What is JVM?

JVM (Java virtual machine) is an abstract machine which works as software simulation and helps computer to run Java program.

Main job of JVM is to load and run java class file.

How many types of virtual machines are there?

Virtual machines are mainly divided into two types:

1. Type1: Hardware or System based
2. Type2: Software or Application or Process based

Which type of virtual machine JVM is?

It comes under software/application/process based virtual machine, which acts as runtime engine to run Java program.

What is the advantage of Hardware based virtual machine, give some example and who is responsible for these kind of virtual machine?

Advantage:

This enables effective utilization of hardware resources.

Example:

VMware, Xen, VirtualBox, KVM …

Responsibility:

IT or Admin team is responsible for these kind of virtual machine.

What is the main role of JVM and different component involved in this role?

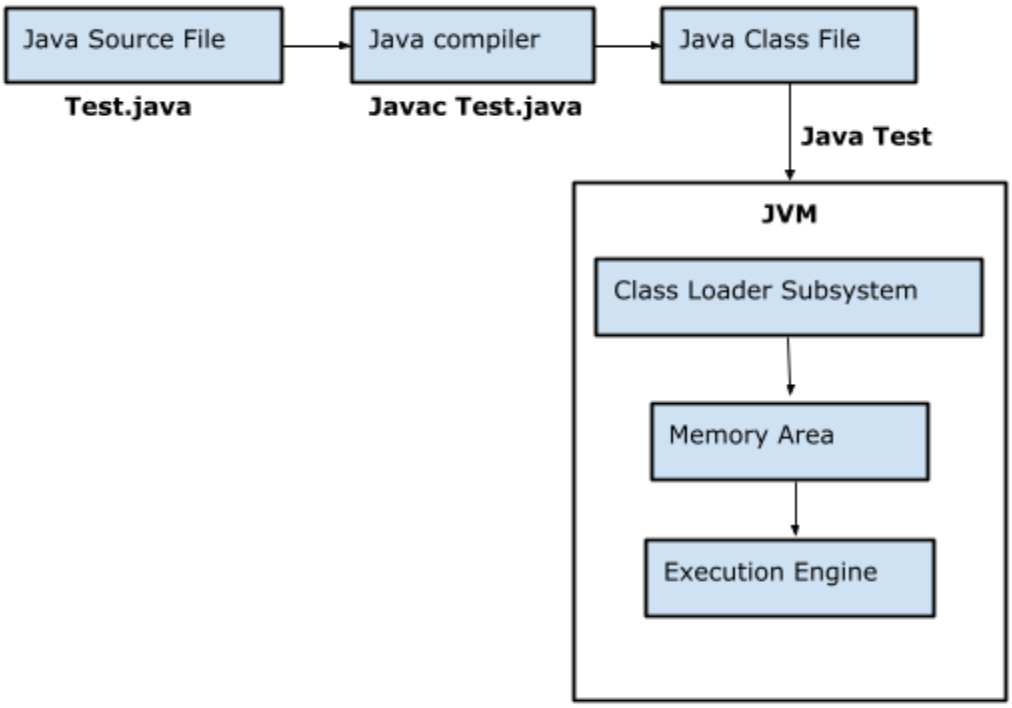
Main job of JVM is to read, load and run Java program line by line. It takes Java compiled code (.class file) as input.

Main component of JVM:

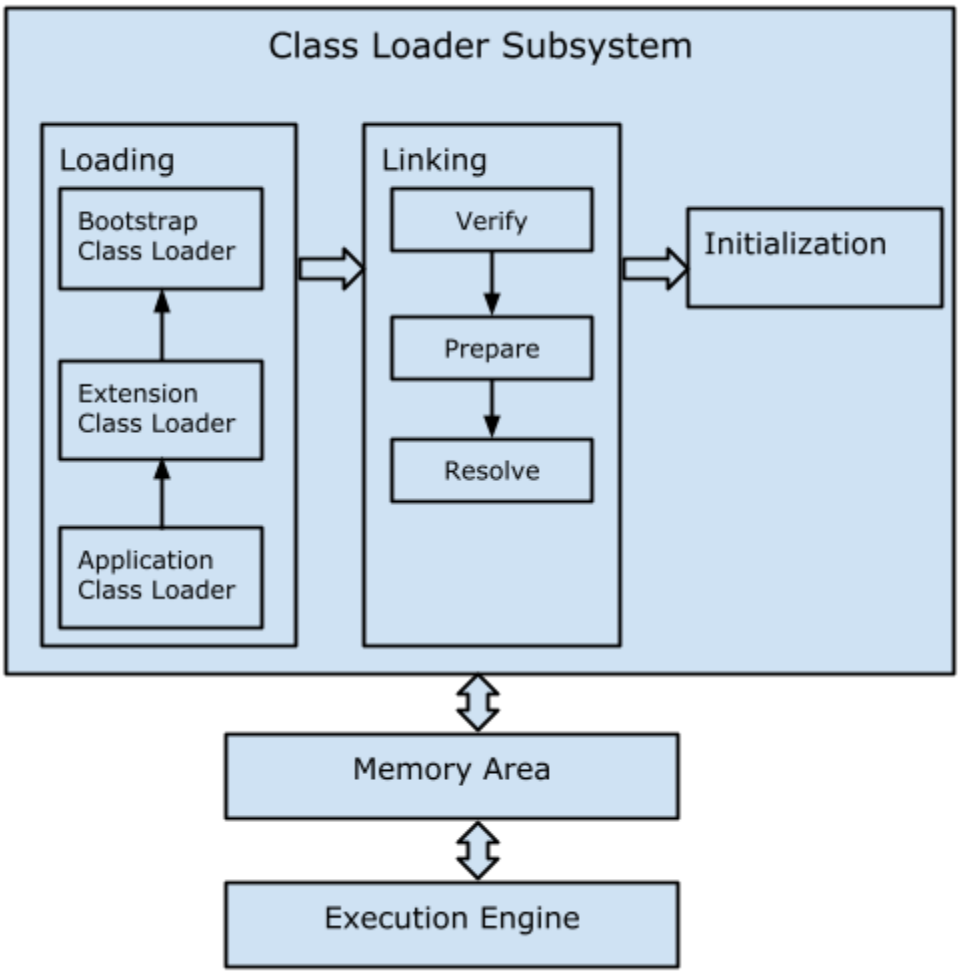
1. Class loader subsystem (Responsible to load class file)
2. Memory Area (Class loader loads class file in some memory area)

3. Execution engine (It actually executes class files line by line)

Basic Flow of java file execution:



What is role of Class loader subsystem?



Class loader subsystem is divided into three components based on roles:

1. Loading
2. Linking
3. Initialization

1. **Loading:**
   Read class file from file system and store corresponding binary data into JVM method area.
2. **Linking:**
   Verify compiled code, allocate memory for static variables and assign with default value, and replace all symbolic references with actual reference.
3. **Initialization:**
   Assign actual values for static variables.

Explain functionality of Loading in details:

When it sees any class name written in program, class loader performs below operations:

- Read class file from file system and store corresponding binary data into JVM method area. It stores complete metadata with below information for loaded class:
    - Fully qualified name
    - Immediate super class
    - Represents Interface/class/enum?
    - Constant
    - Method information
    - Variable information
    - Modifier information
    - Constructor information
- Load parent class too.
- After loading loader will create object of Class class which can be used by developer to read binary data. Class class object is used to represent all .class information.

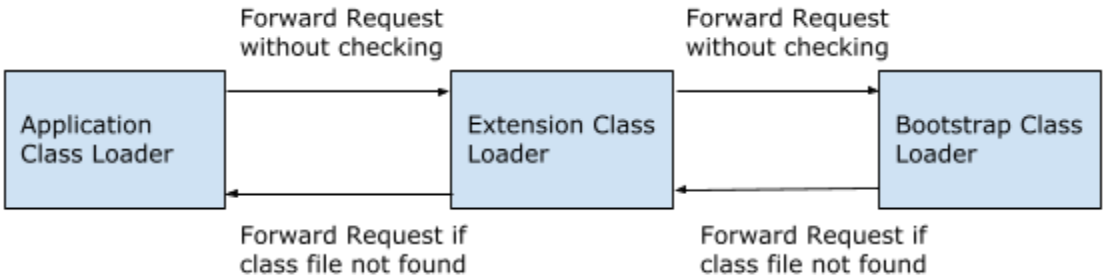
How does loader loads class and how many class loaders are there?

Loader loads classes with the help of below 3 class loaders:

1. **Bootstrap Class Loader**
   This loads classes present inside rt.jar (jdk/jre/lib/rt.jar), here rt in rt.jar stands for runtime jar which includes all basic apis provided by java (ex: lang package, util package ..)
2. **Extension Class Loader**
   This loads classes present inside extension folder (jdk/jre/lib/ext)
3. **Application or System Class Loader**
   This loads classes present in application. Actual classes developed by developer.

In which algorithm does class loader works and which gets higher priority?

Class loader works in delegation hierarchy algorithm.



Below are the steps which class loader follows:

1. It sends request to Application class loader to load class
2. Application class loader forward that request to Extension class loader, it does not check whether class is available to load.
3. Extension class loader forward that request to Bootstrap class loader, it also does not check whether class is available or not.
4. Bootstrap checks for that class and loads if found within rt.jar otherwise forward that request again to Extension class loader.
5. Since request has forwarded from Bootstrap class loader, this time Extension class loader checks whether class available to load, if found in ext folder it loads otherwise it forward that request to Application class loader.
6. Since request has forwarded from Extension class loader, this time Application class loader checks whether class available to load, if found in application path it loads otherwise it throws "ClassNotFoundException" exception.

With above flow it is clear that Bootstrap gets higher priority to load class.

Why above algorithm is called delegation hierarchy?

Because whenever we send request to any class loader it simply delegates that request to its parent class without checking if it can serve that request, it perform some operation only when it gets request from its parent class.

Application Class Loader is subclass of Extension Class Loader which is subclass of Bootstrap Class Loader.

Which class implements which class loader?

Application Class loader: sun.misc.Launcher$AppClassLoader
Extension Class Loader: sun.misc.Launcher$ExtClassLoader
Bootstrap Class Loader: This class loader is written in native code not in Java.

If Bootstrap class loader is written in native code and not in Java, how can Extension class loader (which is written in Java) inherits Bootstrap class?

For Extension class loader, Bootstrap class acts as parent but there is no exact parent-child relationship of inheritance.

How can developer access information about loaded class?

After loading class, loader creates object of Class class for each loaded class, which developer can use to read class information. Class class object is used to represent all .class information. Same like any other object, this also gets stored in heap memory.

Ex:
Class c = Class.forName("pkg.Test");
Here pkg is package name and Test is class name.

There are many method available to get information about class -

isInterface / isArray / isEnum: To check if it is interface / array / enum
getInterfaces: Get all implemented interfaces
getDeclaredFields: Get all declared fields inside class
getDeclaredMethods: Get all declared methods inside class
getFields: Get all declared fields inside class, super class hierarchy (till Object class) and implemented interface.
getMethods: Get all declared methods inside class and super class hierarchy (till Object class).
getConstructors: Get all constructors

How to get information about declared variables and methods inside any class?

Create Class class object and call getDeclaredMethods() to get declared method and getDeclaredFields() to get declared fields.

```
public class Main {
	public static void main(String[] args) throws ClassNotFoundException {
		Class c = Class.forName("testpro.Test");
		System.out.println(c.getName());
		Method[] methods = c.getDeclaredMethods();
		for (Method m : methods) {
			System.out.println(m);
		}
		Field[] fields = c.getDeclaredFields();
		for (Field f : fields) {
			System.out.println(f);
		}
	}
}
class Test {
	public int var1;
	public String var2;
	public void method1 () {}
	public void method2 () {}
}
```

O/P:
testpro.Test
public void testpro.Test.method1()
public void testpro.Test.method2()
public int testpro.Test.var1
public java.lang.String testpro.Test.var2

How many times Class class object will get created if we load same class multiple times?

Class class object will be created only once even if we load same class multiple times or create multiple object of that class.

How to prove that Class class object created only once when class gets loaded?

```
public class Main {
	public static void main(String[] args) throws ClassNotFoundException {
		Test t1 = new Test();
		Class c1 = t1.getClass();
		Test t2 = new Test();
		Class c2 = t2.getClass();
		System.out.println(c1.hashCode() + " .. " + c2.hashCode() + " .. " + (c1 ==
	c2));
	}
}

class Test {
}
```

O/P:
366712642 .. 366712642 .. true

Since c1 and c2 are equal, it is clear that both are pointing to same reference.

What will be the output of below code and why?

```
package testpro;
public class Main {
	public static void main(String[] args) throws ClassNotFoundException {
		Class c = Class.forName("Test");
		System.out.println(c.getName());
	}
}
class Test {
```

```
}
```

O/P:

```
Exception in thread "main" java.lang.ClassNotFoundException: Test
	at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
	at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
	at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
	at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
	at java.lang.Class.forName0(Native Method)
	at java.lang.Class.forName(Class.java:264)
	at testpro.Main.main(Main.java:11)
```

With above output it is clear that when we call Class.forName() method, JVM tries to load that class, but it stores all information about class along with package name so it can not be used alone without package name. To get required about class it has to mentioned along with package name.

Ex:

```
package testpro;
public class Main {
	public static void main(String[] args) throws ClassNotFoundException {
		Class c = Class.forName("testpro.Test");
		System.out.println(c.getName());
	}
}
class Test {
}
```

O/P:

```
testpro.Test
```

How to prove practically that Bootstrap class loader loads classes available in rt.jar, Extension class loader loads classes from ext folder and Application class loader loads classes from application path?

Create sample extension class:

```
package myext;
public class MyExtClass {
	public int var1 = 0;
	private static String var2 = "My Extension";
	public int m1() {
		return 0;
	}
}
```

Create jar of this class and put it under ext folder.

Create below sample code 1 and run:

```
package jvm;
public class Main {
	public static void main (String args[]) throws ClassNotFoundException {
		Class cl = Class.forName("jvm.Test");
		System.out.println(cl.getClassLoader());
		Class c2 = Class.forName("myext.MyExtClass");
		System.out.println(c2.getClassLoader());
		Class c3 = Class.forName("java.lang.String");
		System.out.println(c3.getClassLoader());
		Class c4 = Class.forName("somepkg.someclass");
		System.out.println(c4.getClassLoader());
	}
}
class Test {
}
```

O/P:

```
sun.misc.Launcher$AppClassLoader@73d16e93
sun.misc.Launcher$ExtClassLoader@7852e922
null
Exception in thread "main" java.lang.ClassNotFoundException: somepkg.someclass
   at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
   at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
   at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
   at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
   at java.lang.Class.forName0(Native Method)
   at java.lang.Class.forName(Class.java:264)
   at jvm.Main.main(Main.java:13)
Java Result: 1
```

Create below sample code 2 and run:

```
package jvm;
import myext.MyExtClass;
public class Main {
	public static void main (String args[]) throws ClassNotFoundException {
		Test c1 = new Test();
		System.out.println(c1.getClass().getClassLoader());
		MyExtClass c2 = new MyExtClass();
		System.out.println(c2.getClass().getClassLoader());
		String c3 = new String("ABC");
		System.out.println(c3.getClass().getClassLoader());
	}
```

```
}
class Test {
}
```

With above code it is clear that:

- Test class available at application path is loaded by Application class loader
- MyExtClass class available at ext folder is loaded by Extension class loader
- String class not loaded by Application and Extension class loader which leaves only option as Bootstrap class loader.

Why do we get null value which fetching class loader name if it is loaded by Bootstrap class loader?

Bootstrap class loader implemented in native code and since this is not Java code if does not contain any Java specific class name it returns null.

How many objects will be loaded by below code and why?

```
public class Main {
	public static void main (String args[]) throws ClassNotFoundException {
		String str = new String("123");
		MyClass mc = new MyClass();
	}
}
class MyClass {
}
```

Total 4 class will be loaded, String and MyClass are obvious answers. Main class will be loaded since we are running main of this class only, Object class will also gets loaded since this is Parent for Object.

What is second main component of Class loader subsystem after loading?

After loading, Class loader subsystems performs Linking. Linking again divided into 3 parts:

1. Verification: This is also called “Byte code verifier”. This is responsible for below tasks
    a. Check format of class file
    b. Check if class compiled with valid compiler
    c. Check if class file structured properly.
    d. Check for virus.

    If verification fails it throws VerifyError.

2. Preparation: This is responsible for below tasks
    a. Allocate memory to static variables.
    b. Assign default value for static variables.

3. Resolution: This is responsible for below tasks
    a. Replace all symbolic reference with actual reference from method area.

What do you mean by replacing symbolic reference?

Symbolic reference means references which are used in coding.

Ex:

```
public class A {
        public static void main (String arr[]) {
                System.out.println("In A");
                B b = new B();
        }
}
```

Here A, String and B are considered as symbolic references which are stored in constant pool of A class. These all symbolic references will be replaces with actual memory references present in method area.

In which order does linking execute above mentioned steps?

It based on JVM implementation, in most case in runs in Verify → Prepare → Resolve order but it can implemented to run some part in parallel.

What is the next step performed by class loader after linking and what it does in that phase?

After linking class loader performs “Initialization”. In initialization phase, class loader performs below task:

1. Execute static block if written in that class.
2. Assign actual values to static variables.

Which component makes Java secure?

Verify component of Linking which is part of class loader subsystem checks for byte code generated by compiler, it also scan for virus, someone can try to write virus in Java file but verify component will not allow this to run.

What will happen if any class file fails during loading, linking and initialization phase?

In such case, JVM will throw java.lang.Linkage error.

How many types of memory area are there in JVM and why it is needed?

Total there are 5 types of memory area:

1. Method Area
2. Heap Area
3. Stack
4. PCRegister
5. Native method stack

How many types of memory area are there in JVM and why it is needed?

Total there are 5 types of memory area:

1. Method / Class Area
2. Heap Area
3. Stack
4. PC Register
5. Native method stack

Need:
It is needed to store class level metadata, Objects, instance variables, static data, next instruction to be executed etc. These memory areas are there to perform operation for execution engine, developers can also use these memory areas to get information like class level information, memory utilization etc..

Explain Method area in detail?

Method area contains all class level data including static variables and constant pool for that class. Whenever we run and class file class loader loads binary data of that class file in method area.

Some important points for method area:

- As soon as JVM starts, it creates method area to store class file information.
- This memory area is per JVM, so all threads share same data.
- Since multiple threads can share data, method area is not thread safe.

As a developer what information we can get from method area and how?

We can get complete metadata information (Ex: method information, field information, parent class, etc ..) about any class with the help of Class class object (Reflection API). Examples mentioned above.

Explain Heap area in detail?

Heap area contains Objects and instance variable data. Whenever we create object of any class it's complete data along with instance variable will get stored in heap.

Some important points for heap area:

- Like method area, JVM also creates heap area during start up.
- This memory area is also per JVM, so all threads share same data.
- Since multiple threads can share data, method area is not thread safe.
- Since Arrays are considered as object only, this also get stored in heap.

As a developer what information we can get from heap area and how?

We can get information from heap with the help of object of Runtime class which is singleton.

Runtime class provides some methods to get heap memory statistics.
freeMemory(): To get free memory inside heap.
maxMemory(): To get max memory allotted to heap.
totalMemory(): To get total memory (Initial memory) allotted to heap.

Ex:

```
public class Main {
	public static void main (String arr[]) throws IOException {
		Runtime run = Runtime.getRuntime();
		System.out.println("Free Memory: " + run.freeMemory());
		System.out.println("Max Memory: " + run.maxMemory());
		System.out.println("Total Memory: " + run.totalMemory());
	}
}
```

O/P:

Free Memory: 1000760696
Max Memory: 15004598272
Total Memory: 1011351552

All digits are in bytes.

What extra functionality does Runtime class provides?

Runtime class provides below additional methods:
gc(): To run garbage collector, this does not give guarantee that GC will release memory as soon as we run this method.
availableProcessors(): To get available processors.
exec(): To execute any command on system.
addShutdownHook(): To register any thread as shutdown hook which gets executed just before terminating program.

How to set max and min heap size?

For max: java -Xmx512M Test
For min: Java -Xms64M Test

Here Test is a class, it can also be a jar file.

Explain Stack area in detail?

Stack area contains method call information along with local variable by thread. It also stores reference variable which points to objects and instance variable created on heap.

Some important points for stack area:

- Stack area is created for each and every thread.
- All threads access their own stack hence this memory is thread safe.
- Each and every method call performed by that thread will be stored in the stack including local variables.
- As soon as method call completed, related stack will get deleted from stack.
- After completing all method calls and just before terminating thread, stack will get empty, which gets destroyed by JVM.
- Each entry in stack is called Stack Frame or Activation Record.

Explain Stack Frame in details:

Each Stack Frame is divided into 3 parts:

1. Local variable array

2. Operand stack
3. Frame Data

Add diagram *******

Explain each stack frame component in detail.

Local Variable Array:

- Local variable divided into multiple slots to store local variables used within method.
- Each slot holds 4 byte information.
- All int, float and object reference occupy 1 slot.
- All double and long occupy 2 slots.
- All byte (1 byte), short (2 byte) and char (2 byte) occupy 1 slot.
- All boolean values can take 1 or 2 slots depends on JVM, in most cases it occupies 1 slot.

Add diagram along with some code values *******

Operand Stack:

- JVM uses operand stack as workspace to perform some operation.
- There are some predefined instruction which can be used to push value to operand stack and pop value from it.

Ex:
```
int a = 10;
int b = 20;
int c = a + b;
```

Here with first two lines local variable array will get 10 and 20 as first and second index. Addition will happen in operand stack and sum value will return back to store as third index for local variable array.

Frame Data:

- Frame data created for each and every thread.
- Frame data contains metadata and some specific information related to that method. (Ex: Constant pool, symbolic references, exception table which provides catch block information in case of exception.)

Explain PC Register in detail?

PC stands for program counter, this memory area gets created as soon as we run any java application which stores current and next execution instruction for any thread.

Some important points for PC Register:

- This memory area created per thread.
- Each thread can refer their own PC register only, hence this memory area is thread safe.
- This is purely used by JVM only and not related to developer.
- Each entry in PC register holds current execution instruction.
- Once current execution completed, program counter incremented to hold next execution instruction.

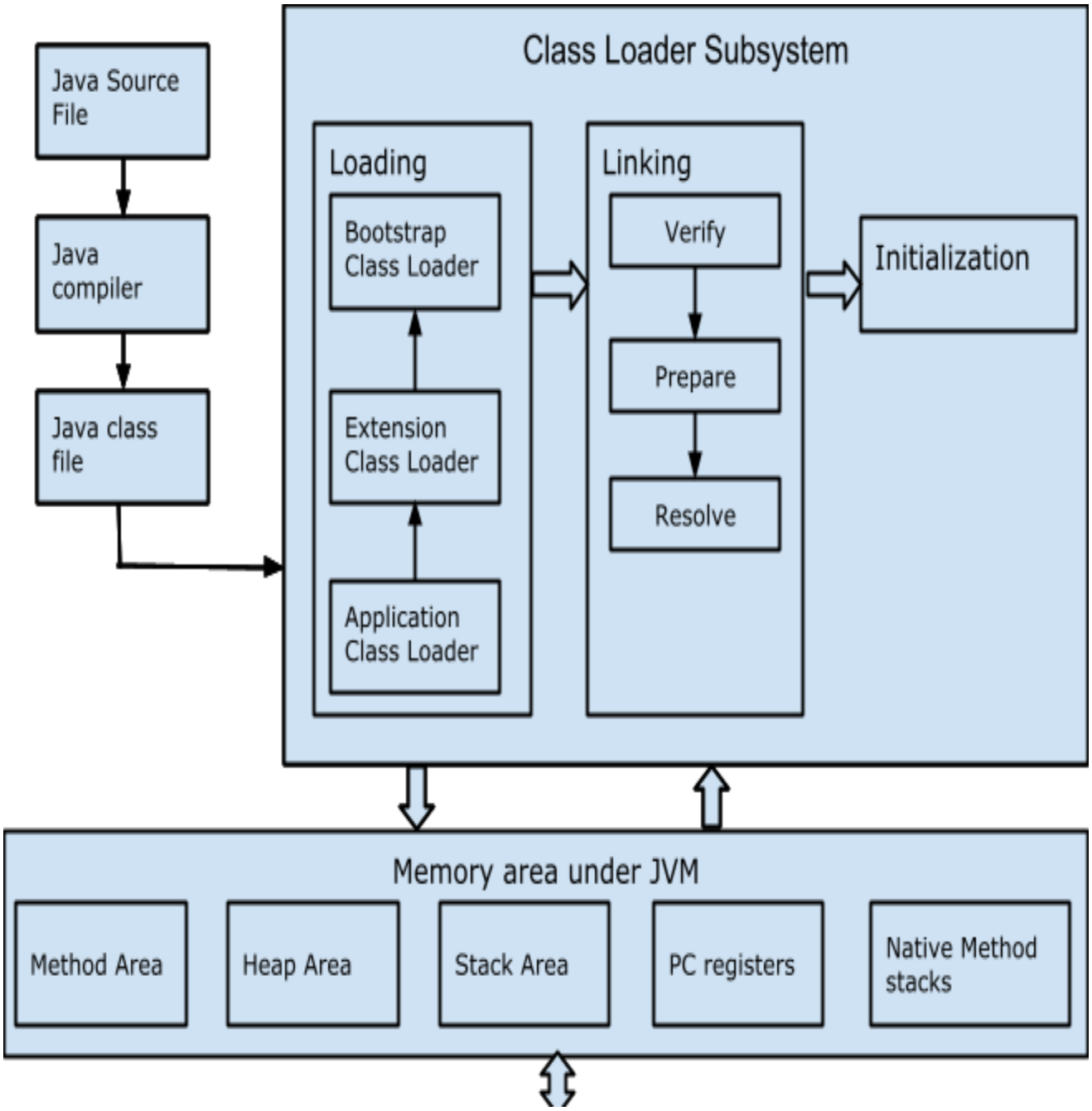
Explain Native method stack in detail?

This memory area gets created as soon as we run any java application which stores native method call for any thread.

Some important points for Native method stack:

- This memory area created per thread.
- Each thread can refer their own Native method stack only, hence this memory area is thread safe.
- This is purely used by JVM only and not related to developer.
- All native method calls invoked by thread will be stored in this area.

JVM Architecture:


Java Source File
Java compiler
Java class file
Class Loader Subsystem
Loading
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Linking
Verify
Prepare
Resolve
Initialization
Memory area under JVM
Method Area
Heap Area
Stack Area
PC registers
Native Method stacks