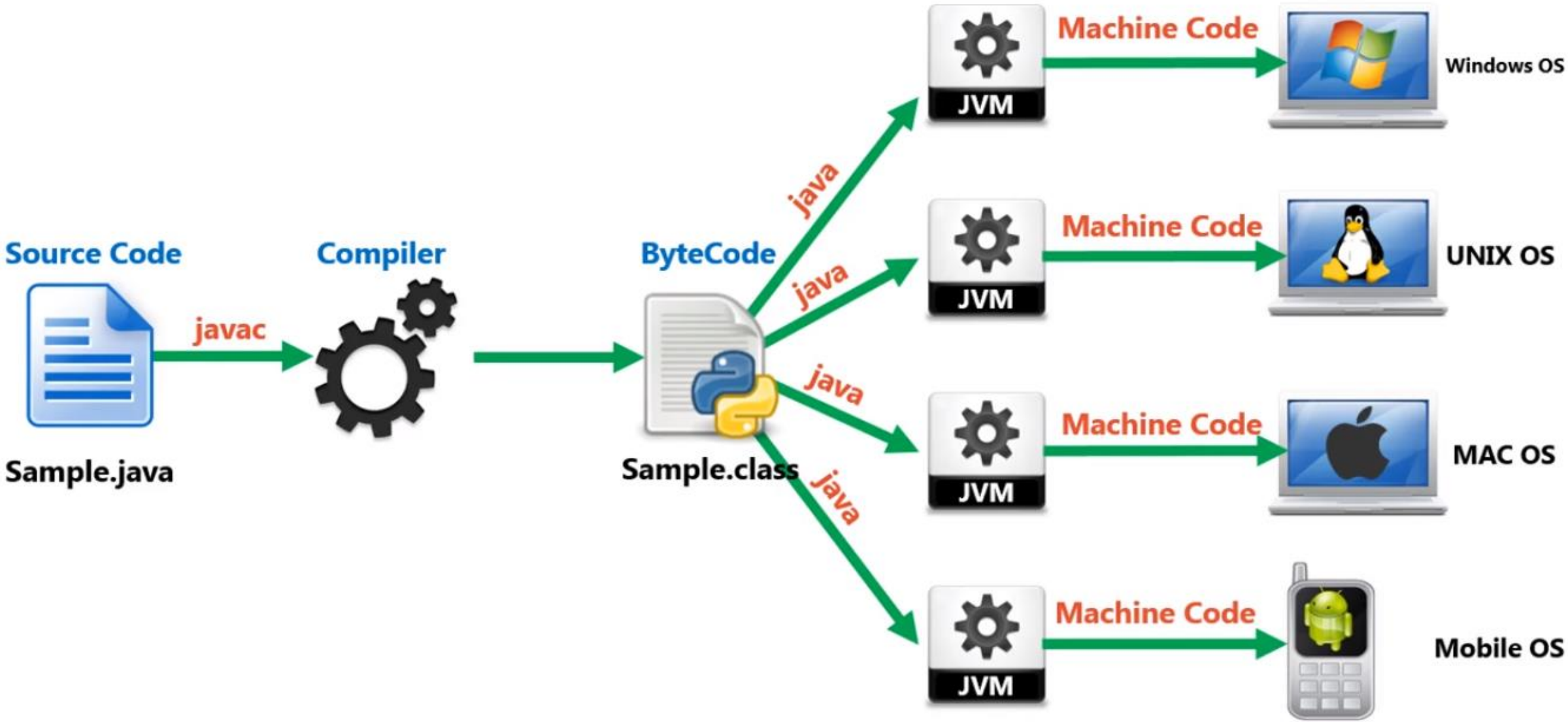
Java™

ORACLE


GALILEO
GALILEI


Source Code
Compiler
ByteCode
javac
Sample.java
Sample.class
java
java
java
java
JVM
JVM
JVM
JVM
Machine Code
Machine Code
Machine Code
Machine Code
Windows OS
UNIX OS
MAC OS
Mobile OS

JVM Architecture

content

- What is virtual machine

- Types of virtual machines
    1). Hardware based VM
    2). Application based VM

- Basic Architecture of JVM

- Class loader subsystem
    1). Loading
    2). Linking
    3). Initialization

- Types of class loaders
    1). Bootstrap class loader
    2). Extension class loader
    3). Application class loader

- How class loader works
- What is the need of customized class loader
- Pseudo code for customized class loader
- Various memory area of JVM
    1). Method Area
    2). Heap Area
    3). Stack
    4). Program Count Register
    5). Native Method Stack

- Program to display heap memory statistics
- How to set maximum and minimum heap size
- Execution Engine
    1). Interpreter
    2). JIT compiler
- JNI (Java Native Interface)
- Complete architecture diagram of JVM
- Class file structure

➢Virtual Machine

it is a software simulation of a machine which can perform operations like a physical machine

- There are two types of virtual machine

1) Hardware based or system based virtual machine
2) Application based or processed based virtual machine

- Hardware or System based virtual machine

- It provides several logical systems on the same computer with strong isolation from each other

- That is on one physical machine we are defining multiple logical machines

- The main advantage of hardware based virtual machine is hardware resource sharing and improved utilization of hardware resources

- Examples for hardware based virtual machines

- KVM – kernel based virtual machines for Linux systems
- VMWare
- Xen
- Cloud Computing

- Application or process based virtual machines

- These virtual machines act as runtime engines to run a particular programming language application

- JVM act as runtime engine to run java based applications

➢PVM– Parrot Virtual Machine

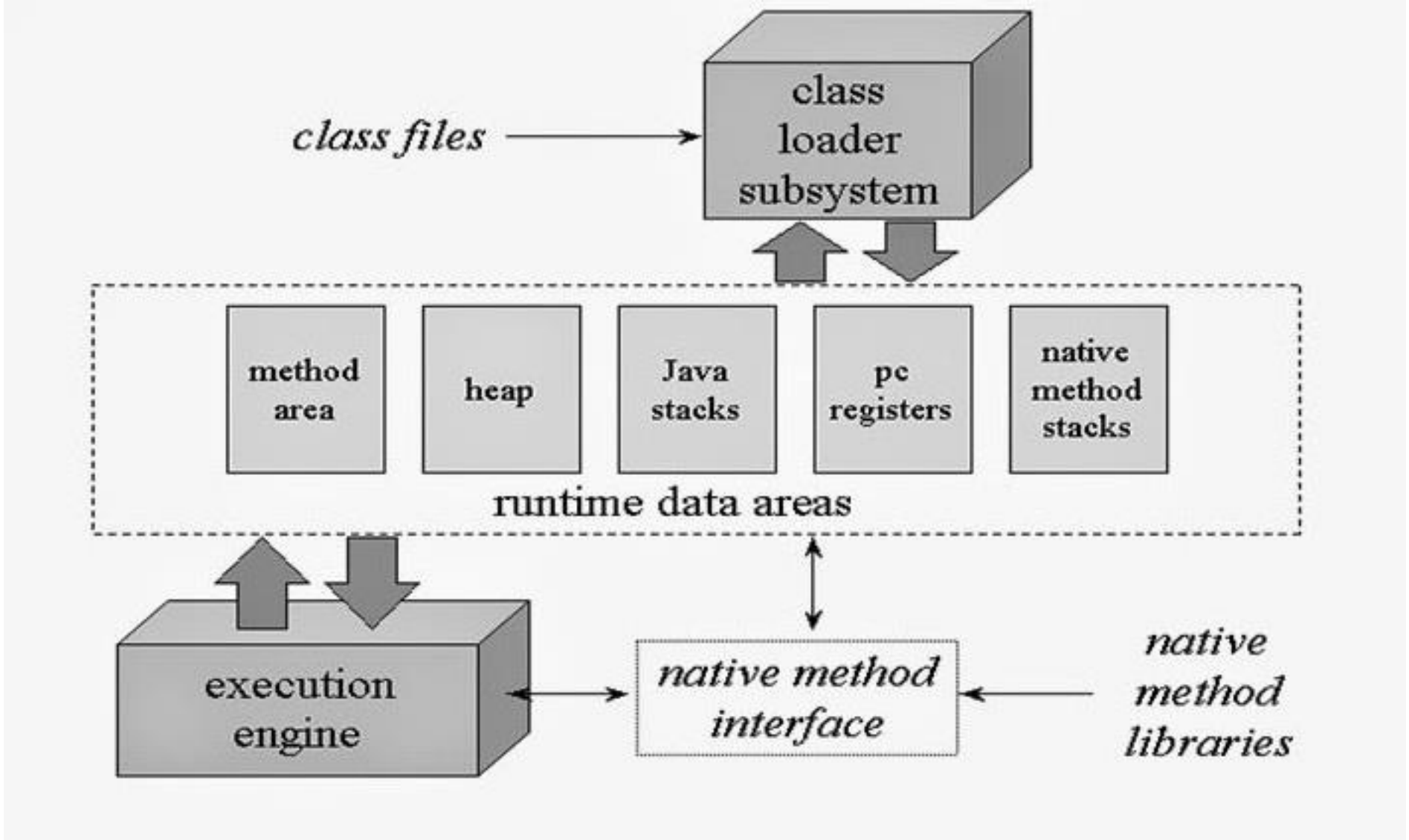
PVM acts as runtime engine to run Perl based applications

➢CLR– Common Language Runtime

CLR acts as runtime engine to run .NET based applications

JVM

➢Basic architecture diagram of JVM


class files
class loader subsystem
method area
heap
Java stacks
pc registers
native method stacks
runtime data areas
execution engine
native method interface
native method libraries

➢Class loader subsystem

- Class loader subsystem is responsible to read and load .class files into memory area

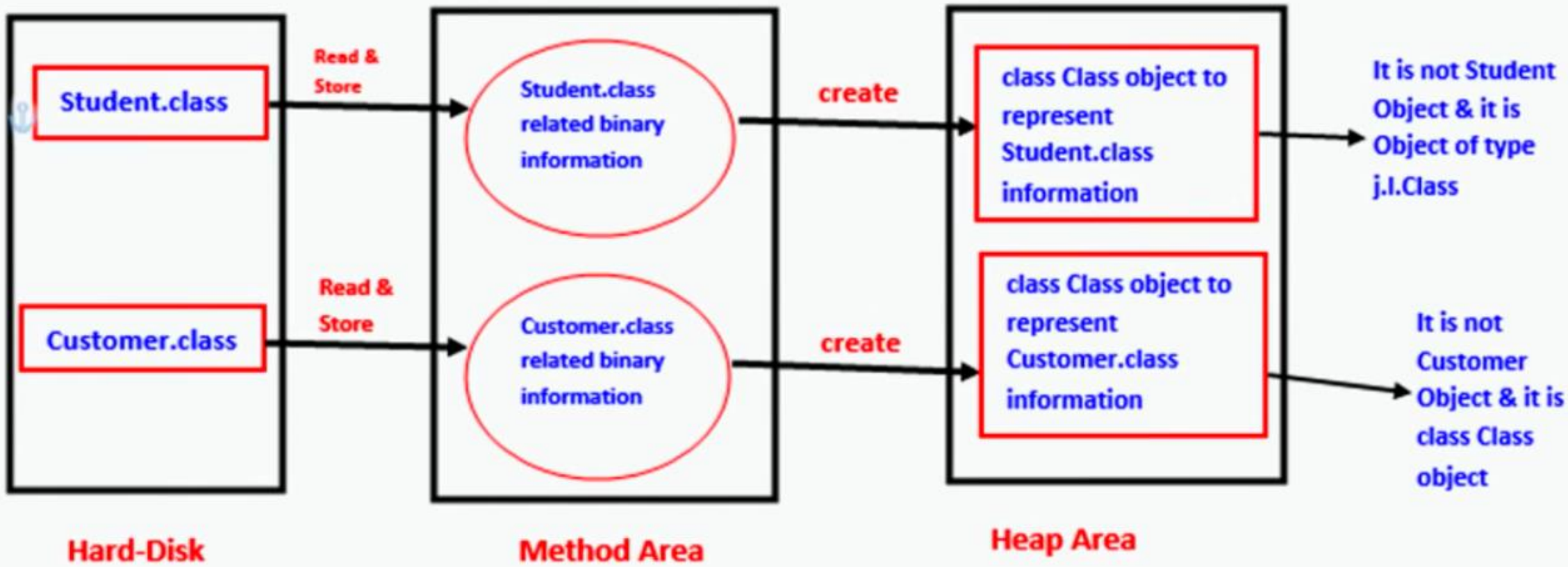
- Class loader subsystem is responsible for the following three activities
    1) Loading
    2) Linking
    3) Initialization

- Loading

- Loading means reading class file and store corresponding binary data in Method Area

- For each class file JVM will store below information in the Method Area

- Fully qualified class name
- Fully qualified class name of the immediate parent
- Method information
- Variable information
- Constructor information
- Modifier information
- Constant pool information etc…



The Class Object can be used by Programmer to get Class Level Information Like Fully Qualified Name of the Class, Parent Name, Methods and Variables Information Etc.

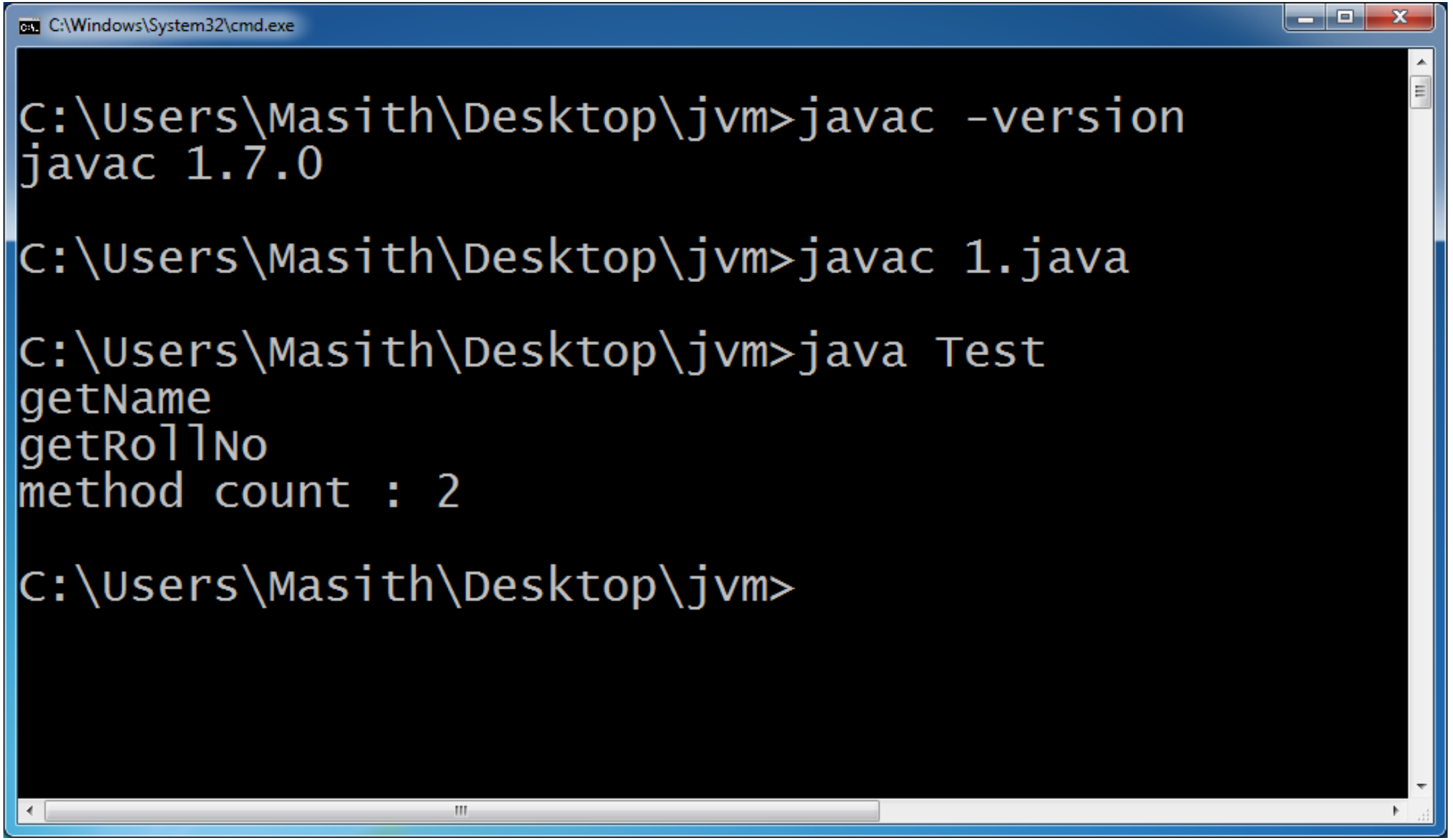
- After loading .class file immediately JVM creates and object for that loaded class on the heap memory of the type java.lang.Class

- The class Class object can be used by programmer to get class level information like method information variable information constructor information etc…

- For every class one class Class object is created

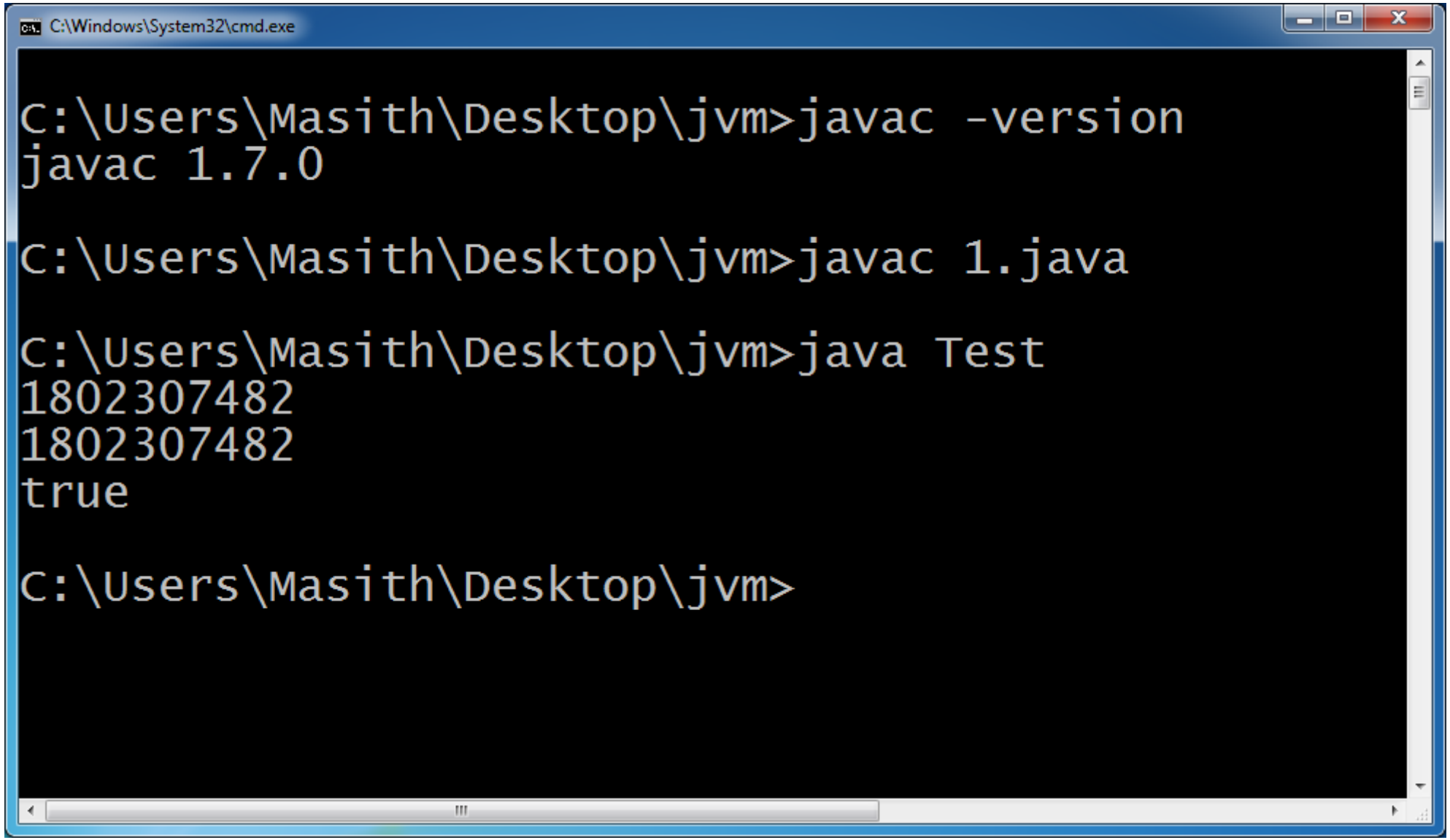
```
import java.lang.reflect.Method;

class Student {
	public String getName(){
		return null;
	}
	public int getRollNo(){
		return 10;
	}
}
```

```
class Test {
    public static void main(String[] args)throws Exception{
        int count = 0;
        Class c = Class.forName("Student");
        Method[] methods = c.getDeclaredMethods();
        for(Method m : methods){
            System.out.println(m.getName());
            count++;
        }
        System.out.println("method count : "+count);
    }
}
```


C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>javac -version
javac 1.7.0
C:\Users\Masith\Desktop\jvm>javac 1.java
C:\Users\Masith\Desktop\jvm>java Test
getName
getRollNo
method count : 2
C:\Users\Masith\Desktop\jvm>

```
class Student {
    public String getName(){
        return null;
    }
    public int getRollNo(){
        return 10;
    }
}
```

```
class Test {
    public static void main(String[] args)throws Exception{
        Student s1 = new Student();
        Class c1 = s1.getClass();
        Student s2 = new Student();
        Class c2 = s2.getClass();
        System.out.println(c1.hashCode());
        System.out.println(c2.hashCode());
        System.out.println(c1 == c2);
    }
}
```


C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>javac -version
javac 1.7.0
C:\Users\Masith\Desktop\jvm>javac 1.java
C:\Users\Masith\Desktop\jvm>java Test
1802307482
1802307482
true
C:\Users\Masith\Desktop\jvm>

- For every loaded class only one class Class object is created even though we are using the class multiple times in our program

Linking

- Linking consist of 3 activities
    1) Verification
    2) Preparation
    3) Resolution

➢Verification

- It is the process of ensuring that binary representation of a class is structurally correct or not

- That is JVM will check whether .class file is generated by valid compiler or not

- That is whether .class file is properly formatted or not

- Internally byte code verifier is responsible for this activity

- Byte code verifier is the part of class loader subsystem

- If verification fails then we will get Runtime exception saying java.lang.VerifyError

- Assume the .class is not generated by the compiler and created by the human to spread some virus

- Then immediately Byte code verifier will identify and it will raise java.lang.VerifyError

➢Preparation

- In this phase JVM will allocate memory for class level static variables and assigned default values

- In initialization phase original values will be assigned to the static variables

- And here only default values will be assigned

➢Resolutions

- It is the process of replacing symbolic names in out program with original memory references from method area

```
class Test {
    public static void main(String[] args){
        String string = new String("java");
        Student student = new Student();
    }
}
```

- For the above class Class Loader loads Test.class String.class Student.class and Object.class

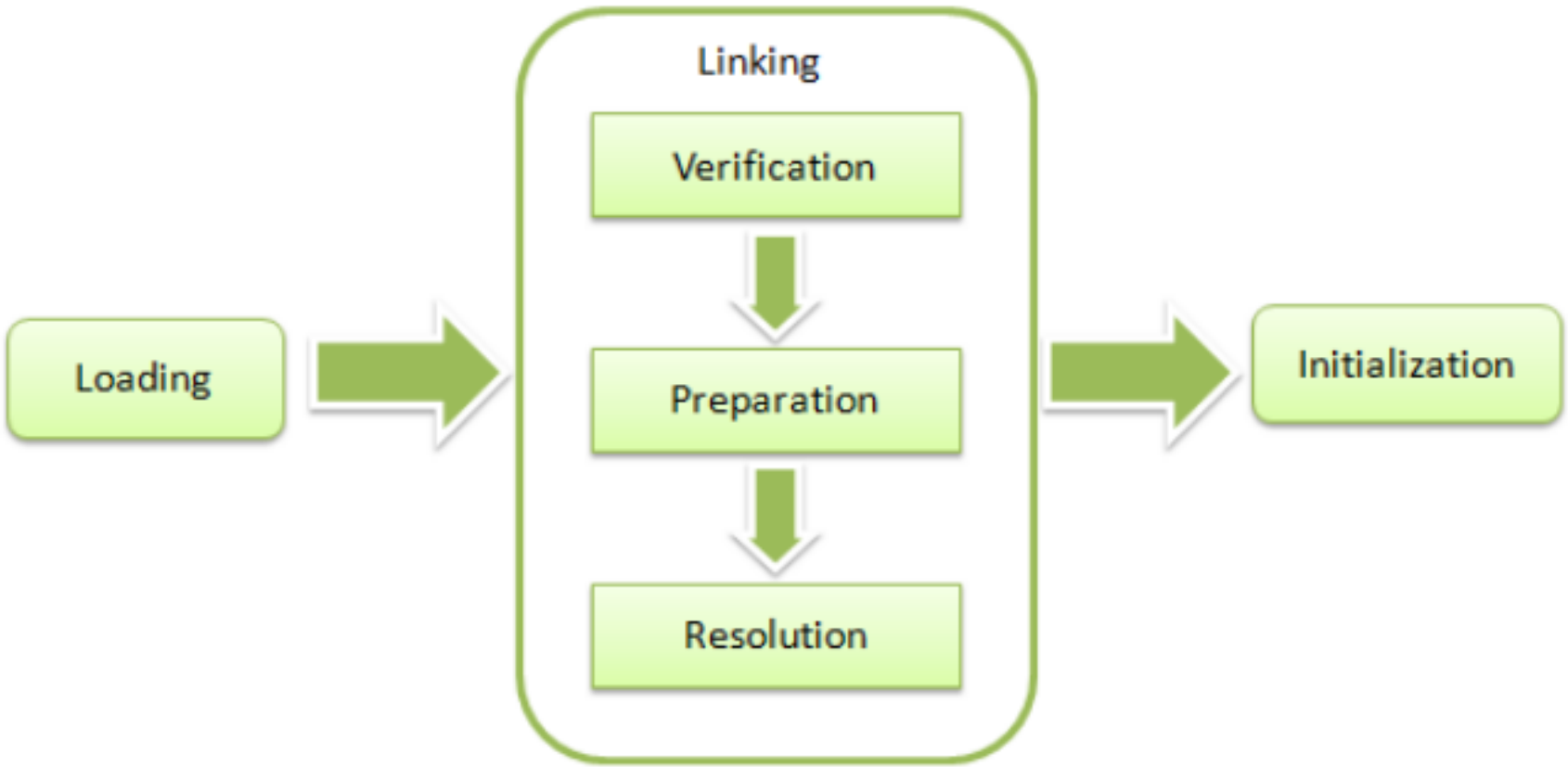
- The names of these classes are stored in constant pool of Test.class

- In resolution phase these names are replaced with original memory level references from method area

➢Initialization

- In initialization phase all static variables are assigned with original values and static blocks will be executed from parent to child and from top to bottom

Loading of java class



- while Loading Linking and Initialization if any error occurs the we will get runtime exception saying java.lang.LinkageError

- VerifyError is child class of LinkageError

- Types of class loaders

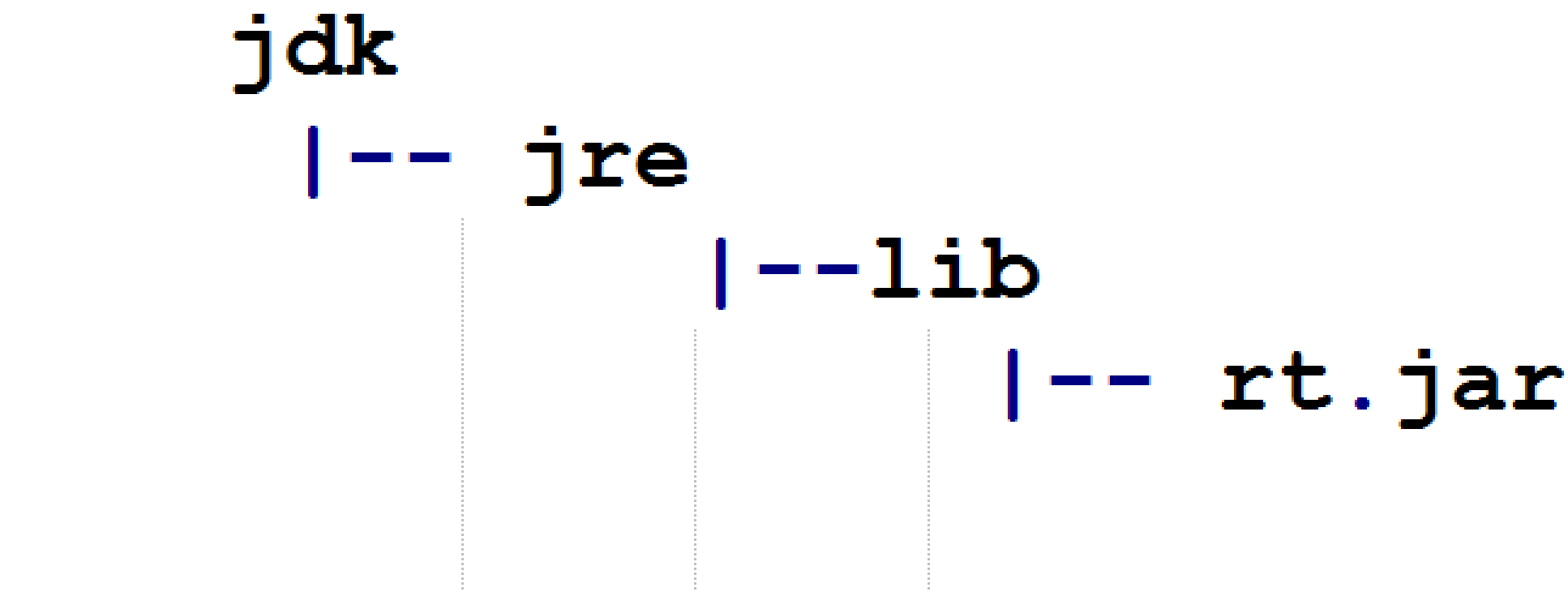
- Class loader subsystem contains following 3 types of class loaders

1) Bootstrap class loader (primordial class loader)
2) Extension class loader
3) Application class loader (system class loader)

- Bootstrap class loader

- Bootstrap class loader is responsible to load the core java API classes

- That is classes present in rt.jar


jdk
|-- jre
|--lib
|-- rt.jar

- This location is called bootstrap class path

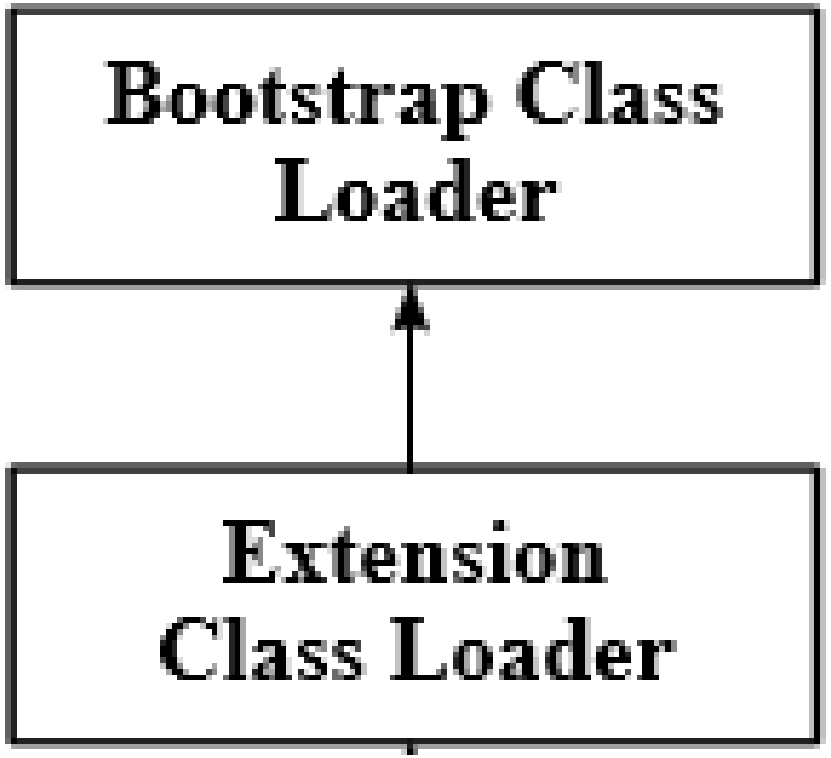
- That is bootstrap class loader is responsible to load the classes from bootstrap class path

- Bootstrap class loader is by default available with every JVM

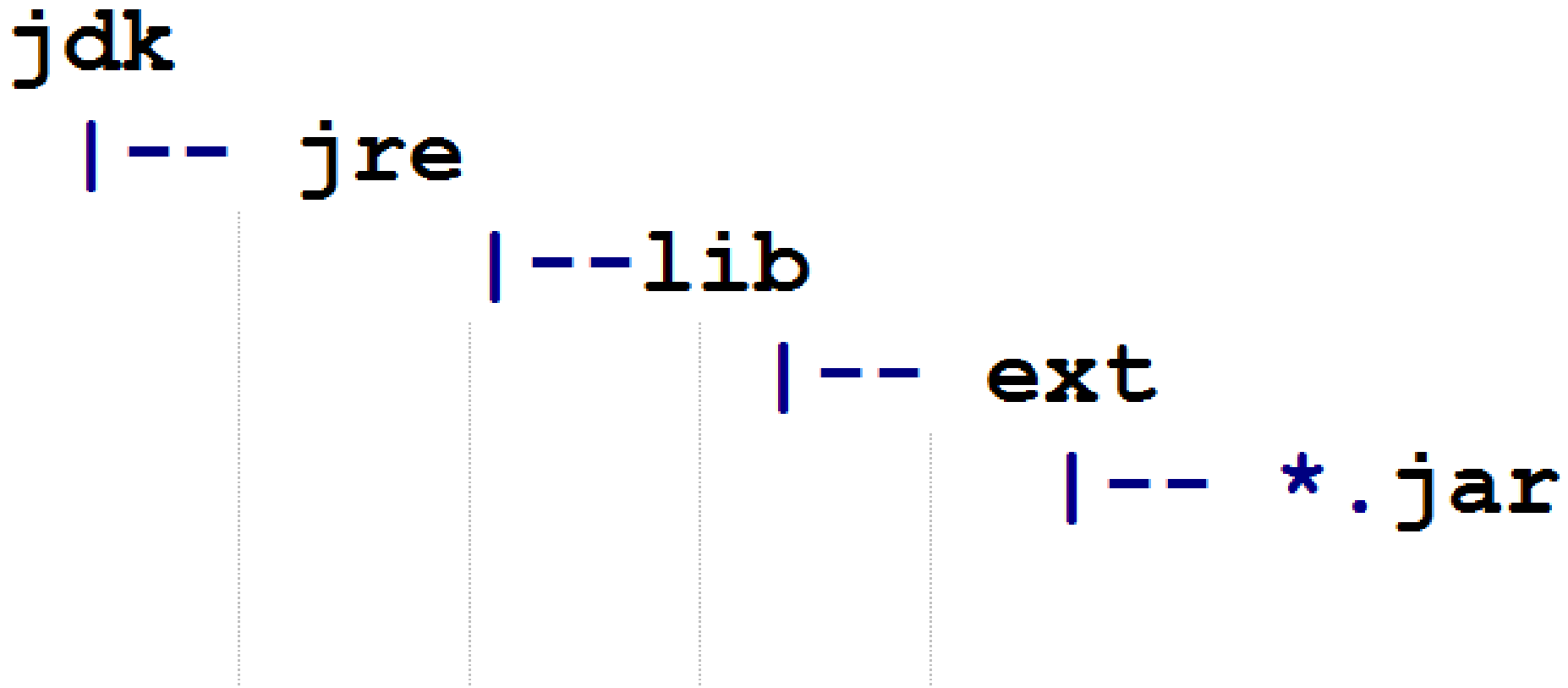
- It is implemented in native languages like C or C++ and not implemented in Java

- Extension class loader

- Extension class loader is the child class of bootstrap class loader


Bootstrap Class Loader
Extension Class Loader

- Extension class loader is responsible to load classes from extension class path


jdk
|-- jre
|--lib
|-- ext
|-- *.jar

- Extension class loader is implemented in java and the corresponding .class file is

- sun.misc.Launcher$ExtClassLoader.class

- Application class loader or System class loader

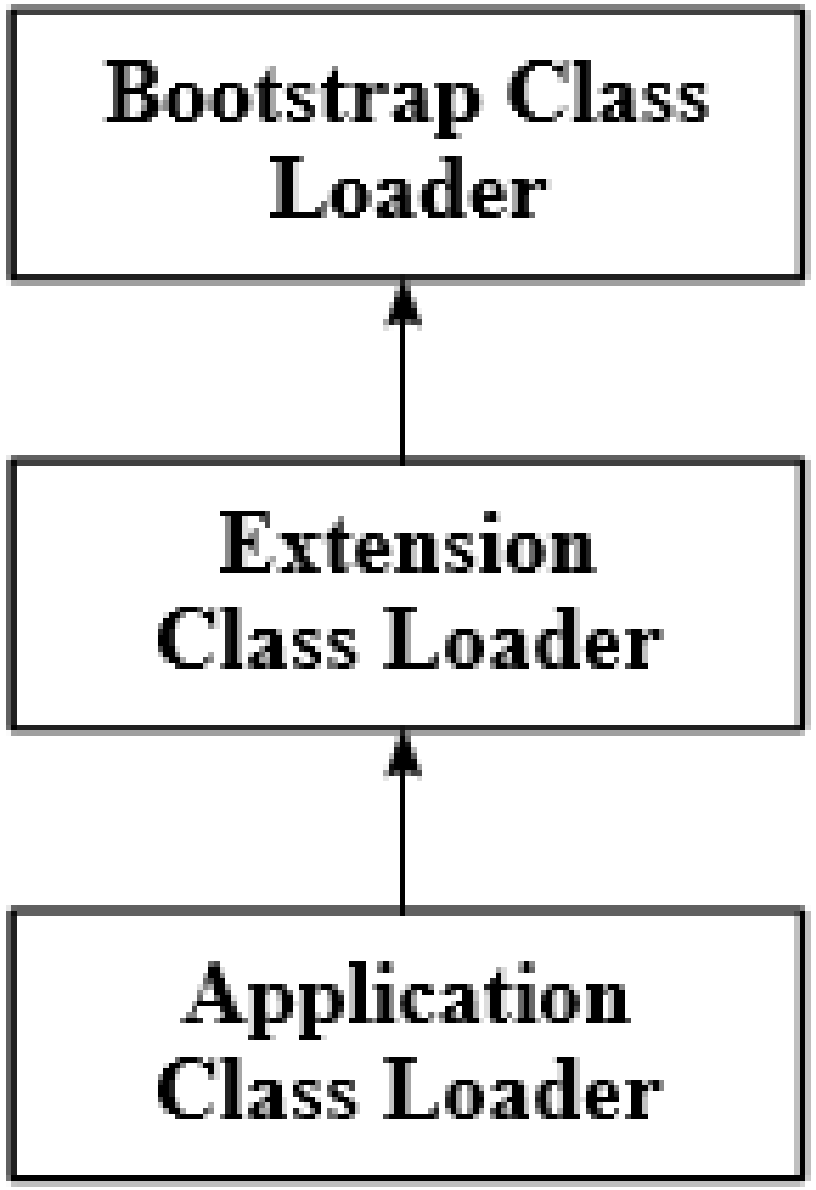
- Application class loader is the child class of Extension class loader

- This class loader is responsible to load classes from application class path

- It internally uses environment variable classpath

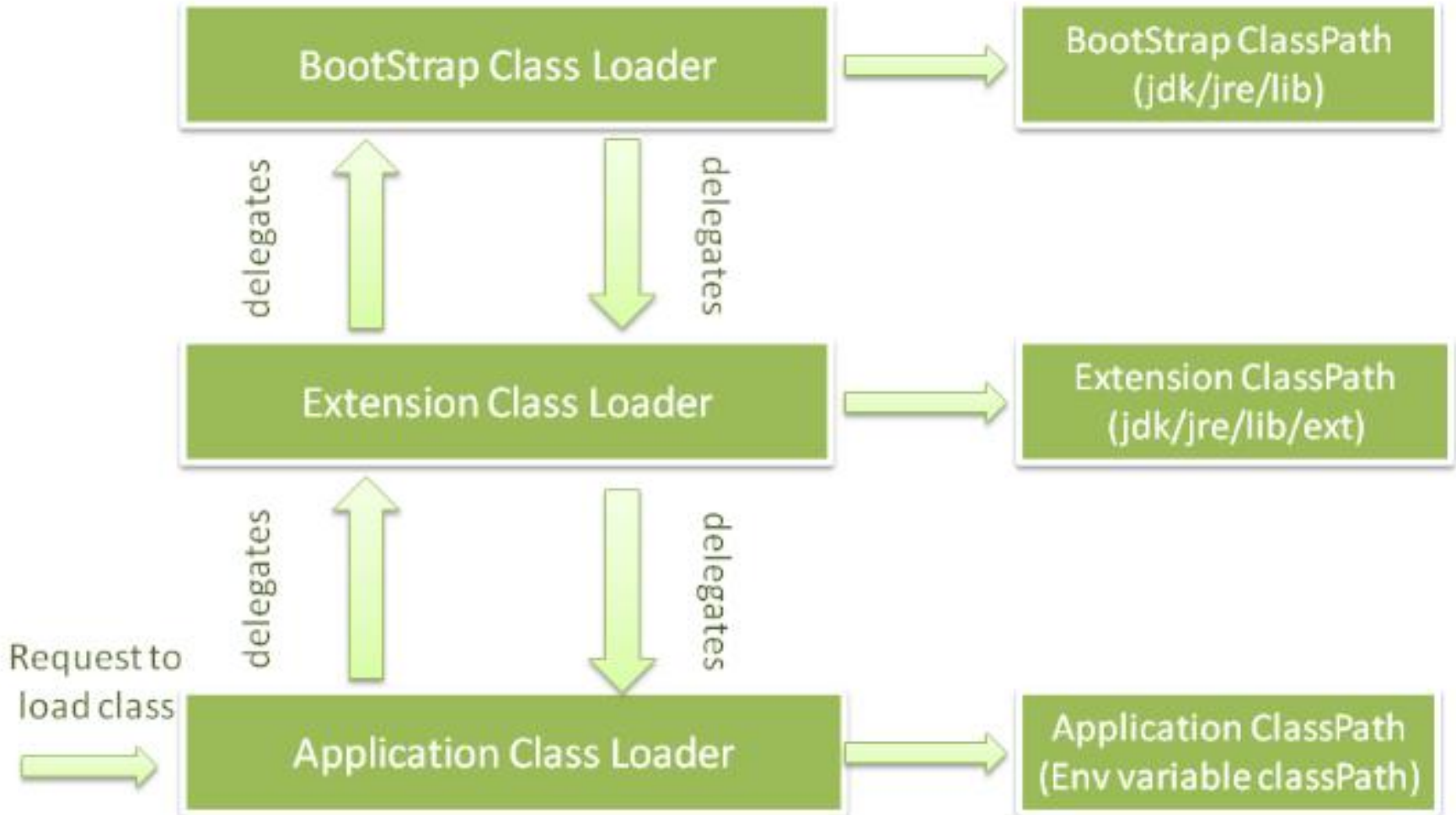
- Application class loader is implemented in java and the corresponding .class file name is

- sun.misc.Launcher$ApppClassLoader.class


Bootstrap Class Loader
Extension Class Loader
Application Class Loader

➢How class loader works

Class Loader Delegation Algorithm



- Class loader follows delegation hierarchy algorithm

- Whenever JVM come across a particular class first it will check whether the corresponding .class file is already loaded or not

- If it is already loaded in the method area then JVM will consider that loaded class

- If it is not loaded then JVM request class loader subsystem to load that particular class

- Then class loader subsystem handovers the request to application class loader

- Application class loader delegates the request to extension class loader

- Extension class loader delegates the request to the bootstrap class loader

- The Bootstrap class loader will search the class in Bootstrap class path if it is available then corresponding class will be loaded by Bootstrap class loader

- If the class if not available in the Bootstrap class path then the Bootstrap class loader delegates the request to the Extension class loader

- Extension class loader will search in Extension class path

- If the class is available in extension class path then it will be loaded

- Otherwise extension class loader delegates the request to the Application class loader

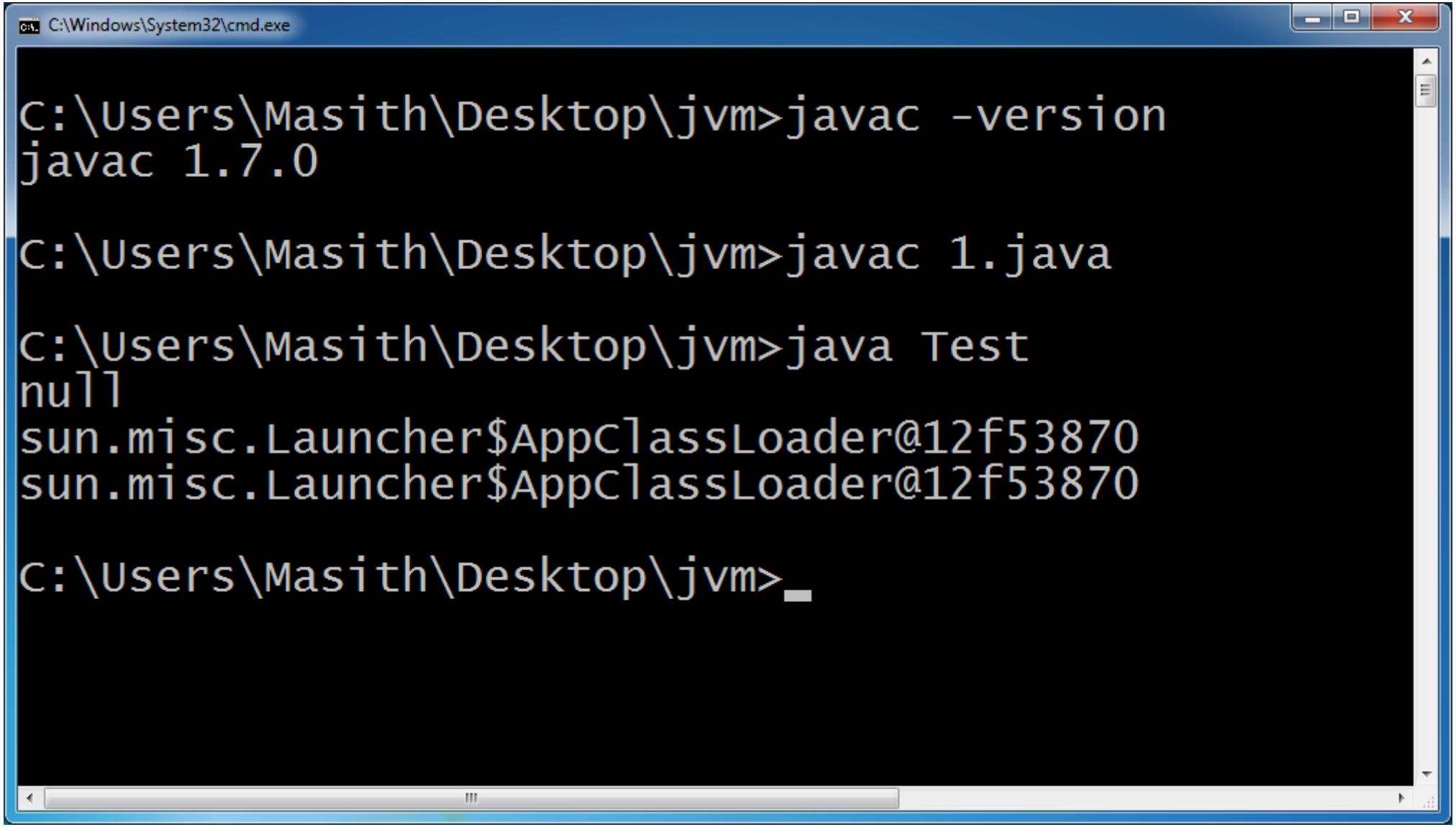
- Application class loader will search the class in application class path

- If the required class file is available in application class path then it will be loaded

- Otherwise we will get Runtime exception saying ClassNotFountException

```
class Customer {

}
```

```
class Test {
    public static void main(String[] args){
        System.out.println(String.class.getClassLoader());
        System.out.println(Customer.class.getClassLoader());
        System.out.println(Test.class.getClassLoader());
    }
}
```


C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>javac -version
javac 1.7.0
C:\Users\Masith\Desktop\jvm>javac 1.java
C:\Users\Masith\Desktop\jvm>java Test
null
sun.misc.Launcher$AppClassLoader@12f53870
sun.misc.Launcher$AppClassLoader@12f53870
C:\Users\Masith\Desktop\jvm>

- for String.class

Bootstrap class loader from bootstrap class paths

- for Tests.class

  Application class loader from application class paths

- for Customer.class

extension class loader is from extension class path

- Bootstrap class loader is not a java object
- Hence we got null in the first case
- But extension and application class loaders are java objects
- Hence we are getting corresponding output

- Class loader subsystem will give the highest priority for bootstrap class path and then extension class path followed by application class path

- Need of customized class loader

- Default class loaders will load .class files only once even though we are using multiple times that class in out program

- After loading .class file if it is modified outside then default class loader will not load and will not update version of the class file

- Because .class file is already available in method area

- We can resolve this problem by defining our own customized class loader

- The main advantage of customized class loader is we can control class loading mechanism based on our requirement

- For a example we can load .class file separately every time so that updated version of .class file is available to our program

```
public class CustomizedClassLoader extends ClassLoader {
    public Class loadClass(String className)throws ClassNotFoundException{
        /*
            check for updats

            load the updated .class file

            return corresponding class Class object

        */
    }
}
```

```
class Test {
    public static void main(String[] args){
        Dog d1 = new Dog();
        CustomizedClassLoader cl = new CustomizedClassLoader();
        cl.loadClass("Dog");
    }
}
```

- We can define our own customized class loader by extending java.lang.ClassLoader class

- While developing web servers and application servers usually we can go for customized class loaders to customized class loading mechanism

➢What is the need of classloader class

we can use java.lang.ClassLoader class to define our own customized class loaders

- Every customized class loader in java should be child class of java.lang.ClassLoader class either directly or indirectly

- Hence this class act as base class for all customized class loaders

Various Memory Areas Present Inside JVM

- Whenever JVM loads and runs a java program it needs memory to store several things like byte code objects variables etc…

- Total JVM memory organized into following 5 categories

1) Method Area
2) Heap Area
3) Stack Area
4) Native Method Stack
5) PC Registers

➢Method Area

- For every JVM one method area will be available
- Method area will be created at the time of JVM startup
- Inside method area class level binary data including static variables will be stored
- Constant pool of a class will be stored inside method area

- Method Area can be accessed by multiple threads simultaneously

- Heap Area

- Heap area will be created at the time of JVM startup
- For every JVM one heap area is available
- Objects and corresponding instance variables will be stored in the Heap Area
- Every array in java is object only hence arrays also will be stored in the Heap area

- Heap area can be accessed by multiple threads
- Hence the data stored in the Heap memory is not thread safe
- Heap area need not be continuous

- Program to display heap memory statistics

- java application can communicate with JVM by using Runtime object

- Runtime class present in java.lang package and it is a singleton class

- We can create Runtime object as follows

```
Runtime r = Runtime.getRuntime();
```

- Once we create Runtime object we can call the following methods on that object

maxMemory() -

It returns no of bytes of max memory allocated to the heap

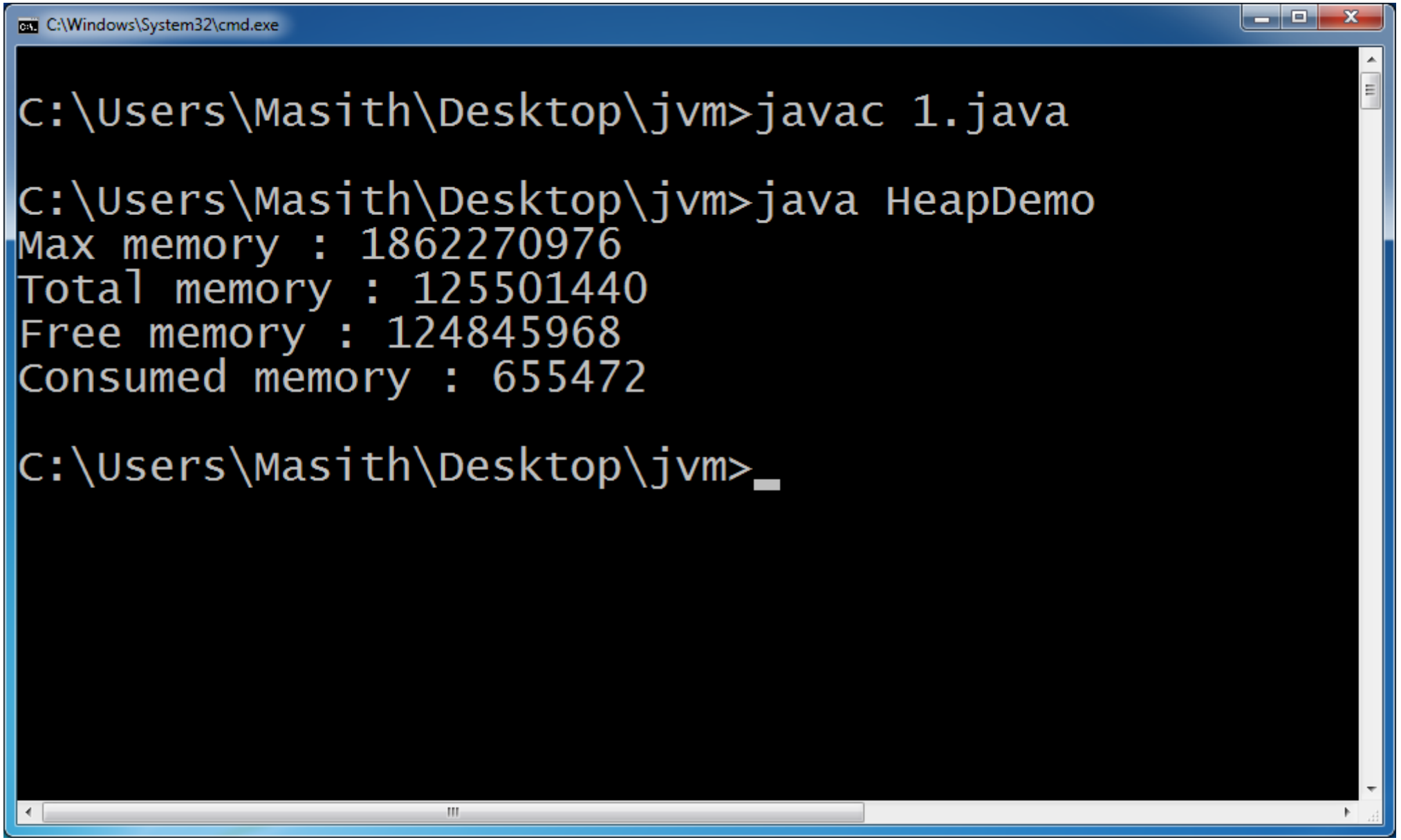
totoalMemory() -

It returns no of bytes of total memory allocated to the heap (initial memeory)

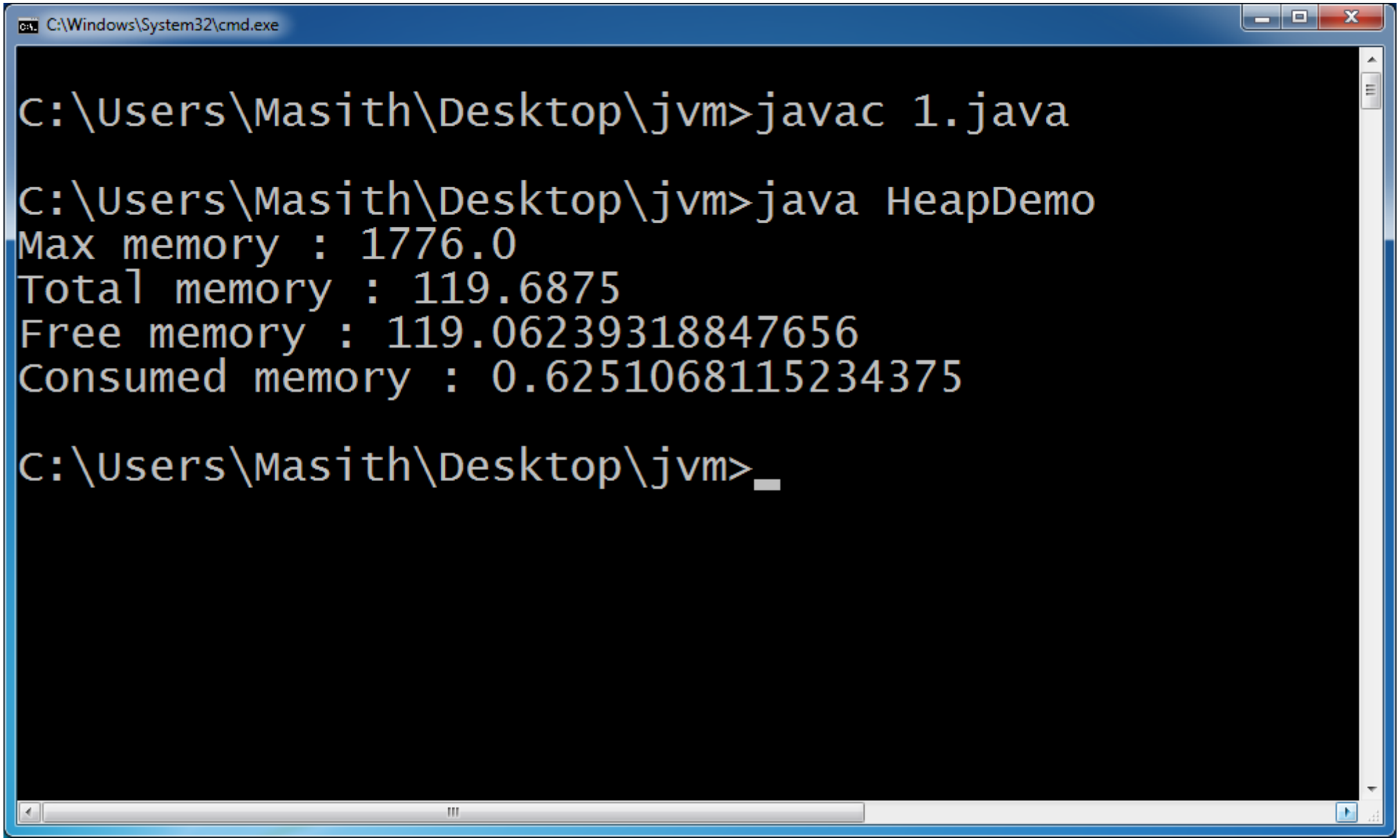
freeMemory() -

It returns no of bytes of free memory present in the heap

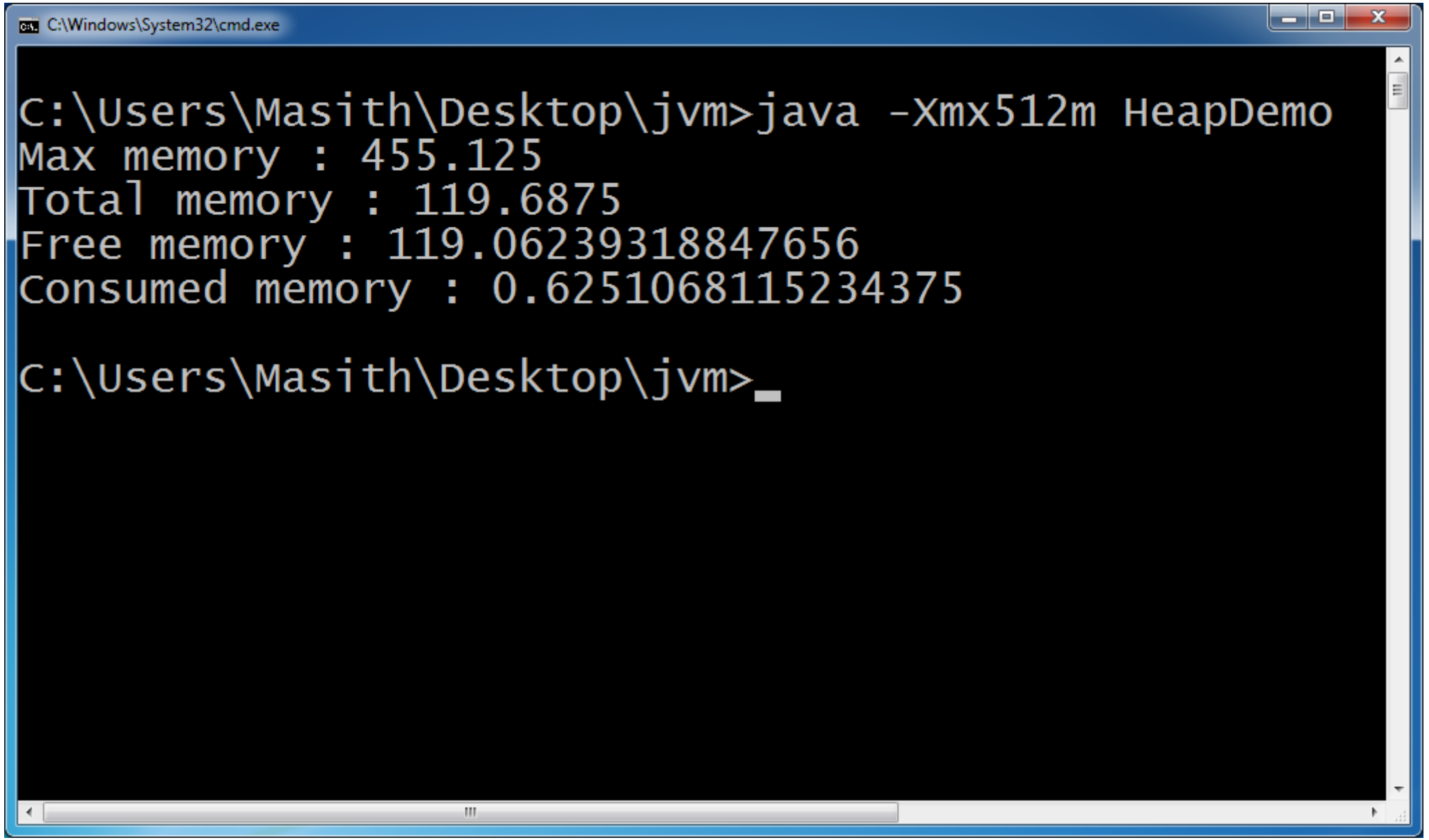
```
class HeapDemo {
    public static void main(String[] args){
        Runtime r = Runtime.getRuntime();
        System.out.println("Max memory : "+r.maxMemory());
        System.out.println("Total memory : "+r.totalMemory());
        System.out.println("Free memory : "+r.freeMemory());
        System.out.println("Consumed memory : "+
        (r.totalMemory()-r.freeMemory()));
    }
}
```

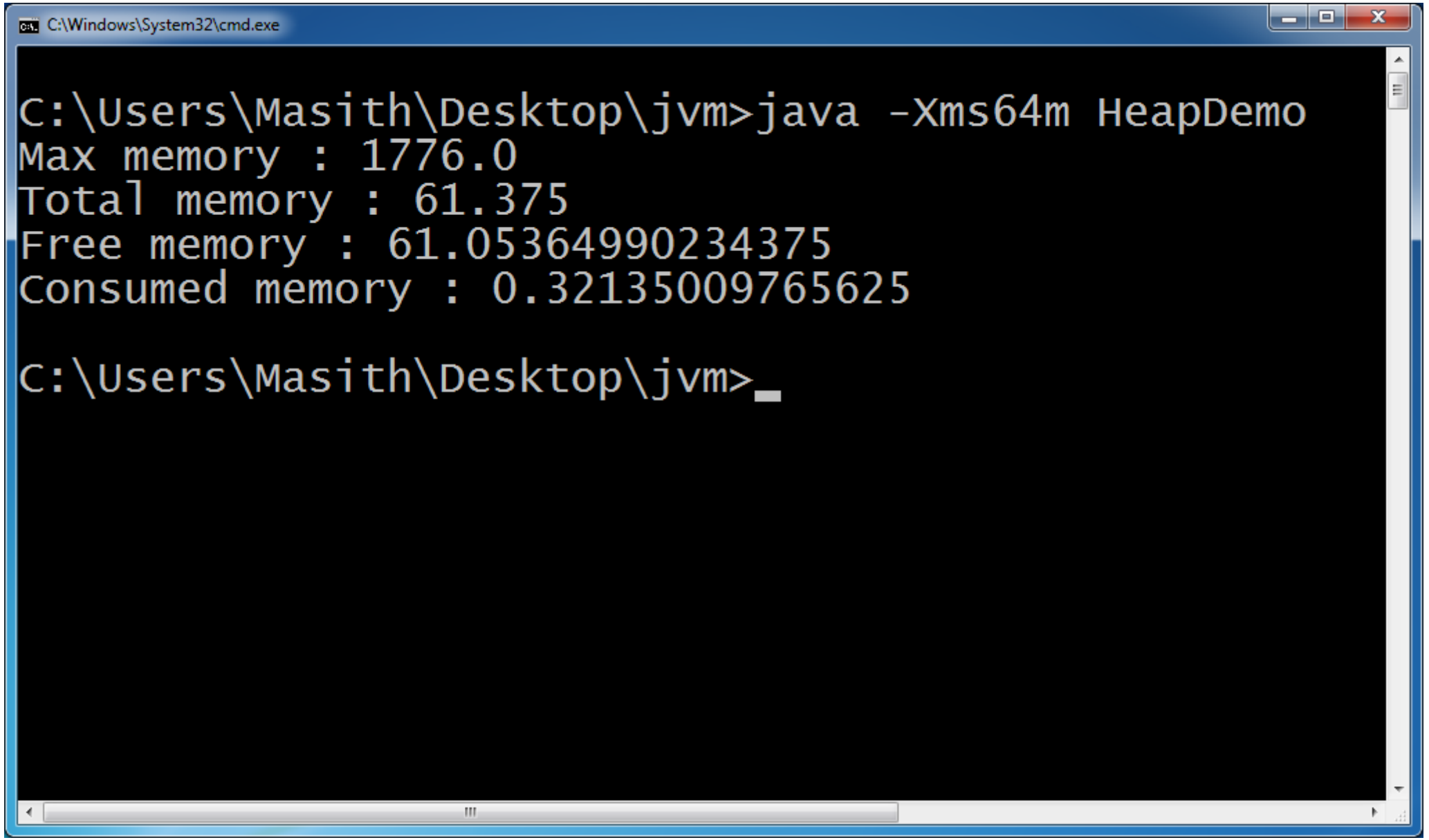

C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>javac 1.java
C:\Users\Masith\Desktop\jvm>java HeapDemo
Max memory : 1862270976
Total memory : 125501440
Free memory : 124845968
Consumed memory : 655472
C:\Users\Masith\Desktop\jvm>

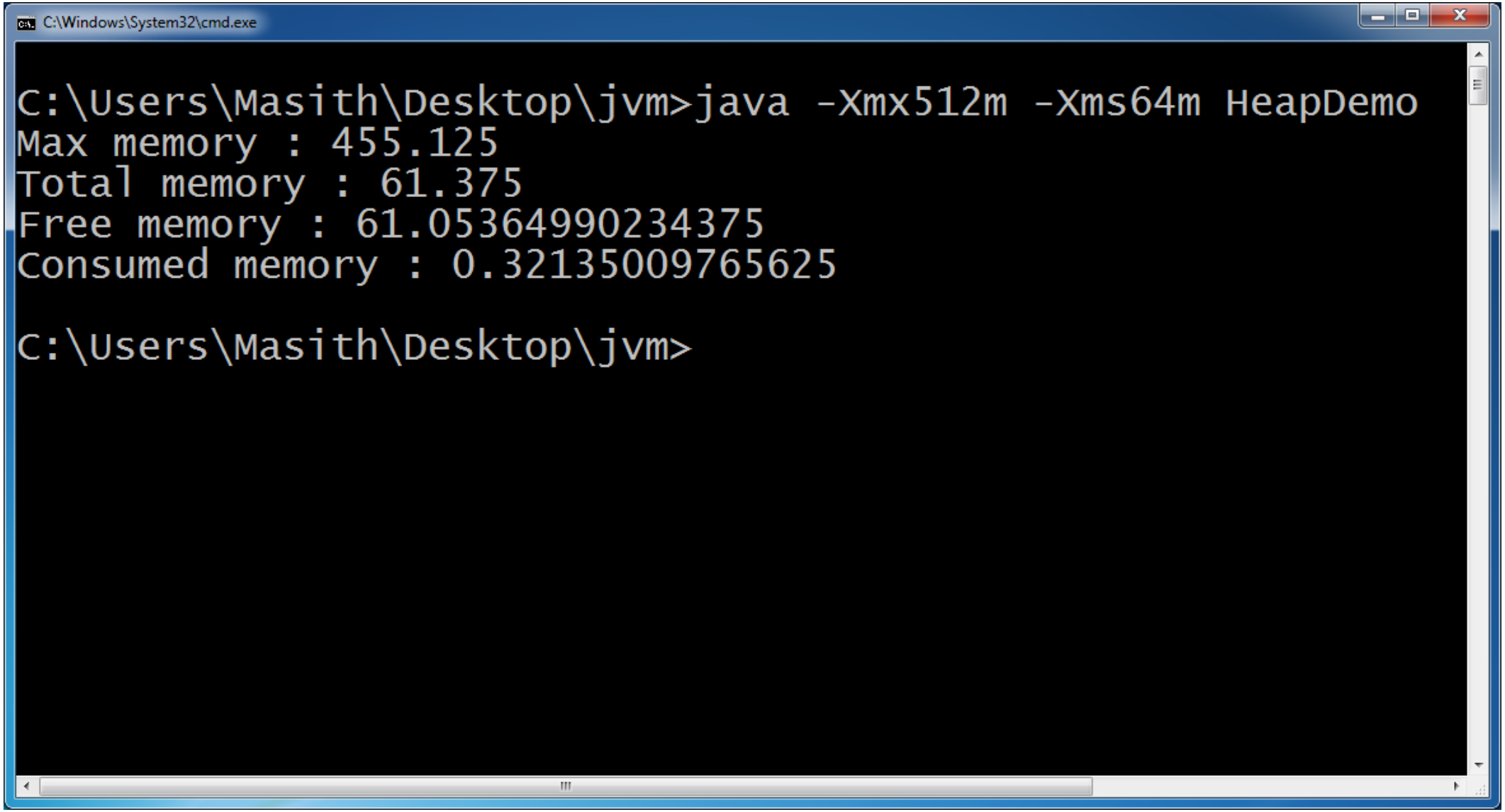
```
class HeapDemo {
    public static void main(String[] args){
        double mb = 1024*1024;
        Runtime r = Runtime.getRuntime();
        System.out.println("Max memory : "+r.maxMemory()/mb);
        System.out.println("Total memory : "+r.totalMemory()/mb);
        System.out.println("Free memory : "+r.freeMemory()/mb);
        System.out.println("Consumed memory : "+
        (r.totalMemory()-r.freeMemory())/mb);
    }
}
```


C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>javac 1.java
C:\Users\Masith\Desktop\jvm>java HeapDemo
Max memory : 1776.0
Total memory : 119.6875
Free memory : 119.062393188476 56
Consumed memory : 0.625106811523437 5
C:\Users\Masith\Desktop\jvm>

- How to set maximum and minimum heap size


C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>java -Xmx512m HeapDemo
Max memory : 455.125
Total memory : 119.6875
Free memory : 119.062393188847656
Consumed memory : 0.625106811523437 5
C:\Users\Masith\Desktop\jvm>


C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>java -Xms64m HeapDemo
Max memory : 1776.0
Total memory : 61.375
Free memory : 61.05364990234375
Consumed memory : 0.32135009765625
C:\Users\Masith\Desktop\jvm>


C:\Windows\System32\cmd.exe
C:\Users\Masith\Desktop\jvm>java -Xmx512m -Xms64m HeapDemo
Max memory : 455.125
Total memory : 61.375
Free memory : 61.05364990234375
Consumed memory : 0.32135009765625
C:\Users\Masith\Desktop\jvm>

- Heap memory is finite memory
- But based on our requirement we can set maximum and minimum heap sizes
- That is we can increase or decrease the heap size based on our requirement

➢Stack Memory Area

```
class Test {
    public static void main(String[] args){
        m1();
    }
    public static void m1(){
        m2();
    }
    public static void m2(){

    }
}
```

- For every thread JVM will create a separate stack at the time of thread creation

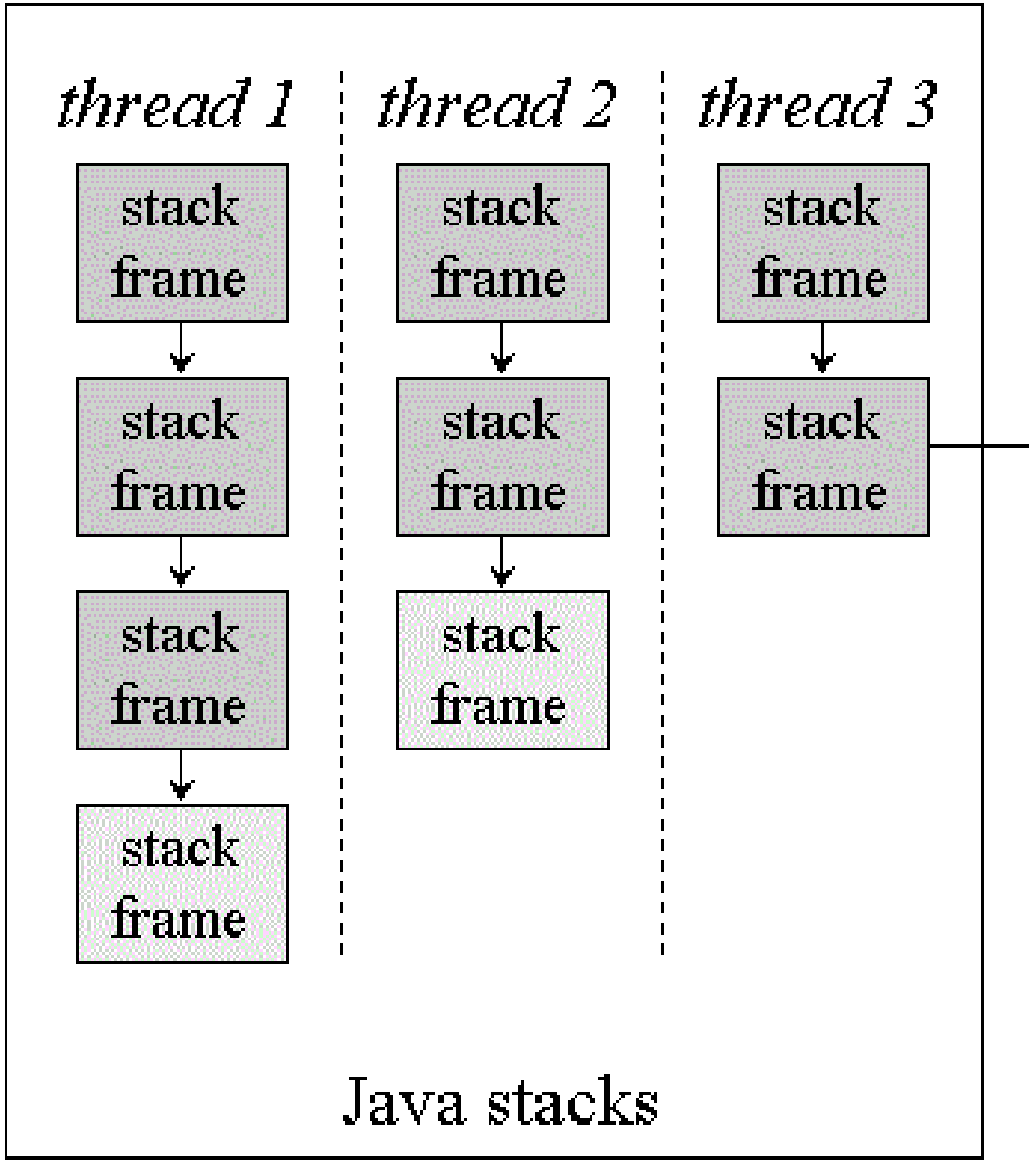
- Each and every method call performed by that thread will be stored in the stack including local variables as well

- After completing a method the corresponding entry from the stack will be removed

- After completing all method calls the stack will become empty

- An empty stack will be destroyed by the JVM before terminating the thread

- Each entry in the stack is called stack frame or activation record


thread 1
thread 2
thread 3
stack frame
stack frame
stack frame
stack frame
stack frame
stack frame
stack frame
stack frame
stack frame
Java stacks

- The data stored in the stack is available only for the corresponding thread and not available to the remaining threads

- Hence this data is thread safe

➢Stack frame structure

- Each stack frame contains 3 parts
    1). Local Variable Array
    2). Operand Stack
    3). Frame Data

➢Local Variable Array

- It contains all parameters and local variables of the method
- Each slot in the array is of 4 bytes
- Values of type int float and reference occupy one entry in the array

- byte short and char values will be converted to int type before storing and occupying one slot

- But the way of storing boolean value is varied from jvm to jvm

- But most of jvm follows one slot for boolean values

- Operand stack

- jvm uses operand stack as work space
- Some instructions can push the values to the operand stack and some instructions can pop the values from operand stack
- And some instructions can perform required operations

➢Frame Data

- Frame data contains all symbolic references related to that method

- It also contains reference to exception table which provides corresponding catch block information in the case of exceptions

➢PC Register

- For every thread a separate PC register will be created at the time of thread creation
- PC registers contains the address of currently executing instruction
- Once instruction execution completes automatically pc register will be updated to address of next instruction

- Native method stacks

- For every thread JVM will create a separate native method stack

- All native method calls invoked by the thread will be stored in the corresponding native method stack

- For every JVM one heap area and one method area is created

- For every thread one stack area one pc register and one native method stack will be created

- Static variables will be stored in method area
- Instance variables will be stored in heap area
- Local variables will be stored in stack area

```
class Test {
    Student s1 = new Student();
    static Student s2 = new Student();
    public static void main(String[] args) {
        Test t = new Test();
        Student s3 = new Student();
    }
}

class Student{ }
```

Execution Engine

- This is the central components of JVM

- Execution engine is responsible to execute java class files

- Execution engine mainly contains two components
    1). Interpreter
    2). JIT compiler

- Interpreter

- It is responsible to read byte code and convert it to machine code (native code) and execute that machine code line by line

- The problem with interpreter is it intrepits every times even some method invoked multiple times which reduces the performance of the system

- To overcome this problem sun microsystem has introduced JIT compiler in java 1.1 version

- JIT compiler

- The main purpose of JIT compiler is to improve performance

- Internally JIT compiler maintains a separate count for every method

- Whenever JVM come across any method call first that method will be interpreted normally by the interpreter

- And JIT compiler increments the corresponding count variable

- This process will be continued for every method

- Once if any method count reaches threshold value then JIT compiler will identify that method is a repeatedly used method

- Such kind of method is called HotSpot

- Immediately JIT compiler compiles that method and generates the corresponding native code

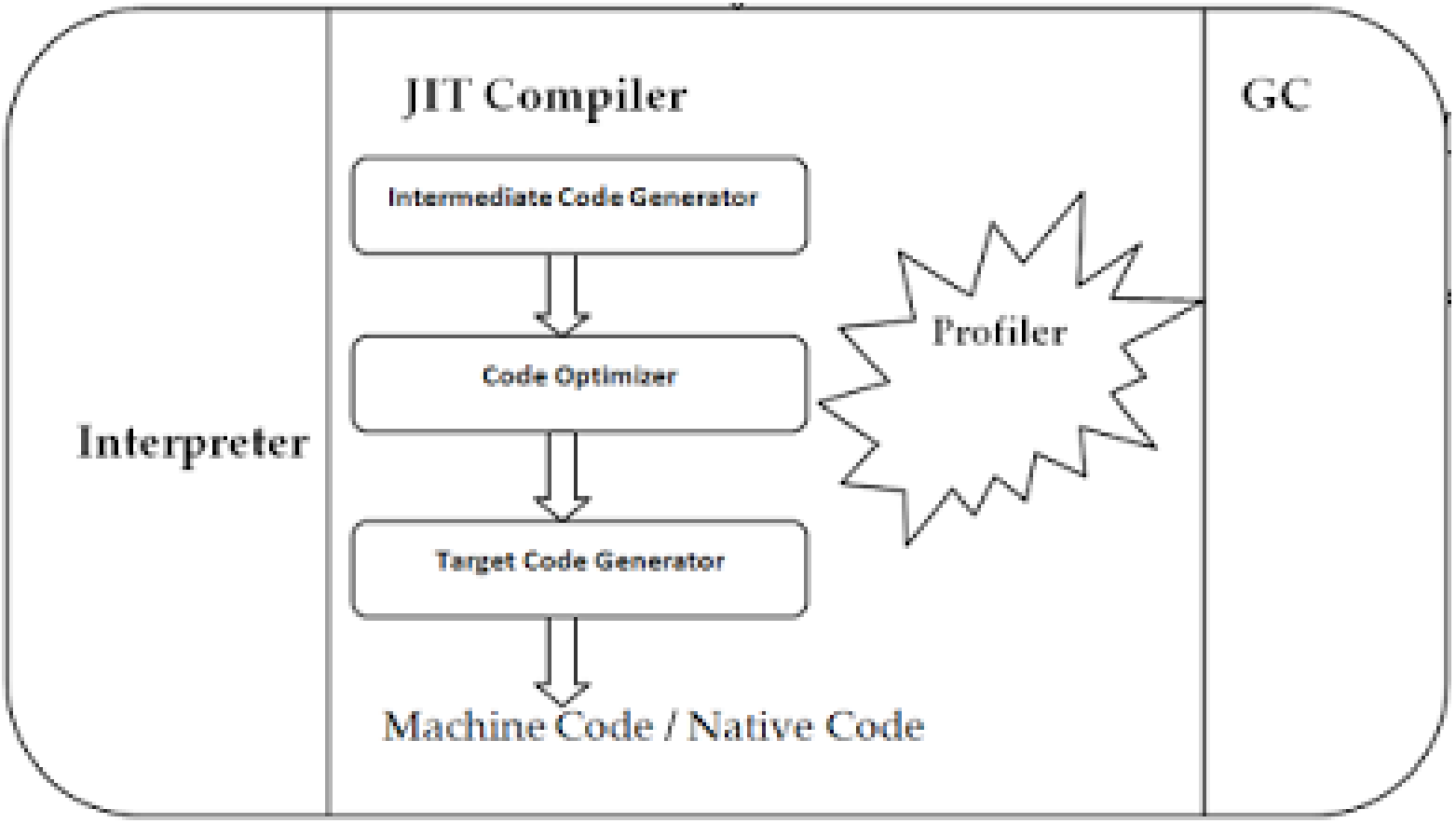
- Next time JVM come across that method call then JVM uses native code directly and executes it instead of interpreting once again

- So the performance of the system will be improved
- The threshold count is varied from JVM to JVM
- Some advanced JIT compilers will recompile generated native code if count reaches threshold value second time
- So more optimized machine code will be generated

- Internally profiler which is the part of JIT compiler is responsible to identify HotSpots

- JVM interprets total program at least once

- JIT compilation is applicable only for repeatedly required methods not for every methods

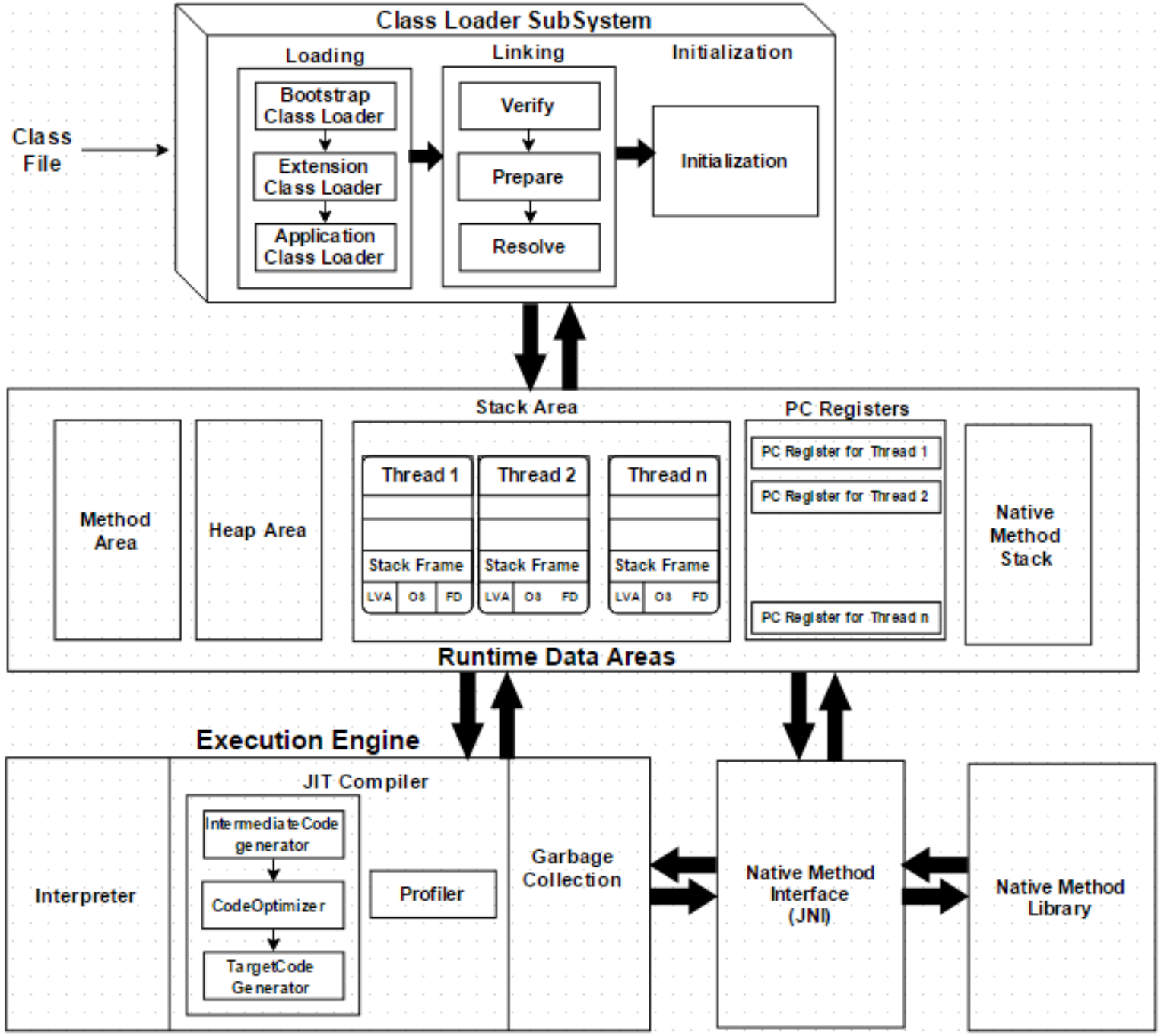


Execution Engine

➢Java Native Interface (JNI)

- JNI acts as mediator for java method calls and corresponding native libraries

- That is JNI is responsible to provide information about native libraries to the JVM


Class Loader SubSystem
Loading
Linking
Initialization
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Verify
Prepare
Resolve
Initialization
Class File
Stack Area
PC Registers
Method Area
Heap Area
Thread 1
Thread 2
Thread n
Stack Frame
LVA
OS
FD
PC Register for Thread 1
PC Register for Thread 2
PC Register for Thread n
Native Method Stack
Runtime Data Areas
Execution Engine
JIT Compiler
Intermediate Code generator
CodeOptimizer
Target Code Generator
Profiler
Interpreter
Garbage Collection
Native Method Interface (JNI)
Native Method Library

➢Class file structure

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```


magic

- The first 4bytes of the class file is magic number
- This is a predefined value used by JVM to identify .class file is generated by valid compiler or not
- The value should be 0XCAFEBABE

- Whenever we are execution a java class if JVM is unable to find valid magic number

- Then we will get runtime exception saying Incompatible magic value

- minor_version major_version

- Major and minor version represents .class file version

- JVM will use these versions to identify which version of compiler generates the current .class file

- M.m M is major version m is minor version

- Lower version compiler generated .class files can be run by higher version JVM

- But higher version compiler generated .class files can not be run by lower version JVM

- If we are trying to run we will get runtime exception saying UnsupportedClassVersionError

- constant_pool_count

- It represents no of constants present in constant pool

- constant_pool[ ]

- It represents information about constant present in constant pool

- access_flags

- It provides information about modifiers which are declared to the class

- this_class

- It represents fully qualified name of the class

- super_class

- It represents fully qualified name of immediate super class of current class

- interface_count

- It returns no of interfaces implemented by current class

- Interface[ ]

- It returns interfaces information implemented by current class

- fields_count

- It returns interfaces no of fields in the current class

- field are static variables

- fields[ ]

- It represents field information present in current class

- methods_count

- It represents no of methods present in current class

- methods[ ]

- It provides information about all methods present in current class

- attributes_count

- It returns no of attributes present in current class

- attributes[ ]

- It provides information about all attributes present in current class

- attributes means instance variables

- javap –verbose Test.class

End