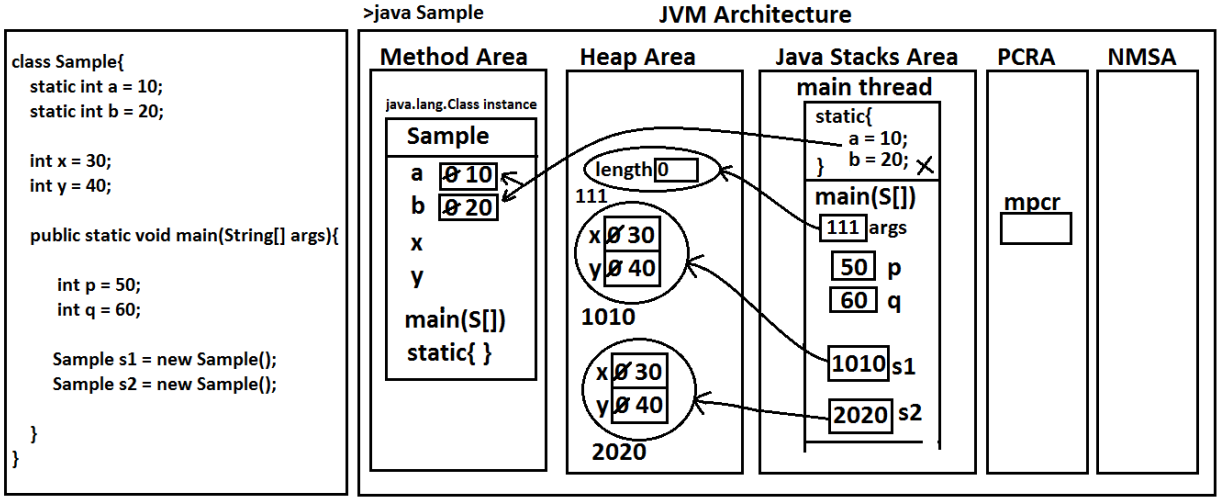
1) In Method area every class bytecode is loaded and stored. It means the given class static and instance variables declaration statements, blocks, methods, and constructors logic will be stored in method area.
2) Further for all *static variables* memory is allocated in this runtime area.
3) Heap is the main memory of JVM.
4) All classes and arrays objects are created in heap area. Means given class instance variables memory is allocated in heap area.
5) For storing given class bytecode inside JVM, one separate instance is created from java.lang.Class. As many classes are loading into JVM those many instances are created for java.lang.Class as shown in the below diagram
6) Below diagram will show you classes bytecode storing and object creation in JVM



7) *Both Method area and Heap area are sharable memory areas to all threads.*
8) Method area and heap area are created on virtual machine start-up.

9) If there is no sufficient memory in method area for loading new class bytecode or there is not sufficient memory in heap area for creating new object, then JVM will throw exception *java.lang.OutOfMemoryError*.
10) The heap memory size can be increased at the time of setting up of runtime environment using non standard option as show in the below command
    java ***-xms <size>*** classname
11) The method area and heap area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary.

Learning JVM architecture theory points is not enough. Let try to understand all above points practically with a program execution with all four types of variable and their memory distribution done inside runtime data areas:



With respect to above diagram:

1. When we run above class Sample, the java launcher, will create JVM as process.
2. Once JVM set up is completed, JVM will search for the Sample class bytecode in Method Area. Since Sample class bytecode is not found, JVM will request class loader subsystem to load the Sample class byte code into JVM.
3. AppClassLoader will load and store Sample class byte code Inside Method Area by creating java.lang.Class object.
4. After bytecode verification is completed, in preparation phase memory will be allocated for static variables a & b in their class context in Method area with default values zero.
5. Then in initialization phase, static block is executed by creating a stack frame inside main thread in Java Stack Area, static variables are initialized with assigned values 10 and 20. Once after static block execution is completed stack frame is destroyed.
6. After initialization, in Execution phase, main method logic is executed by creating separate stack frame in main thread in Java Stack Area. Inside main method stack frame its parameter args, local variables p, q, s1 and s2 memory is allocated.
7. In Heap Area one empty String[] array object, and two objects of Sample class are created. Non-static variables x, y memory is allocated in heap area in their objects. Since we have created two objects from Sample class, two copies of non-static variables memory is created.

8. Once main method execution completed, its stack frame is destroyed, main thread is destroyed and Sample class is unloaded, JVM destroyed.
9. In PC registers area one pc register is created for tracking main thread execution.
10. This complete execution is tracked by execution engine one of the JVM components.

Q) How a class byte code is exactly stored in JVM's method area?

We knew ClassLoader stores the given class bytecodes in JVM by using java.lang.Class object. The java.lang.Class object internally uses several other classes' objects for storing given class bytecode. The classes which are used for storing given class bytecode in JVM are collectively called Reflection API.

Below classes are called Reflection API classes

1. Class
2. Package
3. Field
4. Constructor
5. Method
6. Modifier

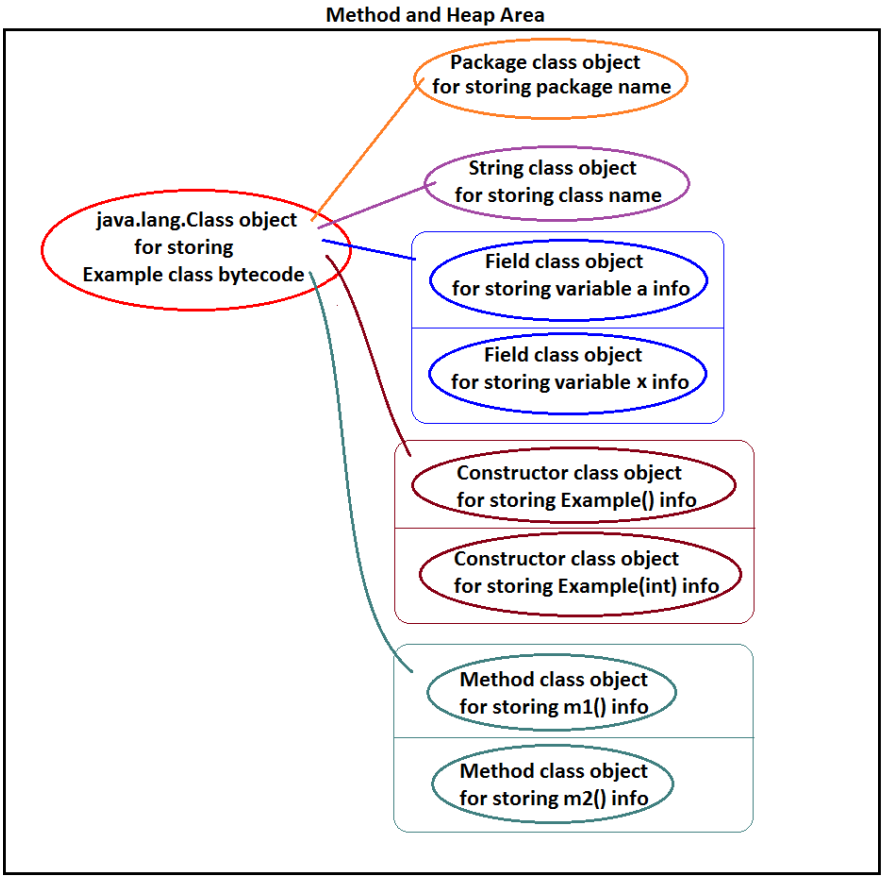
These 2 classes are available in ***java.lang*** package

These 4 classes are available in ***java.lang.reflect*** package

Below diagram will show you *java.lang.Class* object structure for storing given class bytecode using above 6 classes.

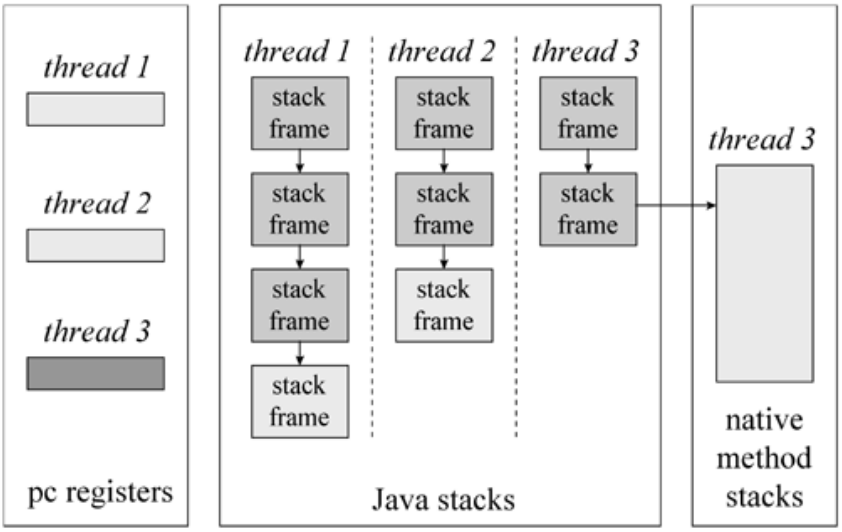
Example.java / Example.class

```
package com.nareshit.jvm;
class Example{

    static int a = 10;
           int x = 20;

    Example(){
        System.out.println("NPC");
    }
    Example(int x){
        System.out.println("IPC");
    }

    static void m1(){
        System.out.println("m1");
    }
    void m2(){
        System.out.println("m2");
    }

}
```



About Java Stacks area

1) In this runtime area all methods (non-native), blocks and constructors logic is executed.
2) In this runtime area we can create several threads for executing methods at a time concurrently. JVM by default creates minimum two threads, they are
    1) main thread
    2) garbage collector thread
3) Main thread is responsible to execute Java methods starts with main method, also responsible to create objects in heap area if it finds "new" keyword in any method logic
4) Garbage collector thread is responsible to destroy all unused objects from heap area.
   Note: Like in C++, in Java we do not have destructors to destroy objects.
5) As each new thread comes into existence, it gets its own pc register and Java stack as shown in the below diagram. The Java stack is composed of stack frames (or frames). A stack frame contains the state of one Java method invocation.
6) When we invoke a method or constructor a new stack frame is created. Once this method execution completed this stack frame is destroyed. Stack Frame holds logic of this method, its parameters, local variables, its return value, and intermediate calculations. Logic of this method or constructor is loaded in this from method area



7) Above diagram shows a snapshot of a virtual machine instance in which three threads are executing. Threads one and two are executing Java methods. Thread three is executing a native method. As shown in above diagram, in this book we will draw stack diagram, the stacks are growing downwards. The "top" of each stack is shown at the bottom of the figure.

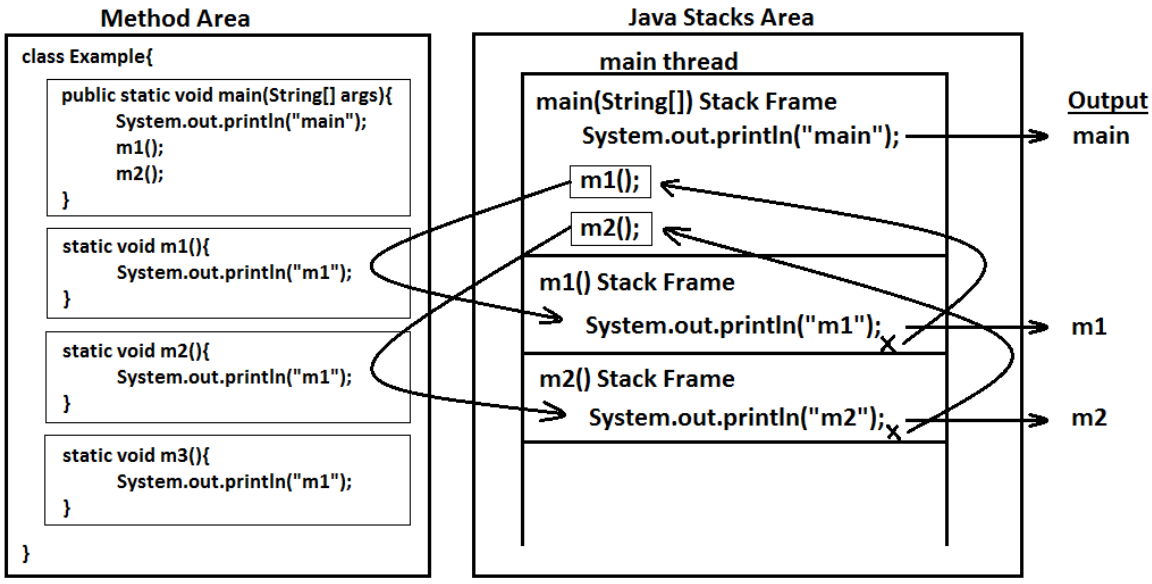
   Note: Inside a thread if there is no sufficient memory for creating new stack frame for execution a method, then JVM will throw exception *java.lang.StackOverflowError*.

Thread Architecture

1) Thread is an independent sequential flow of execution created in JSA.
2) Every thread is internally divided into multiple sub blocks.
3) This sub block technically called as stack frame.
4) As many methods as we invoke those many stack frames are created in this thread sequentially.

5) Inside a thread only one stack frame is active at any point. When we invoke a method from the currently executing method, this current frame is paused, new frame is created for this invoked method, control is sent to this new frame, this invoked method logic is loaded into this frame from method area, logic is executed. After this method execution is completed, its stack frame is destroyed, control return to previous method stack frame from which this method is invoked.
6) If this method has any parameter and local variables, they are all created inside this method's stack frame, and are destroyed automatically when Stack Frame is destroyed.
7) Note that a frame created by a thread is local to that thread and cannot be referenced by any other thread, its means it is not sharable like method area and heap area.

Below diagram will show you main thread with several methods stack frames



With respect to above program

1) Example class byte code is loaded and stored in method area
2) Then immediately JVM starts main method execution by creating a separate stack frame in Java stacks area in main thread.
3) As part of its logic we have called m1() method, then a new stack frame is created for m1() method execution, main method execution is paused, m1() logic is loaded into new stack frame, logic is executed, output is displayed. After m1() execution completed, stack frame is destroyed, control return to main method.
4) In next line of main method, we called m2() method. Then again new stack frame is created, m2() method logic is loaded, logic executed, output displayed, after its execution is completed, stack frame is destroyed, control is return to main method.
5) In main method we do not have more statements to execute, main method execution is completed, and main method stack frame is destroyed, main thread destroyed, JVM destroyed.
6) After all stack frames are destroyed, thread is destroyed.
7) After all threads destroyed, JVM is destroyed.

Stack Frame architecture

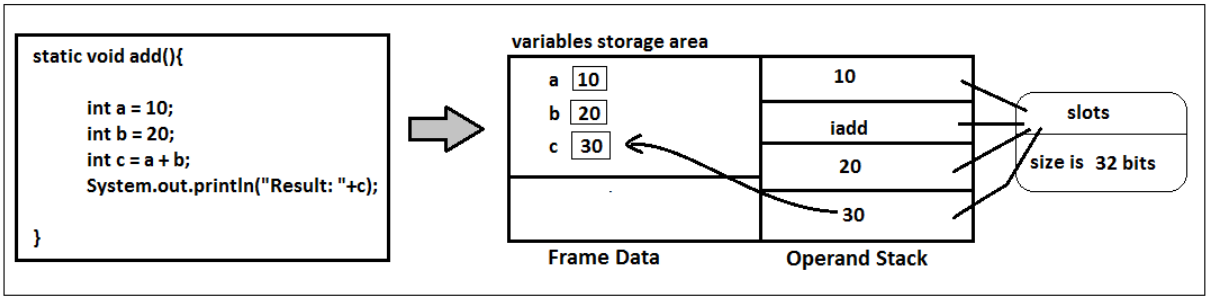
1) Stack Frame is sub block of a thread. When we invoke a method JVM will creates new stack frame, loads the method logic, and executes this logic.
2) Stack Frame is internally divided into three blocks for executing method logic

They are:

1. Local Variables storage area
2. Operand stack (or method stack)
3. Frame data (or Reference to the runtime constant pool of this method)

- All parameters and local variables memory is created in Variables storage area
- Method logic is executed in operand stack. To execute method logic, operand stack is divided into number of blocks. Each block is called slot. This slot size is 32bits (4 bytes - int/ float data type depending of the variable type).
- Due to the slot size 4 bytes the result value will be return as 4 bytes value. So due to this reason in an expression byte, short, char data types are automatically promoted to int data type. So, the result will coming out from an expression is *int* type.
- The frame data portion will contain data to support constant pool resolution, normal method return, and exception dispatch information.

Below diagram shows Stack Frame architecture for add method



Some more details of stack frame with "this" and "super" keywords
We will learn in static & Non-static members chapter.

About Program Counter Registers Area

1) In this runtime area, a separate program counter register is created for every thread. It is used for tracking execution in this thread by storing its instruction address.
2) PC Register is created when new thread is created; after thread execution is completed it is destroyed automatically.

About Native Methods Stacks Area

1) In Native Methods stack area all native methods are executed.
2) In Java, there are two kinds of methods: **Java methods** and **native methods.**
   1. **A Java method** is written in the Java language, compiled to bytecodes, and stored in class files.
   2. **A native method** is written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular processor.

3) To create native method, we must place native keyword in its prototype and it should not have body. A Java program interacts with the host OS by invoking native methods.

For example

```
class Example{
        public static native int add(int x, int y);
        public static void main(String[] args)  {
                add(10, 20);
        }
}
```

It is a native method.

The above program add(int, int) method is a native method. It is compiled fine, but in execution JVM will throw exception: ***java.lang.UnsatisfiedLinkError,*** *because* we just defined native method, but we did not defined it required C program and not linked.

Q) If we define a program in C or C++, how can we link this native logic to native method?

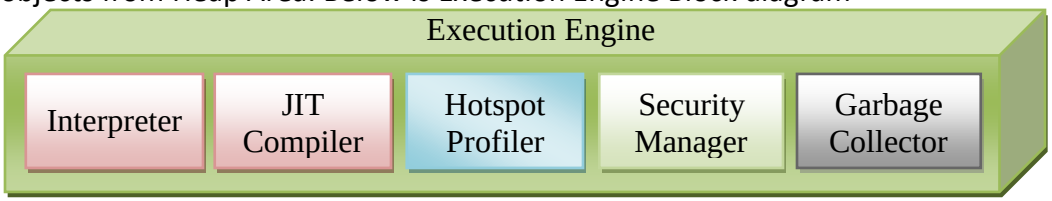
A) By using *JNI* - Java Native Interface. It is a mediator between Java native method and original native method for linking its method calls.

About Java Native Interface (*JNI*):

It is one of the JVM components. It is responsible to link Java method to native method, executes native method from a Java program and returns its result to Java program.

About Execution Engine:

1) It is one of the JVM Components. It is responsible to execute java bytecode.
2) All executions happening in JVM are controlled by Execution Engine.
3) It uses interpreter, JIT compiler, Profiler, SecurityManager for executing a program
4) Execution Engine will also contains Garbage Collector for destroying unreferenced objects from Heap Area. Below is Execution Engine Block diagram



5) Execution engine will comes in two implementations
   1) The simplest kind of execution engine is just with interpreter, it just interprets the bytecodes one at a time.
   2) Another kind of execution engine is a just-in-time compiler, it is faster but requires more memory.
6) JIT Compiler will increase the performance of program execution.
7) JIT Compiler will find the code which is executed repeatedly, generates machine language for this code only once, buffer it and reuses the same machine language code for repeated execution. This is how JIT Compiler will give high performance in execution

JVM Architecture final diagram with conclusions

If you really want to learn more details about execution flow of your byte code do below three things

1. **After every class you compile, read its byte code using below command**

   javap –verbose classname

2. **Study a Book called** *Inside the Java Virtual Machine - by Bill Venners*