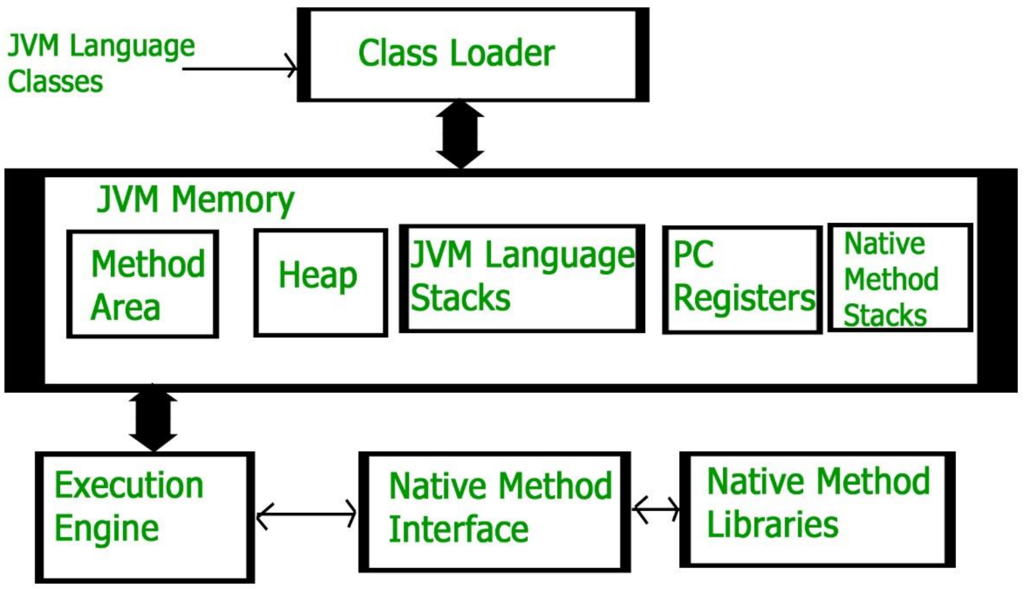
How JVM Works – JVM Architecture?

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.

```
// A Java program to demonstrate working of a Class type
// object created by JVM to represent .class file in
// memory.
import java.lang.reflect.Field;
import java.lang.reflect.Method;

// Java code to demonstrate use of Class object
// created by JVM
public class Test
{
  public static void main(String[] args)
  {
    Student s1 = new Student();

    // Getting hold of Class object created
    // by JVM.
    Class c1 = s1.getClass();

    // Printing type of object using c1.
    System.out.println(c1.getName());

    // getting all methods in an array
    Method m[] = c1.getDeclaredMethods();
    for (Method method : m)
      System.out.println(method.getName());

    // getting all fields in an array
    Field f[] = c1.getDeclaredFields();
    for (Field field : f)
      System.out.println(field.getName());
  }
}

// A sample class whose information is fetched above using
// its Class object.
class Student
{
  private String name;
  private int roll_No;

  public String getName()  {  return name;   }
  public void setName(String name) { this.name = name; }
  public int getRoll_no()  { return roll_No;  }
  public void setRoll_no(int roll_no) {
    this.roll_No = roll_no;
  }
}
```
Output:

```
Student

getName

setName

getRoll_no
```

```
setRoll_no

name

roll_No
```

Note: For every loaded *.class* file, only **one** object of Class is created.

```
Student s2 = new Student ();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass ();

System.out.println (c1==c2); // true
```

Linking: Performs verification, preparation, and (optionally) resolution.

- *Verification*: It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
- *Preparation*: JVM allocates memory for class variables and initializing the memory to default values.
- *Resolution*: It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

Initialization: In this phase, all static variables are assigned with their values defined in the code and static block (if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.

In general, there are three class loaders:

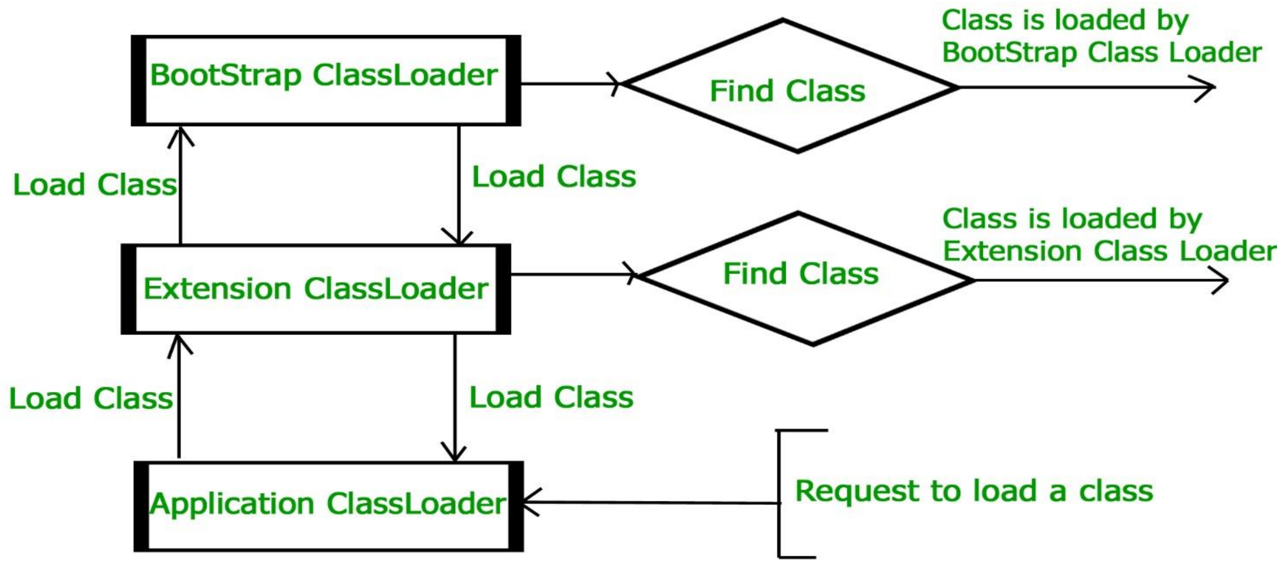
- *Bootstrap class loader*: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in *JAVA_HOME/jre/lib* directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
- *Extension class loader*: It is child of bootstrap class loader. It loads the classes present in the extensions directories *JAVA_HOME/jre/lib/ext* (Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
- *System/Application class loader*: It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader* class.

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

      // Test class is loaded by Application loader
      System.out.println(Test.class.getClassLoader());
   }
}
```

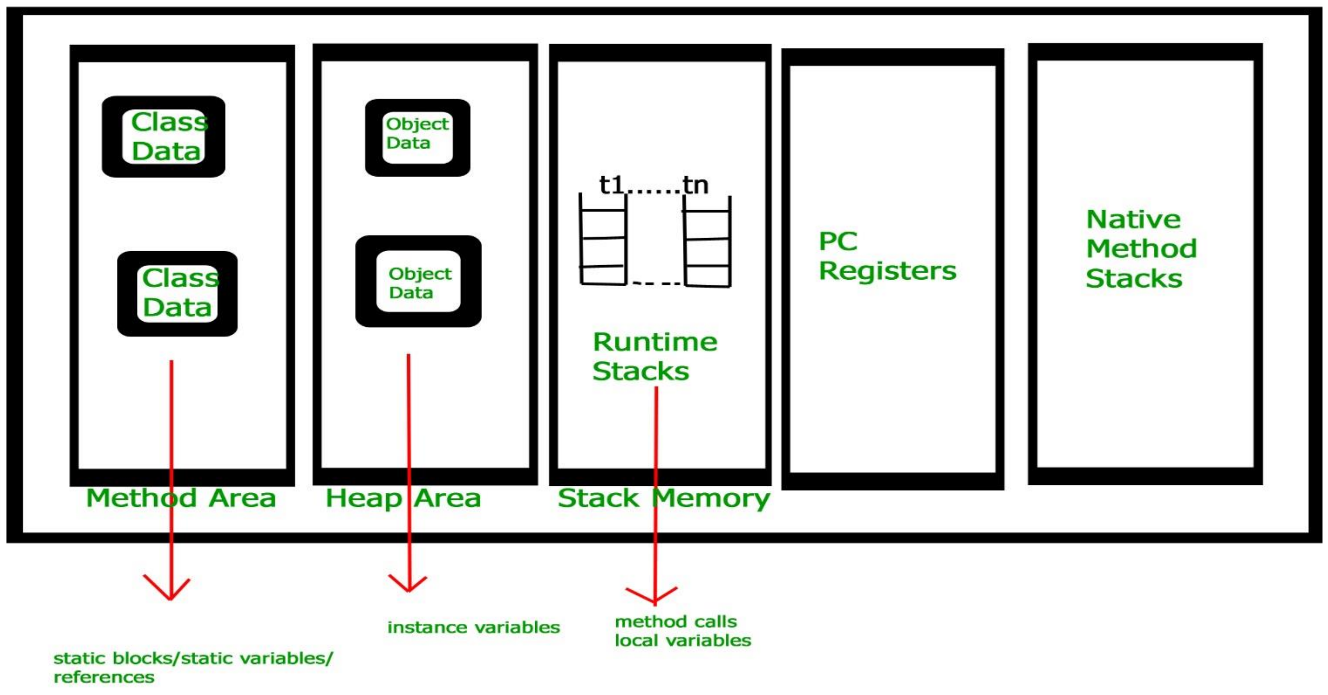
Output:

```
null

sun.misc.Launcher$AppClassLoader@73d16e93
```

Note: JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.



JVM Memory
Method area : In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.
Heap area: Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.
Stack area: For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.
PC Registers: Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.
Native method stacks: For every thread, separate native stack is created. It stores native method information.



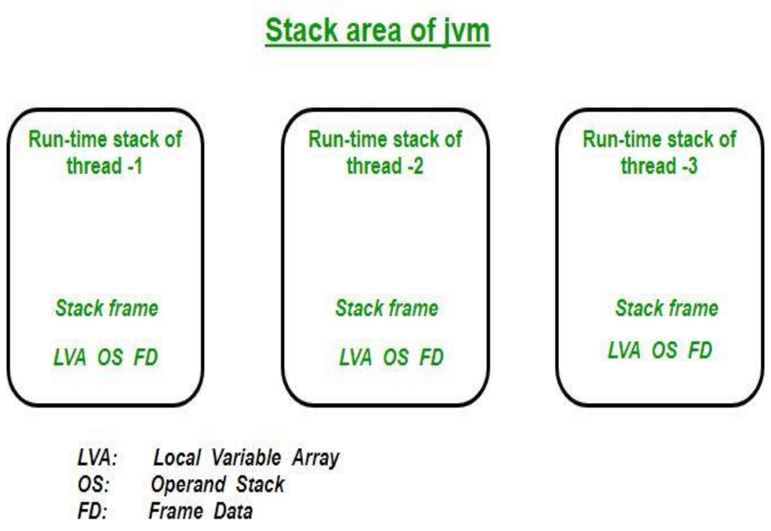
Execution Engine
Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

- *Interpreter*: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- *Just-In-Time Compiler(JIT)* : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.
- *Garbage Collector*: It destroy un-referenced objects.For more on Garbage Collector,refer Garbage Collector.

Java Native Interface (JNI):
It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.
Native Method Libraries:
It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

Java Virtual Machine (JVM) Stack Area

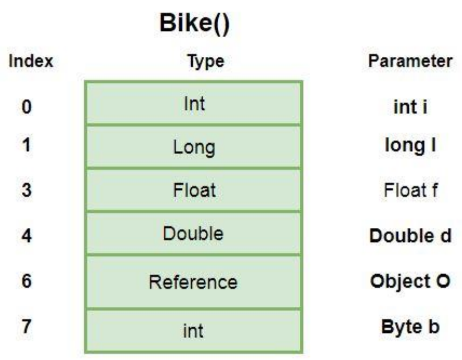
For every thread, JVM creates a separate stack at the time of thread creation. The memory for a Java Virtual Machine stack does not need to be contiguous. The Java virtual machine only performs two operations directly on Java Stacks: it pushes and pops frames. And stack for a particular thread may be termed as **Run – Time Stack**. Each and every method call performed by that thread is stored in the corresponding run time stack including parameters, local variables, intermediate computations, and other data. After completing a method, corresponding entry from the stack is removed. After completing all method calls the stack becomes empty and that empty stack is destroyed by the JVM just before terminating the thread. The data stored in the stack is available for the corresponding thread and not available to the remaining threads. Hence we can say local data is thread safe. Each entry in the stack is called **Stack Frame** or **Activation Record**.



Stack Frame Structure
The stack frame basically consists of **three** parts: **Local Variable Array, Operand Stack & Frame Data**. When JVM invokes a Java method, first it checks the class data to determine the number of words *(size of the local variable array and operand stack, which are measured in words for each individual method)* required by the method in the local variables array and operand stack. It creates a stack frame of the proper size for invoked method and pushes it onto the Java stack.
1. Local Variable Array (LVA):

- The local variables part of stack frame is organized as a zero-based array of words.
- It contains all parameters and local variables of the method.
- Each slot or entry in the array is of 4 Bytes.
- Values of type int, float, and reference occupy 1 entry or slot in the array i.e. 4 bytes.
- Values of double and long occupy 2 consecutive entries in the array i.e. 8 bytes total.
- **Byte, short and char values will be converted to int type before storing** and occupy 1 slot i.e. 4 Bytes.
- But the way of storing Boolean values is varied from JVM to JVM. But most of the JVM gives 1 slot for Boolean values in the local variable array.
- The parameters are placed into the local variable array first, in the order in which they are declared.
- **For Example:** Let us consider a class Example having a method **bike()** then local variable array will be as shown in below diagram:

```
// Class Declaration

class Example

{

  public void bike(int i, long l, float f,

             double d, Object o, byte b)

  {

    return 0;

  }

}
```

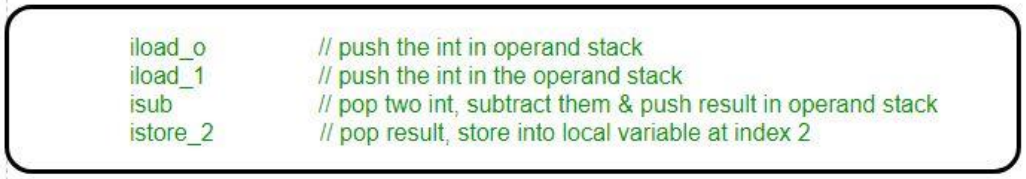


Bike()

Index	Type	Parameter
0	Int	int i
1	Long	long l
3	Float	Float f
4	Double	Double d
6	Reference	Object O
7	int	Byte b

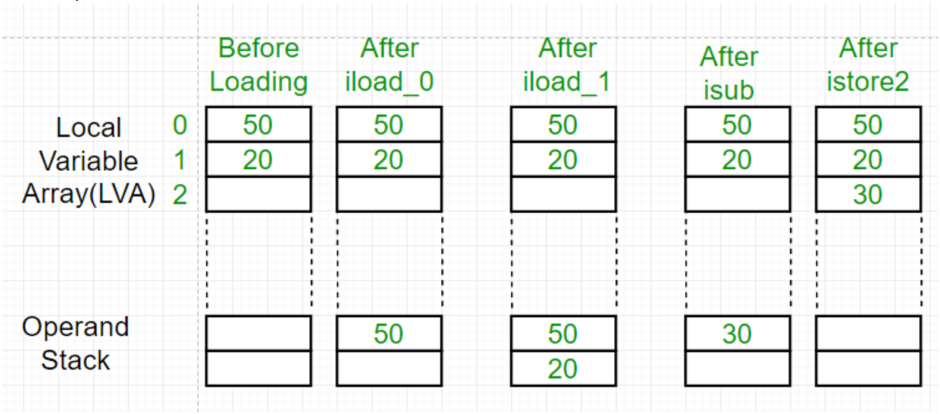
2. Operand Stack (OS):

- JVM uses operand stack as work space like rough work or we can say for storing intermediate calculation's result.
- Operand stack is organized as array of words like local variable array. But this is not accessed by using index like local variable array rather it is accessed by some instructions that can push the value to the operand stack and some instructions that can pop values from operand stack and some instructions that can perform required operations.
- **For Example:** Here is how a JVM will use this below code that would subtract two local variables that contain two ints and store the int result in a third local variable:



```
iload_o        // push the int in operand stack
iload_1        // push the int in the operand stack
isub           // pop two int, subtract them & push result in operand stack
istore_2       // pop result, store into local variable at index 2
```

- So here first two instructions *iload_0* and *iload_1* will push the values in operand stack from local variable array. And instruction *isub* will subtract these two values and stores the result back to the operand stack and after *istore_2*the result will pop out from the operand stack and will store into local variable array at position 2.



3. Frame Data (FD):

- It contains all symbolic reference (*constant pool resolution)* and normal method return related to that particular method.
- It also contains a reference to Exception table which provide the corresponding catch block information in the case of exceptions.

JVM Shutdown Hook in Java

Shutdown Hooks are a special construct that allow developers to plug in a piece of code to be executed when the JVM is shutting down. This comes in handy in cases where we need to do special clean up operations in case the VM is shutting down.

Handling this using the general constructs such as making sure that we call a special procedure before the application exists (calling System.exit(0)) will not work for situations where the VM is shutting down due to an external reason (ex. kill request from O/S), or due to a resource problem (out of memory). As we will see soon, shutdown hooks solve this problem easily, by allowing us to provide an arbitrary code block, which will be called by the JVM when it is shutting down.

From the surface, using a shutdown hook is downright straight forward. All we have to do is simply write a class which extends the java.lang.Thread class, and provide the logic that we want to perform when the VM is shutting down, inside the public void run() method. Then we register an instance of this class as a shutdown hook to the VM by calling Runtime.getRuntime().addShutdownHook(Thread) method. If you need to remove a previously registered shutdown hook, the Runtime class provides the removeShutdownHook(Thread) method as well.

For Example :

filter_none

edit

play_arrow

brightness_4

```
public class ShutDownHook

{

  public static void main(String[] args)

  {



    Runtime.getRuntime().addShutdownHook(new Thread()

    {

      public void run()

      {

        System.out.println("Shutdown Hook is running !");

      }

    });
```

```
        System.out.println("Application Terminating ...");

    }

}
```

When we run the above code, you will see that the shutdown hook is getting called by the JVM when it finishes execution of the main method.

Output:

```
Application Terminating ...

Shutdown Hook is running !
```

Simple right? Yes it is.

While it is pretty simple to write a shutdown hook, one needs to know the internals behind the shutdown hooks to make use of those properly. Therefore, in this article, we will be exploring some of the 'gotchas' behind the shutdown hook design.

1. Shutdown Hooks may not be executed in some cases!
First thing to keep in mind is that it is not guaranteed that shutdown hooks will always run. If the JVM crashes due to some internal error, then it might crash down without having a chance to execute a single instruction. Also, if the O/S gives a SIGKILL (http://en.wikipedia.org/wiki/SIGKILL) signal (kill -9 in Unix/Linux) or TerminateProcess (Windows), then the application is required to terminate immediately without doing even waiting for any cleanup activities. In addition to the above, it is also possible to terminate the JVM without allowing the shutdown hooks to run by calling Runime.halt() method. Shutdown hooks are called when the application terminates normally (when all threads finish, or when System.exit(0) is called). Also, when the JVM is shutting down due to external causes such as user requesting a termination (Ctrl+C), a SIGTERM being issued by O/S (normal kill command, without -9), or when the operating system is shutting down.

2. Once started, Shutdown Hooks can be forcibly stopped before completion.
This is actually a special case of the case explained before. Although the hook starts execution, it is possible to be terminated before it completes, in cases such as operating system shutdowns. In this type of cases, the O/S waits for a process to terminate for a specified amount of time once the SIGTERM is given. If the process does not terminate within this time limit, then the O/S terminates the process forcibly by issuing a SIGTERM (or the counterparts in Windows). So it is possible that this happens when the shutdown hook is half-way through its execution.
Therefore, it is advised to make sure that the Shutdown Hooks are written cautiously, ensuring that they finish quickly, and do not cause situations such as deadlocks. Also, the JavaDoc [1] specifically mentions that one should not perform long calculations or wait for User I/O operations in a shutdown hook.

3. We can have more than one Shutdown Hooks, but their execution order is not guaranteed.
As you might have correctly guessed by the method name of addShutdownHook method (instead of setShutdownHook), you can register more than one shutdown hook. But the execution order of these multiple hooks is not guaranteed by the JVM. The JVM can execute shutdown hooks in any arbitrary order. Moreover, the JVM might execute all these hooks concurrently.
4. We cannot register / unregister Shutdown Hooks with in Shutdown Hooks
Once the shutdown sequence is initiated by the JVM, it is not allowed to add more or remove any existing shutdown hooks. If this is attempted, the JVM throws IllegalStateException.
5. Once shutdown sequence starts, it can be stopped by Runtime.halt() only.
Once the shutdown sequence starts, only Runtime.halt() (which forcefully terminates the JVM) can stop the execution of the shutdown sequence (except for external influences such as SIGKILL). This means that calling System.exit() with in a Shutdown Hook will not work. Actually, if you call System.exit() with in a Shutdown Hook, the VM may get stuck, and we may have to terminate the process forcefully.
6. Using shutdown hooks require security permissions.
If we are using Java Security Managers, then the code which performs adding / removing of shutdown hooks need to have the shutdownHooks permission at runtime. If we invoke this method without the permission in a secure environment, then it will result in SecurityException.