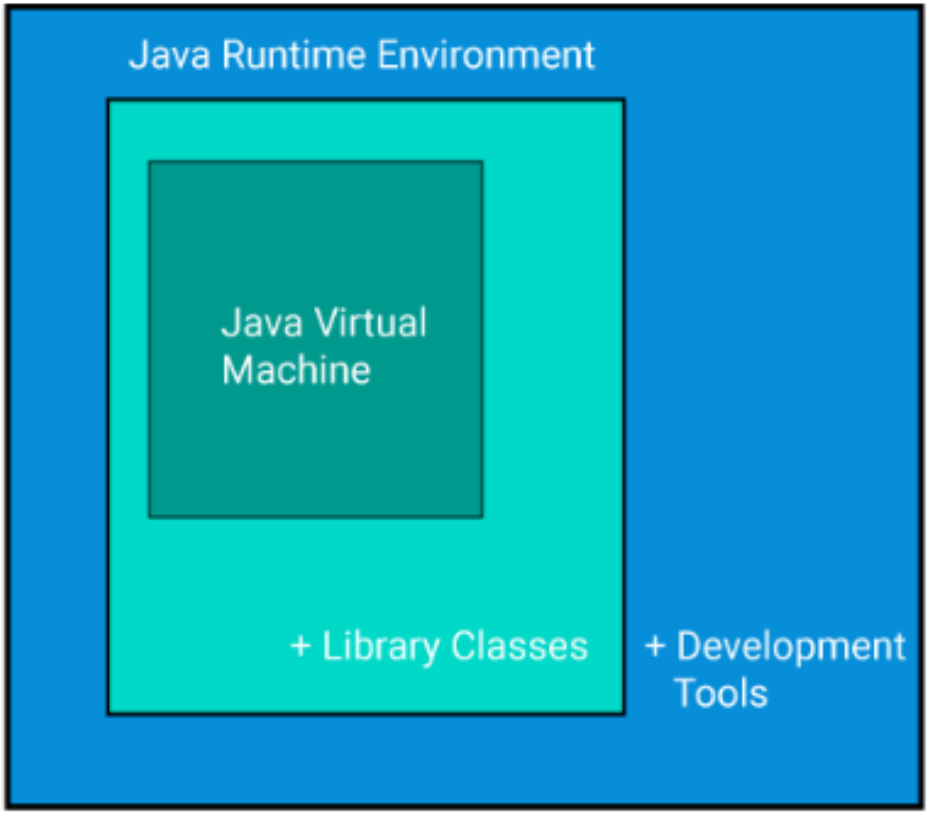
Java Architecture

Java Architecture in a nutshell

- The code written in Java, is converted into byte codes which is done by the Java Compiler.
- The byte codes, then are converted into machine code by the JVM.
- The Machine code is executed directly by the machine.

High level diagram of JVM



JDK = JRE + Development Tool

JRE = JVM + Library Classes

Development tools

- **java** : it is the launcher for all the java applications.
- **javac** : complier of the java programming languages.
- **javadoc**: it is the API documentation generator.
- **jar**: creates and manage all the JAR files.

How is Java platform independent?

- When is any programming language called as platform-independent?
- if and only if it can run on all available operating systems with respect to its development and compilation.
  Now, Java is platform-independent just because of the bytecode.
- Bytecode is a code of the JVM which is machine-understandable.
  Bytecode execution in Java proves it is a platform-independent language.
  steps involved in the process of java bytecode execution.
- ***sample.java → javac (sample. class) → JVM(sample.obj) → final output***

Behavior of JVM

- Is JVM available any time ?

  No ...then ,

- Until a JAVA application runs ,a JVM instance is executing ,IF not it will die ,

  When

- There is no any non daemon threads

  OR

- Application suicide by the System. Exit()

Compile - Creates the .class file

```
D:\waruni>javac NumPrinter.java

D:\waruni>java NumPrinter
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
exit...
```

Execute - Ask OS to create a JVM instance

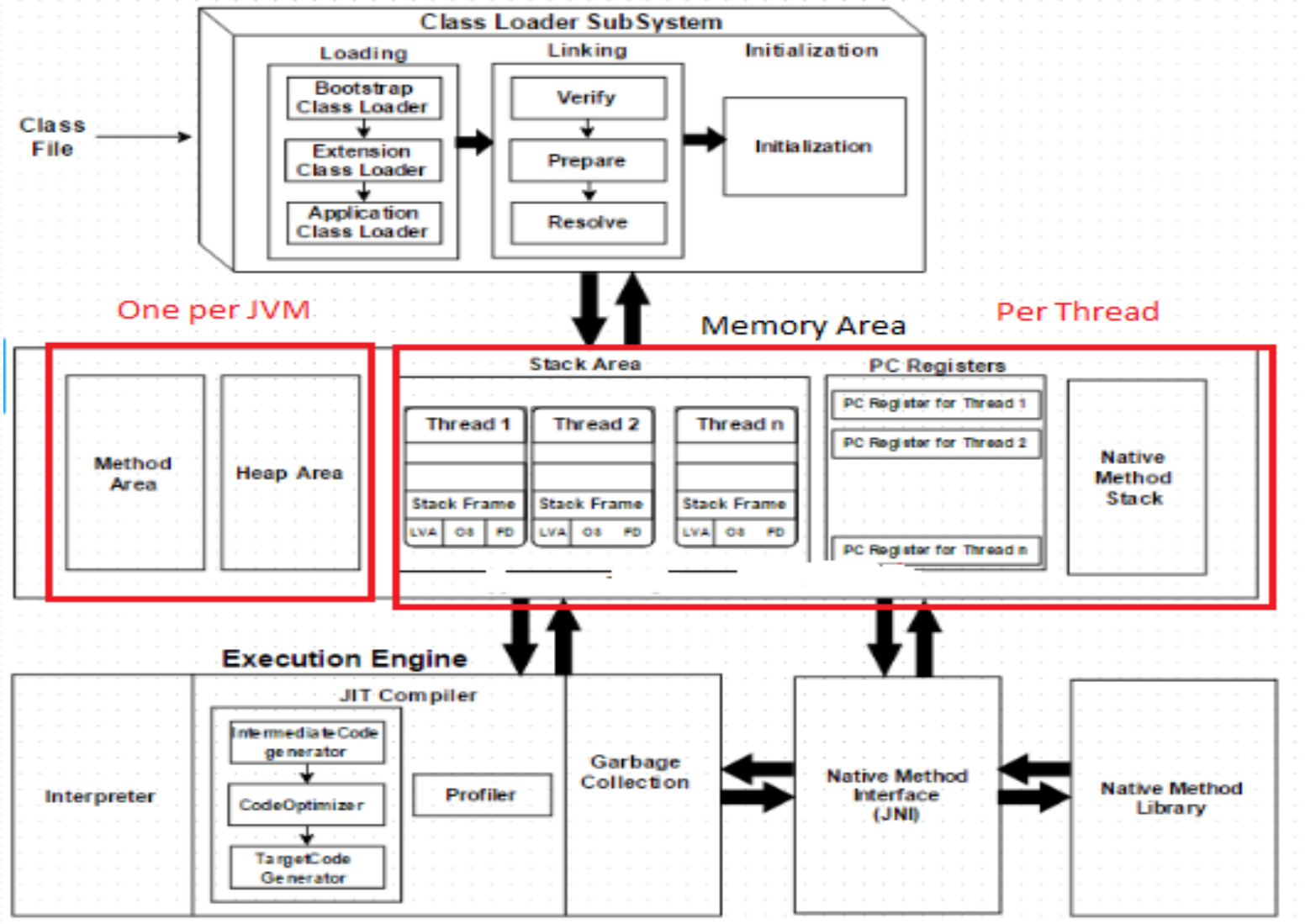
JVM takes the Main method for its execution

```
class NumPrinter
{
    public static void main(String[] args)
    {
        int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};

        for (int i = 0; i < arr.length; i++)
        {
            if (arr[i] >= 5)
            {
                System.out.println("exit...");

                // Terminate JVM
                System.exit(0);
            }
            else
                System.out.println("arr["+i+"] = " +
                                   arr[i]);
        }
        System.out.println("End of Program");
    }
}
```

JVM Architecture Diagram



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

Linking

- Performs verification, preparation, and resolution.
- *Verification* :
  - It ensures the correctness of *.class* file
  - Check whether this file is properly formatted &
  - generated by valid compiler or not.
- If verification fails, we get run-time exception *java.lang.VerifyError*. That defines the file has been altered .

- So , as a result of the Bytecode Verifier , JAVA gets the safety to run .

- *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.
- Every class must be initialized before its active use.

Variations of being a Active use of a Class

- When creating a object

New key word - Student s1 = new Student ();

- Invoking a Static method (When object is not created)

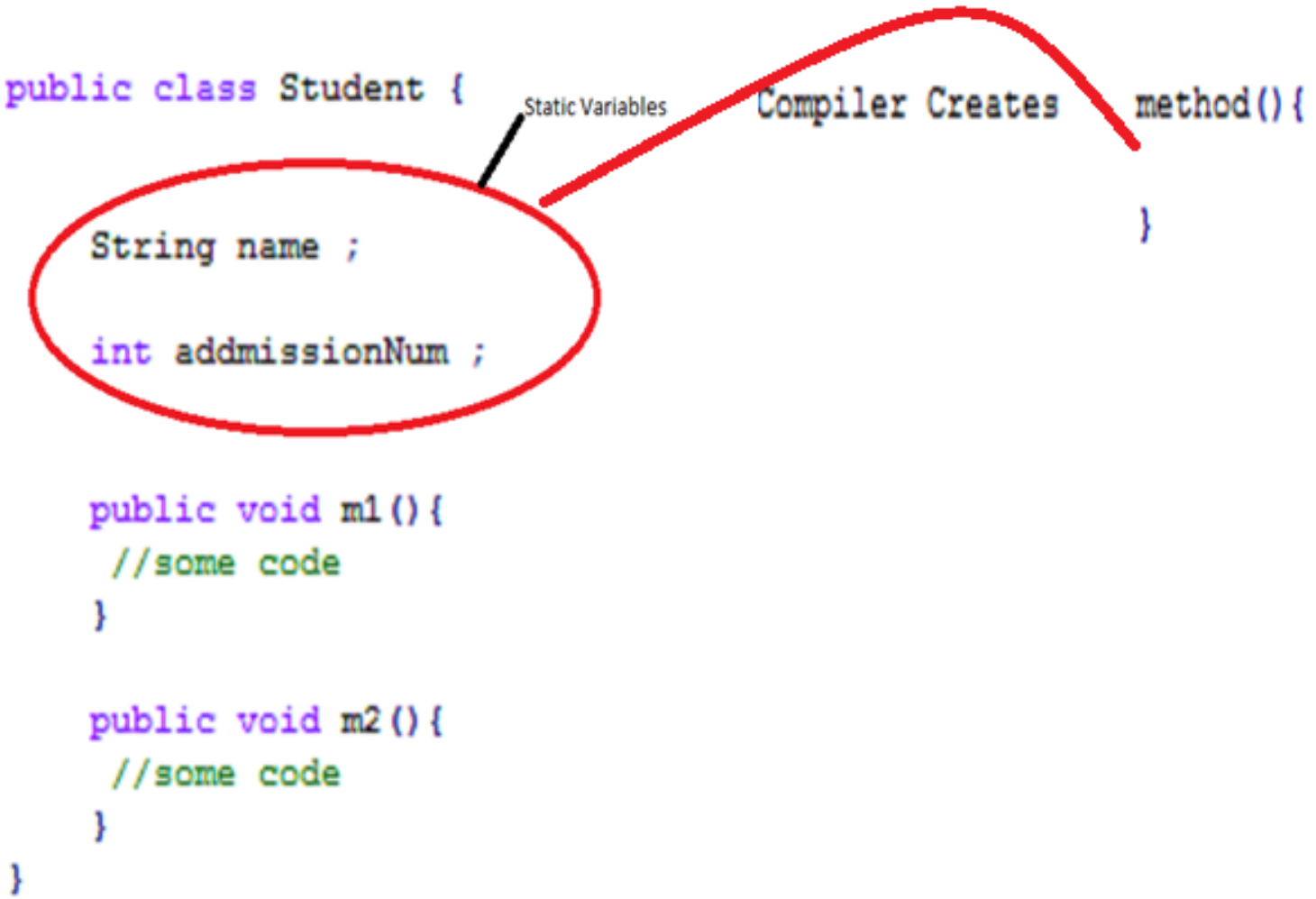
If Student class has a method called “verify”

Student.verify();

- Assign a value for the static field ,

  Student.grade= “one” ;

  (If the variable declared as final , will not be considered for Initialization)
- Initial class – (main)
- getInstance(); - Reflection
- Instantiation of a sub class


public class Student {
Static Variables
Compiler Creates
method(){
}
String name ;
int addmissionNum ;
public void m1(){
//some code
}
public void m2(){
//some code
}
}

Ways of Initializing a class by JAVA

- ***New* key word**
- ***Clone() ;* - Assign initial value from Parent class**
- ***getInstance();***
- **IO.ObjectInputStream ;**

How JAVA go through constructors and instantiate parent class

```
public class Human {

    Human ();



    //some code

}

class Student extends Human{

    String name ;

    int addmissionNum ;

  //some code
}
```

```
public class Human {

    int age ;

    Human ();

    Human(int age)



    //some code

}

class Student extends Human{

    String name ;

    int addmissionNum ;

  //some code
}
```

When pervious scenario will not be happened

```
public class Human {

    private  Human()

    //some code

}

class Student extends Human{

    String name ;

    int addmissionNum ;

  //some code
}
```

```
public class Human {

    int age ;

    Human(int age)

    //some code

}

class Student extends Human{

    String name ;

    int addmissionNum ;

  //some code
}
```

JIT in Java

- Just In Time compiler commonly known as JIT, is basically responsible for performance optimization of java based applications at run time. The performance of an application is dependent on a compiler.