Overview of Java

The Creation of Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995. Now it is merged into Oracle Corporation. Java is used to develop Desktop Applications such as MediaPlayer, Antivirus, Develop Web Applications, Develop Enterprise Application such as Banking applications, Mobile Applications, Develop Embedded System, SmartCards, Robotics, Games etc.

As per the sun micro system standard the java language is divided into three types.

1. **J2SE/JSE (java 2 standard edition)**
   J2SE is used to develop the standalone applications. Ex: - notepad, WordPad, paint, Google Talk, etc.
2. **J2EE/JEE (java 2 enterprise edition)**
   With J2EE we develop web-based applications. Ex: Gmail, yahoo mail, bank, reservation, etc.
3. **J2ME/JME (java 2 micro edition)**
   By using J2ME we develop the applications that only run on mobile devices.

Java is guaranteed to be **Write Once, Run Anywhere.**

JAVA Features

1. **Simple:**
   Java is a simple programming language because:
   - Java technology has eliminated all the difficult and confusion-oriented concepts like pointers, multiple inheritance in the java language.
   - The c, cpp syntaxes easy to understand and easy to write. Java maintains C and CPP syntax mainly hence java is simple language.
   - Java tech takes less time to compile and execute the program.
2. **Object Oriented:**
   Java is object-oriented technology because to represent total data in the form of object. By using object reference we call all the methods, variables which is present in that class.
3. **Platform Independent:**
   Compile the Java program on one OS (operating system) and that compiled file can be executed in any OS (operating system) is called Platform Independent Nature. The java is platform independent language.
4. **Architectural Neutral:**
   Java application compiled in one Architecture (hardware----RAM, Hard Disk) can run on any hardware architecture(hardware) is called Architectural Neutral.
5. **Portable:**
   In Java tech the applications are compiled and executed in any OS (operating system) and any Architecture(hardware) hence we can say java is a portable language.
6. **Robust:**
   Any technology if it is good at two main areas it is said to be ROBUST
   - Exception Handling.
   - Memory Allocation.

   JAVA is Robust because:
   - JAVA has good predefined Exception Handling mechanism.
   - JAVA is having very good memory management system that is Dynamic Memory (at runtime the memory is allocated) Allocation which allocates and deallocates memory for objects at runtime.
7. **Secure**
   To provide implicit security Java provide one component inside JVM called Security Manager. To provide explicit security for the Java applications we are having very good predefined library in the form of java.Security.package.

8. **Multithreaded**
   Thread is a light weight process and a small task in large program. With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously.
9. **Compiled and interpreted**
   Usually a computer language is either compiled or interpreted but, java combines both these approaches. First, Java compiles translates source code into what is known as bytecode instructions. Bytecode are into machine instructions and therefore, in second stage, java interpreter generates machine code that can be executed by machine that is running the java program. Thus, java is both compiled and interpreted.

JDK, JRE, JVM

JDK (Java development kit)

In programming we develop a program, compile, run or execute for output. To Develop and run java application JDK is required.

JRE (Java runtime environment)

But just to run java application only JRE is required. In JRE JVM is responsible to run a program line by line that's why JVM is interpreter.

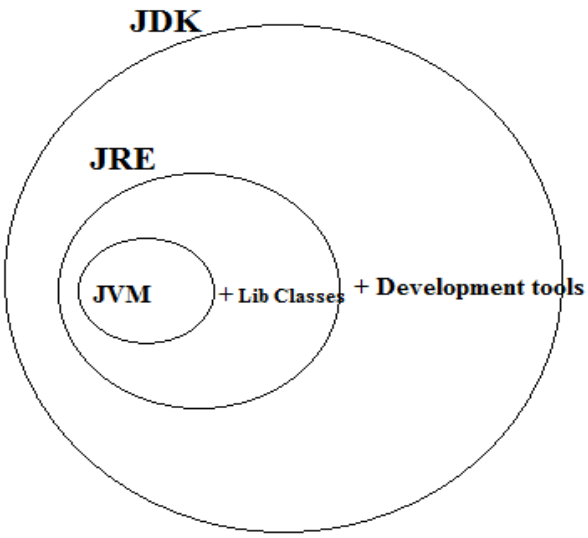


Fig: JDK

Class Path:

Describe location where required class file are available. Compiler and JVM uses class path to locate required class.

Path:

Describes the location where binary executable is available. Inside JDK\bin executables are available.

Java Virtual Machine:

A Virtual Machine is a Software implementation of a Physical Machine which can perform operations like a physical machine. Java was developed with the concept of WORA **(***Write Once Run Anywhere***)** which runs on a VM**.** The compiler will be compiling the java file into a java .class file. The .class file is input to JVM which Loads and executes the class file.

Hardware/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 machines is hardware resources sharing and improves utilization of hardware resources ex. KVM (Kernel Based virtual machine for Linux systems), VMware, Xen, Cloud computing etc.

Application/Process Based virtual machine:
Theses virtual machines act as runtime engines to run particular programming language applications.
Example.

1. JVM (Java virtual machine) acts as runtime engine to run java based application.
2. PVM (Parrot virtual machine) acts as runtime engine to run Perl based applications.
3. CLR (Common language runtime) acts as runtime engine to run .NET based applications.

Java Virtual Machine (JVM):
JVM is the part of JRE and it is responsible to load and run java class files.

Architecture of JVM

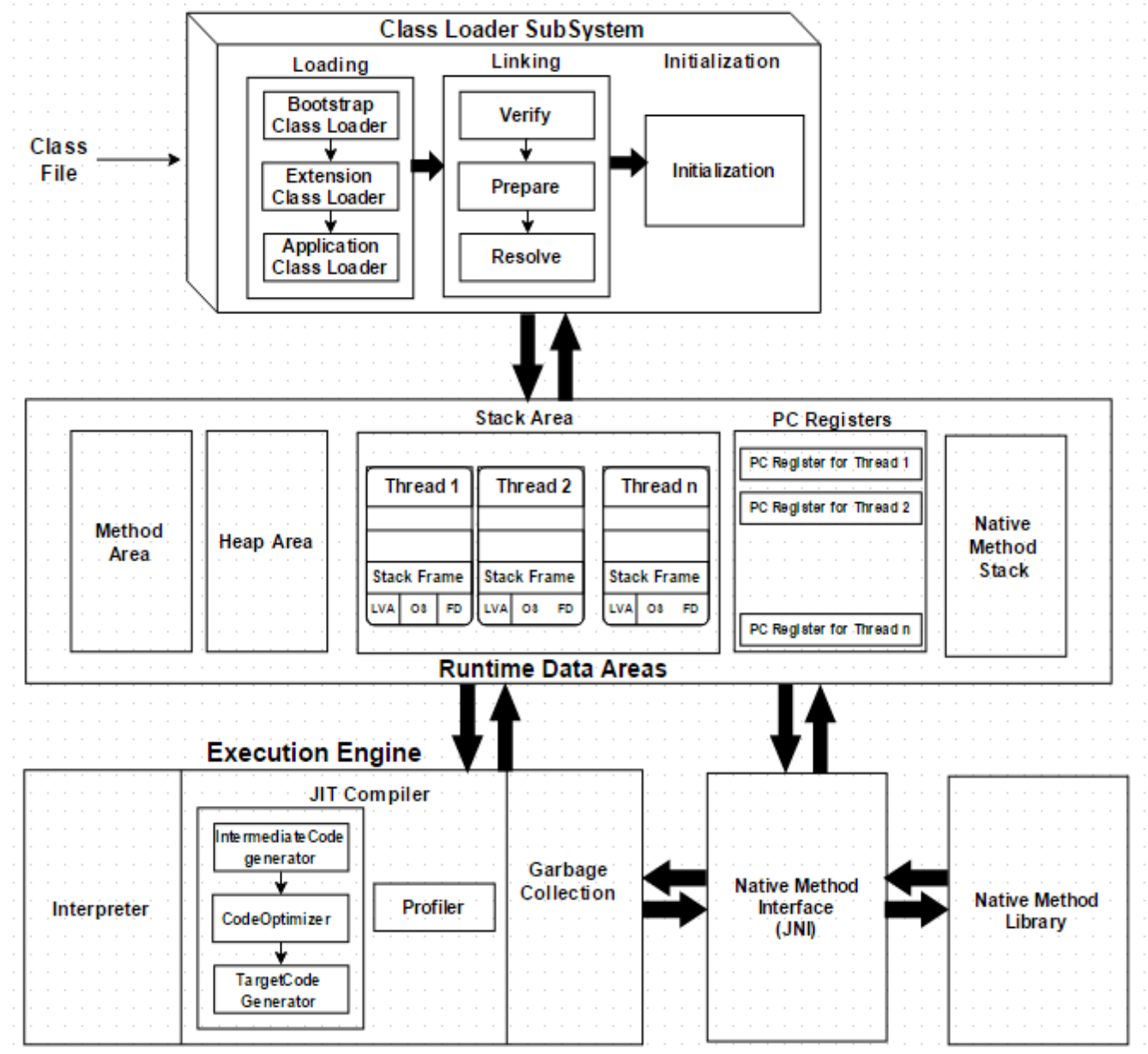


Fig 1: Java Virtual Machine

Class loader subsystem:
Class loader subsystem is responsible for the following three activities:

1. Loading
2. Linking

3. Initialization

1. **Loading:**
Loading means reading class files and store corresponding binary data in the method area. For each class file JVM will store corresponding information in method area.

1. Fully qualified names of class.
2. Fully qualified name of immediate parent class.
3. Methods information.
4. Variable information.
5. Constructor information
6. Modifier information.
7. Constant pool info. Etc.

After loading .class file immediately JVM creates an object (creates a class class object to rep student.class) for that loaded class in the heap memory of type java.lang.Class.

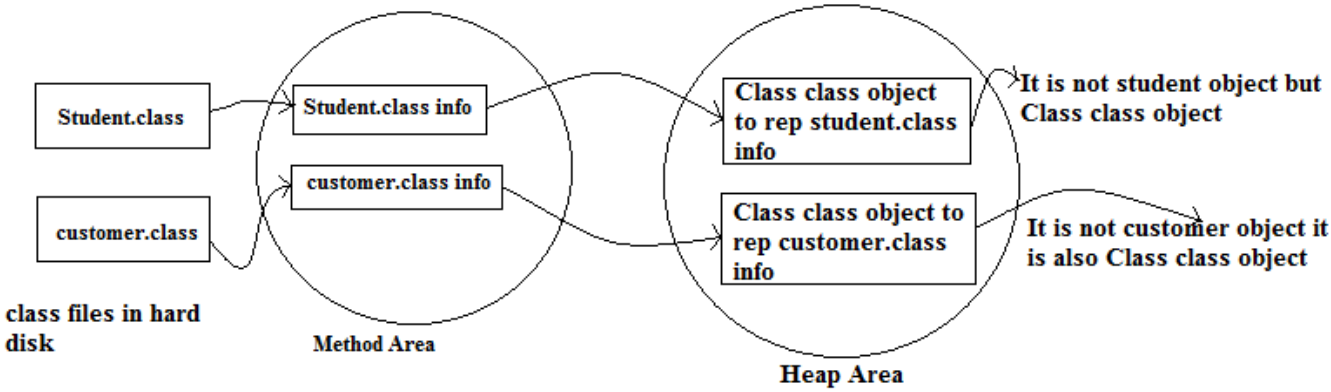


Fig 2: Class Loading

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

Example:

```
import java.lang.reflect.*;
class student
{
	public String getname()
	{
		return null;
	}
	public int getRollNo()
	{
		return 10;
	}
}
class jv
{
	public static void main(String [] args)throws Exception
	{
		int count=0;
		Class c=Class.forName("student");
		Method[] m=c.getDeclaredMethods();
		for (Method m1:m)
		{
			count++;
```

```
                System.out.println(m1.getName());

            }
            System.out.println(count);
        }
}
```

For every loaded type only one Class object will be created even though we are using the class multiple times in our program.

2. Linking:

Linking consists of three activities:

1. Verify.
2. Prepare
3. Resolve.

Verify/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, 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

Java.lang.VerifyError

Prepare/preparation:

In this phase JVM will allocate memory for class level static variable and assign default values.

Note: In initialization phase original values will be assigned to the static variable and here only default values will be assigned

Resolve/Resolution:

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

```
class resolve
{
        public static void main(String ...s)
        {
            String s=new String ("cse");
            Student s1=new Student("Ram");
        }
}
```

For the above class class-loader loads resolve.class, Student.class, String.class, Object.class. The names of these classes are stored in constant pool of resolve class. In resolution phase these names are replaced with original memory level references from method area.

3. Initialization:

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

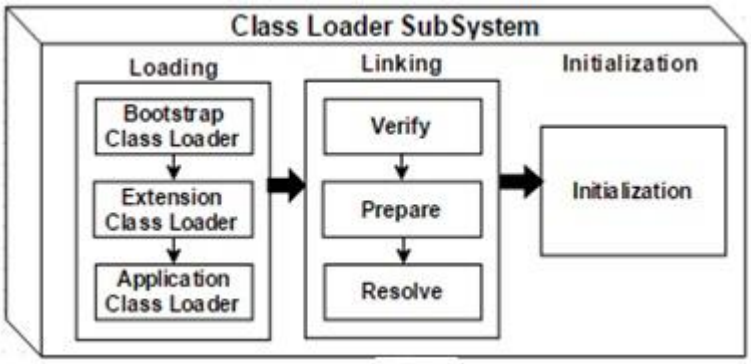


Fig 3: Class loader subsystem

While loading, linking and initialization if any error occurs then, we will get runtime exception **Java.lang.LinkageError**

Note: Java.lang.VerifyError. Here VerifyError is a child class of LinkageError

Types of class Loaders:

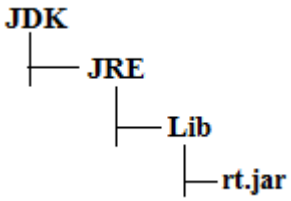
Class loader subsystem contain following three types of class loader:

1. BootStarp class loader/Primordial class loader.
2. Extension class loader.
3. Application/System class loader.

BootStrap class loader:

It is responsible to load core java API classes. These classes (API classes) are present in rt.jar. and the location/path of rt.jar is called bootstrap class path ie., bootstrap class loader is responsible to load classes from bootstrap class path.

Bootstrat class loader is by default available with every JVM. It is implemented in native languages like C, C++ and not in java.



BootStrap Class Path: Jdk\jre\lib\rt.jar

Fig 4: Bootstrap Class Path

Extension Class Loader:

Extension class loader is a child class of bootstrap class loader and is responsible for loading classes from extension class path (jdl\jre\lib\ext).

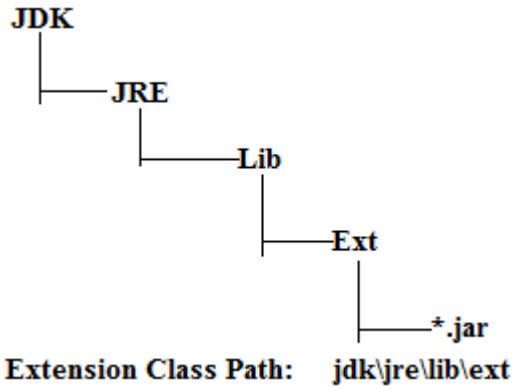


Fig 5: Extension Class Path

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

sun.misc.Launcher$ExtClassLoader.class

Application/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 class path (path we set in system settings). Application loader is implemented in java and the corresponding .class file is.

Sun.misc.Launcher$AppClassLoader.class

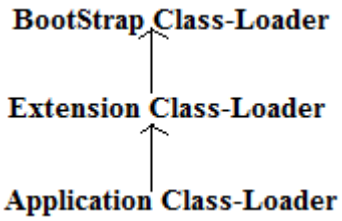


Fig 6: Class Loaders

How class loader Work:
Class loader follows delegation hierarchy principle or 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 method area then JVM will consider that loaded class.

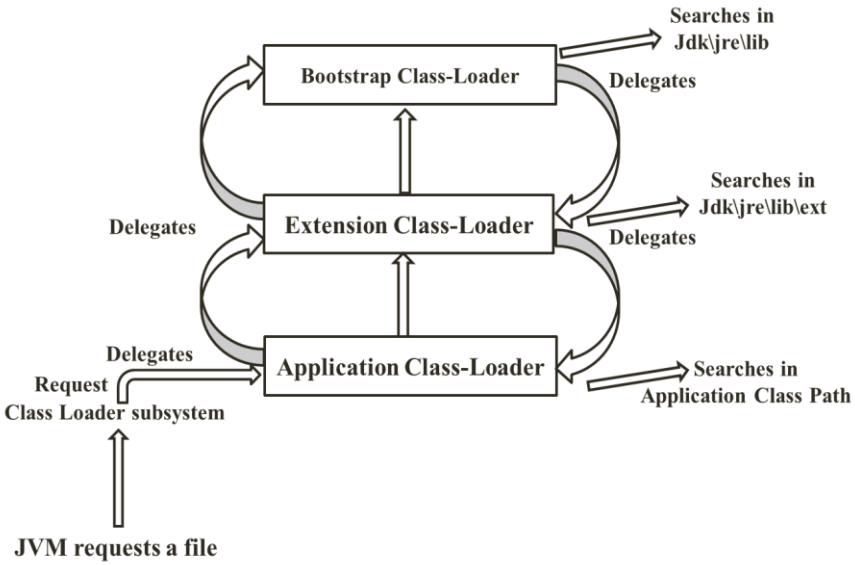


Fig 7: Delegation Hierarchy

If it is not loaded then JVM requests class loader subsystem to load that class. Then class loader subsystem handovers the request to application class loader. Application class loader delegates the request to extension class loader which in turn delegates the request to bootstrap class loader. Then, bootstrap class loader will search in bootstrap class path if it is available then class will be loaded by bootstrap class loader and if not then bootstrap class loader delegates the request to extension class loader. Extension class loader will search in extension class path if it is available then it will be loaded otherwise extension class loader delegates the request to application class loader. Application class loader will search application class path if it available then it will be loaded otherwise runtime exception will be there.

NoClassDefFoundError

Or

ClassNotFoundException

Example:

```
class re
{
        public static void main(String ...s)
        {
                System.out.println(String.class.getClassLoader());
                System.out.println(re.class.getClassLoader());
        }
}
```

Runtime memory Areas of JVM

Whenever JVM loads and run java program it needs memory to store several things like bytecode, objects, variable etc. total JVM memory is organized into following five categories.

1. Method area.
2. Heap area.
3. Stack area.
4. PC registers.
5. Native method stacks.

Method Area:

For every JVM one method area is available and is created at the time of JVM startup. Inside method area class level binary data including static variables will be stored. Constant pools of a class will be stored inside method area. Method area can be accessed by multiple threads simultaneously

Heap Area:

For very JVM one heap area is available. Heap area is created at the time of JVM startup. Objects and corresponding instance variables are stored in the heap area. Every area in java is object only hence, array is also stored in the heap area. Heap area can be accessed by multiple threads and hence the data stored in the heap memory is not thread safe. Heap area need not be contiguous.

Heap memory statistics:

Java application can communicate with JVM by using runtime object. Runtime class (Present in java.lang package) is singleton class and we can create Runtime object as follows:

Runtime r=Runtime.getRuntime();

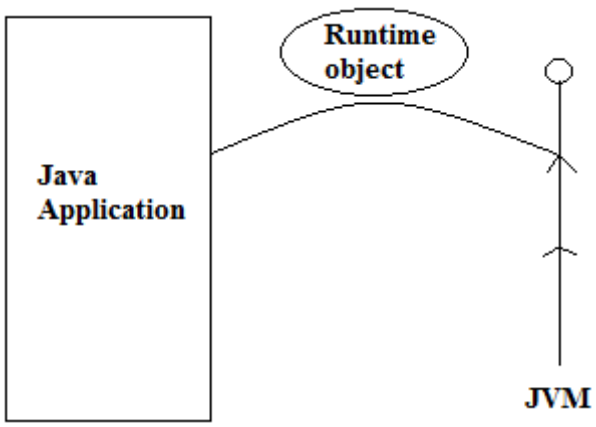


Fig 8: Runtime communication

Once we get Runtime object we can call following methods on that object

1. **maxMemory( ):**
   It returns the no bytes of maxMemory allocated to the heap.

2. **totalMemory( ):**

It returns no of bytes of total memory allocated to the heap (Initial Memory).

3. **freeMemory():**
   It returns no of bytes of free memory present in the heap.

Example:

```
class heap
{
	public static void main(String ...s)
	{
		long 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);

	}
}
```

Setting Min and Max Heap size:
Heap memory is finite/fixed memory but based upon our requirement we can set max and min heap size that is, we can increase or decrease sixe based on our requirement. We can use following flags with java command.

-Xmx to set max heap size (Max memory)
Ex: C:\java> Java –Xmx512m heap

This command will set max heap size as 512 MB.

-Xms to set min heap size
Ex: C:\java> Java –Xms64m heap

To set min heap size as 64MB (total Memory)

Ex: C:\java>Java –Xmx512m –Xms64m heap

Stack Memory Area:
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 corresponding stack including variables. After completing a method the corresponding entry from the stack will be removed. After completing all method calls the stack will become empty and that empty stack will be destroyed by the JVM just before terminating the thread. Each entry in the stack is called stack frame or activation record.

Example:

```
class stack
{
	public static void main(String ...s)
	{
		System.out.println("main calling m1");
		m1();
	}
	public static void m1()
	{
		System.out.println("m1 calling m2");
```

```
        m2();
    }
    public static void m2()
    {
        System.out.println("this is m2");

    }

}
```

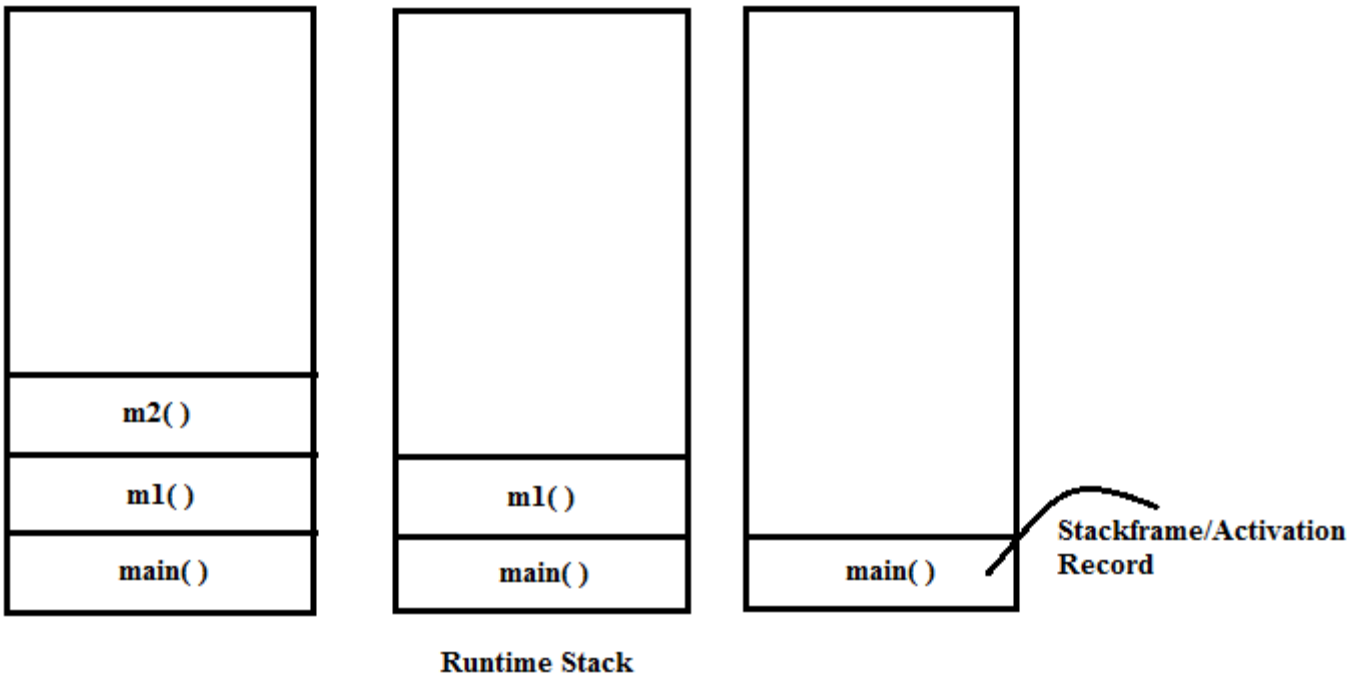


Fig 9: Runtime stack

The data stored in the stack is available for the corresponding thread only and not available to the remaining threads therefore this data is thread safe.

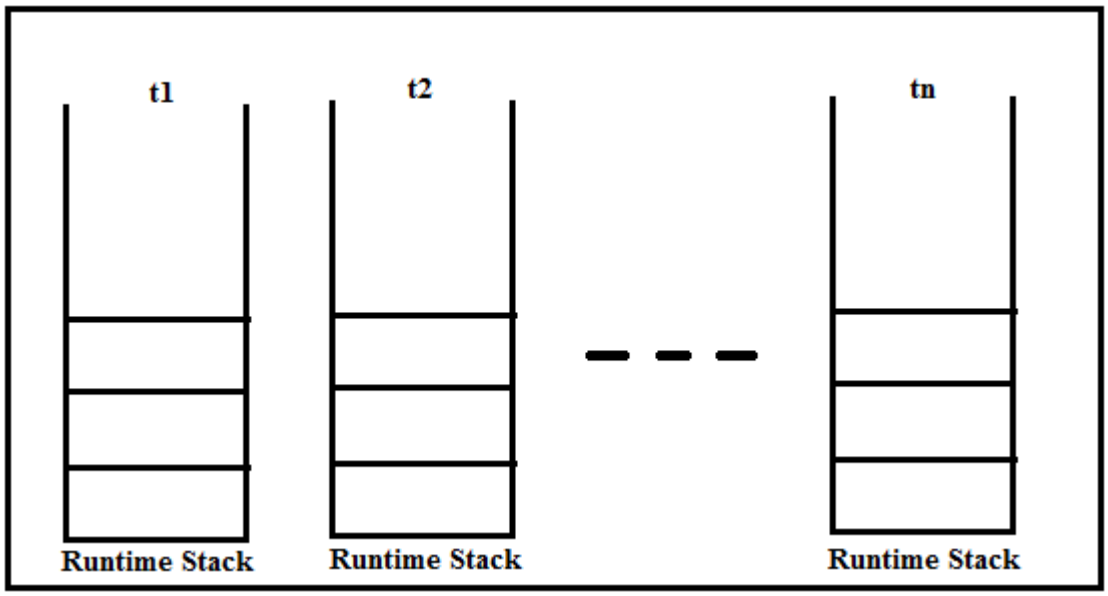


Fig 10: Runtime Stack per thread

Stack frame structure:

Each stack frame contains three parts

1. Local Variable array
2. Operand Stack.

3. Frame Data.

Local Variable Array:

```
Public void m(int i, double d, Object o, float f)
{
        long x;
}
```

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. Values of double and long occupy two conse-

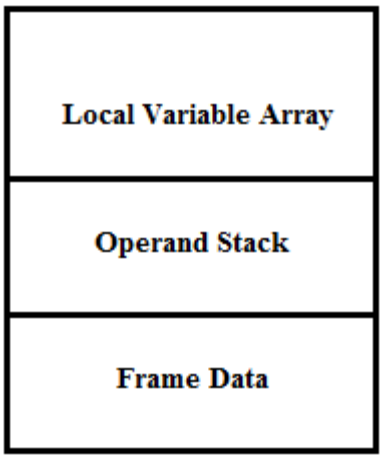


Fig 11: Stack Frame

cutive entries in the array. Byte, short and char values will be converted to int type before storing and occupying one slot each. But, the way storing Boolean values varied from JVM to JVM. But most of the JVM follow one slot for Boolean values.

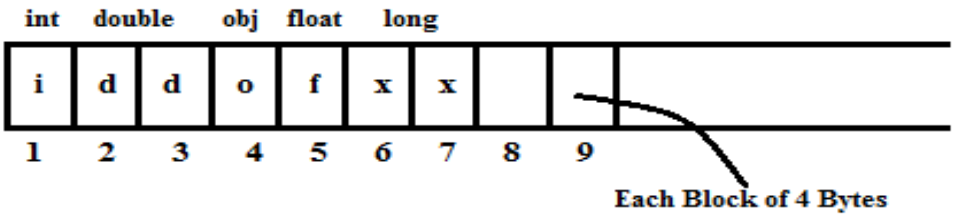


Fig 12: Local variable Array

Operand Stack:

JVM uses operand stack as workspace. Some instructions can push values to operand stack and some instructions can pop values from operand stack and some instructions can perform required operations.

Example:

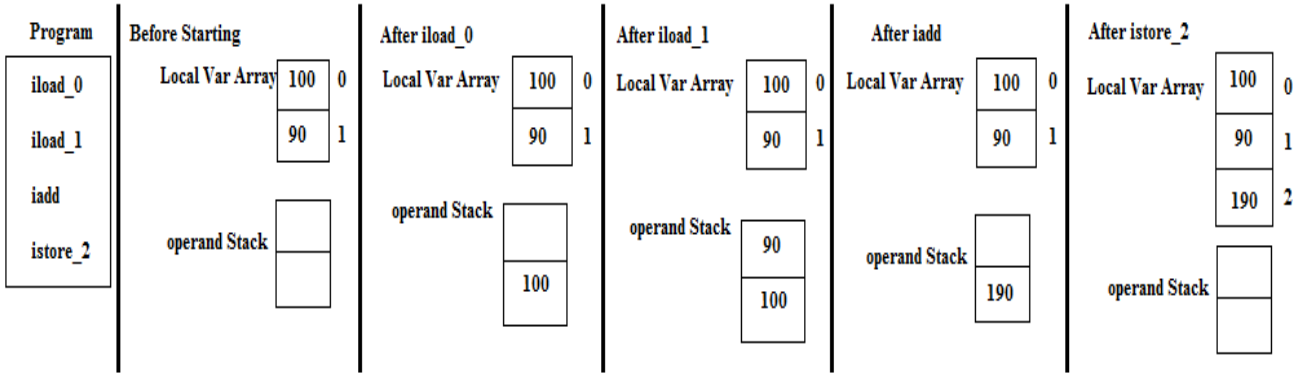


Fig 13: Operand Stack

Frame Data:
Frame data contains all symbolic references related to that method. It also contains a reference to exception table which provides corresponding catch block information in the case of exceptions.

Program Counter (PC) Registers:
For every thread a separate pc register will be created at the time of thread creation. PC registers contains the address of current executing instruction. Once instruction execution completes automatically pc register will be incremented to hold address of next instruction.

Native Method Stack:
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.

Conclusions:
Method area, heap area and stack area are considered as important memory areas with respect to programmer. Method area and heap area are per JVM whereas stack area, pc register and native method stack are per thread.

Static variable	**stored in**	**Method Area**
Instance Variable	**stored in**	**Heap Area.**
Local Variable	**stored in**	**Stack Area.**

Execution engine:
This is the central component of JVM. Execution engine is responsible to execute java class files. Execution engine mainly contains two components.

1. **Interpreter:**
   It is responsible to read byte code and interpret into machine code (native code) and execute that machine code line by line. The problem with interpreter is it interpret every time even same method invoked multiple times, which reduces performance of the system. To overcome this problem JIT compiler was introduced in version 1.1.
2. **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 increments the corresponding count variable.

   This process will be continued for every method. Once, if any method count reaches threshold value then JIT compiler identifies that, that method is a repeatedly used method (Hotspot). So, immediately JIT compiler compiles that method and generates the corresponding native code. Next time JVM across that method call then JVM uses native code directly and executes it instead of interpreting once again so, that performance of the system will be improved. The threshold count varies from JVM to JVM.

   Some advanced JIT compiler will recompile generated native code if count reaches threshold value second time so that more optimized machine code will be generated. Internally profiler (part of JIT compiler) is responsible to identify hotspots.

Note:
JVM interpret total program at least once
JIT compilation is applicable only for repeated required methods not for every method.

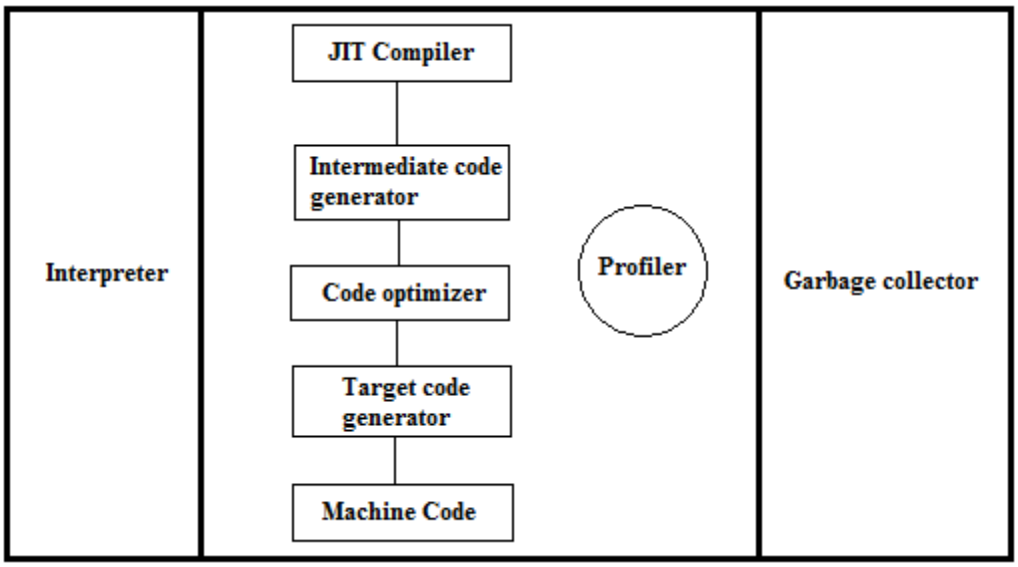


Fig 14: Execution Engine

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.

Native method library
Native method library hold native library information.

Garbage Collection

In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **OutOfMemoryErrors**.

But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.

Garbage collector is best example of Daemon thread as it is always running in background.

Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects (**An object is said to be unreachable if it doesn't contain any reference to it.). And an object is said to be eligible for GC(garbage collection) if it is unreachable.

The ways to make an object eligible for garbage collection
Even though programmer is not responsible to destroy use less objects it is recommended to make an object eligible for GC if it is no longer required. An object is said to be eligible for GC if and only if doesn't contain any reference variable.
The following are various ways to make an object eligible for GC

1. **Nullifying the reference variable**
   When object doesn't have any ref variable then object is eligible for GC
   Student s1=new Student ();
   Student s2=new Student ();
   To make object eligible for GC initialize its ref variable equal to null
   s1=null; // first object is useless now and eligible for GC

2. **Reassigning the reference variable.**

Student s1=new Student ();
Student s2=new Student ();
s1=new student (); //s1 is now pointing to new object so previous object pointed by reference variable is eligible for GC
s2=s1

3. **Objects created inside a method**

Objects created inside method are eligible for GC once method get completed.

```
Class test
{
        PSVM(String [] args)
        {
                m1();
        }
        PSV m1()
        {
                Student s1=new Student ();
                Student s2=new Student ();

        }
}
```

Here s1 and s2 are local variable of m1(). So, after completion of method they will get destroyed and both objects will be eligible for GC

```
Class test
{
        PSVM(String [] args)
        {
                Student s=m1(); // One object for GC
        }
        PS Student m1()
        {
                Student s1=new Student ();
                Student s2=new Student ();
                Return s1
        }
}
```

Two objects s1, s2 are created in function m1() and s1 is returned. For returned object there is ref variable in function m1() that is object s1 will not be eligible for GC. Whereas, s2 variable will destroyed and corresponding object will be eligible for GC.

```
Class test
{
        PSVM(String [] args)
        {
                m1(); //two objects for GC
        }
        PS Student m1()
        {
                Student s1=new Student ();
                Student s2=new Student ();
                Return s1
        }
}
```

In this example function is returning object but there is no reference variable in function m1() to capture that. Therefore, both variables will be destroyed after completion of m1() and both objects will be eligible for GC.

```
Class test
{

        Static Student s;
        PSVM(String [] args)
        {
                m1(); //one objects for GC
        }
        PSV m1()
        {
                s=new Student ();
                Student s1=new Student ();
        }
}
```

Here we have one static variable of class (student) type. Now in m1() s is pointing to one object and there is also one local variable pointing to second object. Now after completion of m1() there will be only one object eligible for GC because local will be destroyed but not static variable.

4. **Island of isolation**

```
Class test
{
        test i;
        PSVM(String [] args)
        {
                test t1=new test();
                test t2=new test();
                test t3=new test();
                t1.i=t2; // i pointing to t2 (i is of test type and
                                instance variable of t1)
                t2.i=t3;
                t3.i=t1;
                t1=NULL; // but object is accessible using t3.i
                t2=NULL; // object can be accessed using t3.i.i
                t3=NULL; // now all three object eligible for GC
        }
}
```

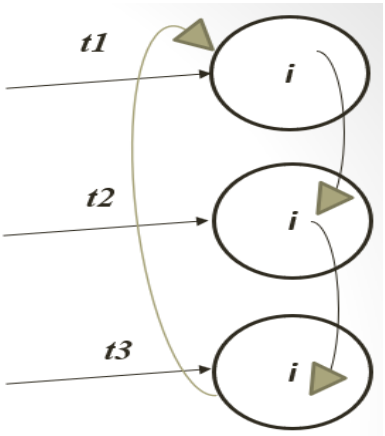


Fig 15: Island of isolation

In this example i is instance variable of type test. In main when we create object then for every object instance variable will be created.

Once object is made eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we cannot expect. But we can request JVM to run Garbage Collector. There are two ways to do it:

1. **Using *System.gc()* method** : System class contain static method *gc()* for requesting JVM to run Garbage Collector.
2. **Using *Runtime.getRuntime().gc()* method** : Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

But there is no guarantee that any one of above two methods will definitely run Garbage Collector. So, Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform cleanup activities. Once *finalize()* method completes, Garbage Collector destroys that object.

Class file structure

The Java class file contains everything a JVM needs to know about one Java class or interface. In their order of appearance in the class file, the major components are: magic, version, constant pool, access flags, this class, super class, interfaces, fields, methods, and attributes.

Information stored in the class file often varies in length -- that is, the actual length of the information cannot be predicted before loading the class file. For instance, the number of methods listed in the methods component can differ among class files, because it depends on the number of methods defined in the source code. Such information is organized in the class file by prefacing the actual information by its size or length. This way, when the class is being loaded by the JVM, the size of variable-length information is read first. Once the JVM knows the size, it can correctly read in the actual information.

The order of class file components is strictly defined so JVMs can know what to expect, and where to expect it, when loading a class file. For example, every JVM knows that the first eight bytes of a class file contain the magic and version numbers, that the constant pool starts on the ninth byte, and that the access flags follow the constant pool. But because the constant pool is variable-length, it doesn't know the exact where abouts of the access flags until it has finished reading in the constant pool. Once it has finished reading in the constant pool, it knows the next two bytes will be the access flags.

Java Class File structure

Aclass file consists of a singleClassFile structure:

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

Magic Number

The first 4 bytes of class file are termed as magic_number. This is a predefined value which the JVM use to identify whether the *.class* file is generated by valid compiler or not. The predefined value will be in hexadecimal form i.e. **0xCAFEBABE**.

Whenever we are executing a Java Class if JVM Unable to Find Valid Magic Number then we get RuntimeException

ClassFormatError: incompatible magic value.

Version
The next four byte of the class file contains major and minor version numbers. This number allows the JVM to verify and identify the class file. Minor and Major Versions Represents Class File Version. JVM will Use these Versions to Identify which Version of Compiler Generates Current .class File

M.m denotes the version of class file where M stands for major_version and m stands for minor_version. Higher Version JVM can Always Run Lower Version Class Files but Lower Version JVM can't Run Class Files generated by Higher Version Compiler.

Whenever we are trying to Execute Higher Version Compiler generated Class File with Lower Version JVM we will get RuntimeException

java.lang.UnsupportedClassVersionError: Employee (Unsupported major.minor version 51.0)

Constant_pool_count
The value of the constant_pool_count item is equal to the number of entries in the constant_pool table plus one. A constant_pool index is considered valid if it is greater than zero and less than constant_pool_count, with the exception for constants of type long and double.

constant_pool[]
It represents the information about constants present in constant pool file. The constant_pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant_pool table entry is indicated by its first "tag" byte. The constant_pool table is indexed from 1 to constant_pool_count-1.

Access flags
Access flags follows the Constant Pool. It is a two byte entry that indicates whether the file defines a class or an interface, whether it is public or abstract or final in case it is a class. Below is a list of some of the access flags and their interpretation.

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared final; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared abstract; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.

this Class
The value of the this_class item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info. structure representing the class or interface defined by this class file. this Class is a two byte entry that points to an index in Constant Pool. In Fig 16., this class has a value 0x0007 which is an index in Constant Pool.

super Class
Next 2 bytes after This Class is of Super Class. Similar to this class, value of two bytes is a pointer that points to Constant pool which has entry for super class of the class.

interfaces_count

The value of the interfaces_count item gives the number of direct super interfaces of this class or interface type

Interfaces []
Each value in the interfaces array must be a valid index into the constant_pool table.

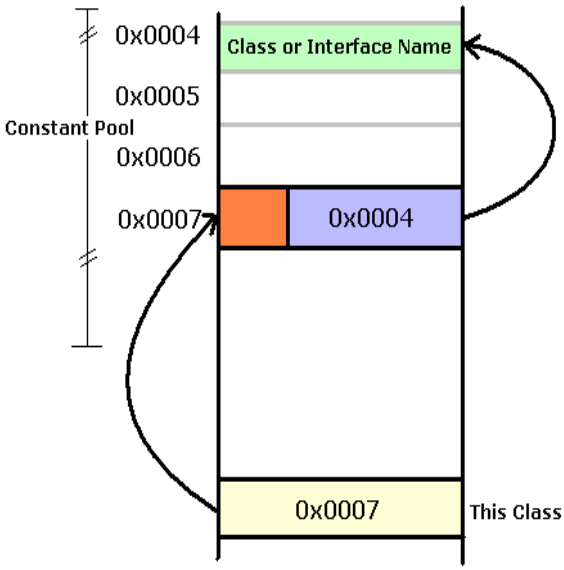


Fig 16: Constant pool table

fields_count
The value of the fields_count item gives the number of field_info structures in the fields table. The field_info structures represent all fields, both class variables and instance variables, declared by this class or interface type.

Fields []
Each value in the fields table must be a field_info. The fields table includes only those fields that are declared by this class or interface. It does not include items representing fields that are inherited from superclasses or super interfaces

methods_count
The value of the methods_count item gives the number of method_info structures in the methods table

methods []
Each value in the methods table must be a method_info giving a complete description of a method in this class or interface. If the method is not native or abstract, the Java virtual machine instructions implementing the method are also supplied

attributes_count
The value of the attributes_count item gives the number of attributes in the attributes table of this class.

attributes []
Each value of the attributes table must be an attribute structure

Java security

- **JVM has Security Manager to provide explicit security for the Java applications**
- **Java has predefined library in the form of java.Security.package.**

Java security includes two aspects

- Java platform is a secure, ready-built platform to run Java-enabled applications in a secure fashion.
- Provide security tools and services implemented in the Java programming language that enable a wider range of security-sensitive applications.

The Original Sandbox Model:

The original security model provided by the Java platform is known as the sandbox model, which existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network. The essence of the sandbox model (limited and controlled execution environment) is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code (an applet) is not trusted and can access only the limited resources provided inside the sandbox. This sandbox model is illustrated in the figure below.

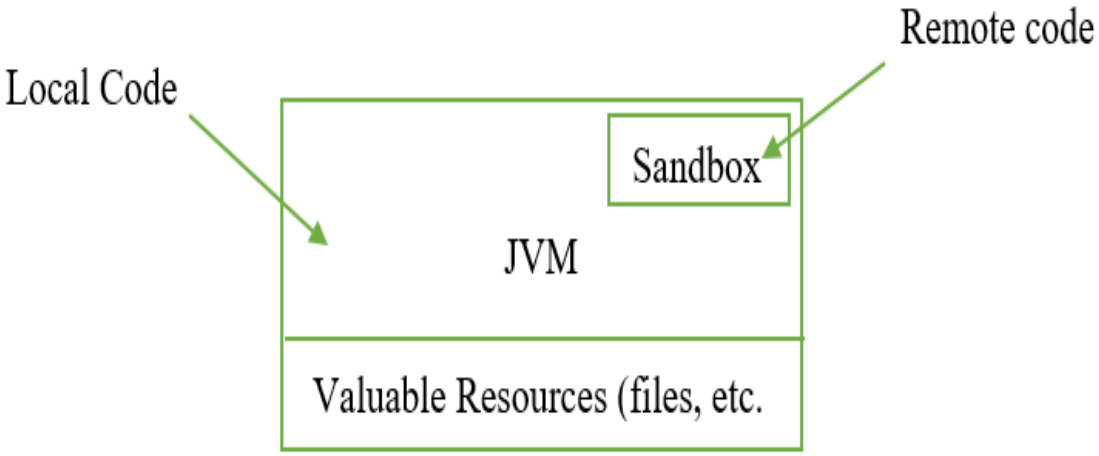


Fig 17: Sandbox Model

The sandbox model was deployed through the Java Development Kit (JDK), and was generally adopted by applications built with JDK 1.0, including Java-enabled web browsers.
Overall security is enforced through a number of mechanisms. First of all, the language is designed to be type-safe and easy to use. Language features such as automatic memory management, garbage collection, and range checking on strings and arrays are examples of how the language helps the programmer to write safe code.

Second, compilers and a bytecode verifier ensure that only legitimate Java bytecodes are executed. The bytecode verifier, together with the Java Virtual Machine, guarantees language safety at run time.

Moreover, a class loader defines a local name space, which can be used to ensure that an untrusted applet cannot interfere with the running of other programs.

Finally, access to crucial system resources is mediated by the Java Virtual Machine and is checked in advance by a SecurityManager class that restricts the actions of a piece of untrusted code to the bare minimum.

JDK 1.1 introduced the concept of a "signed applet", as illustrated by the figure below. In that release, a correctly digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet. Signed applets, together with their signatures, are delivered in the JAR (Java Archive) format. In JDK 1.1, unsigned applets still run in the sandbox.

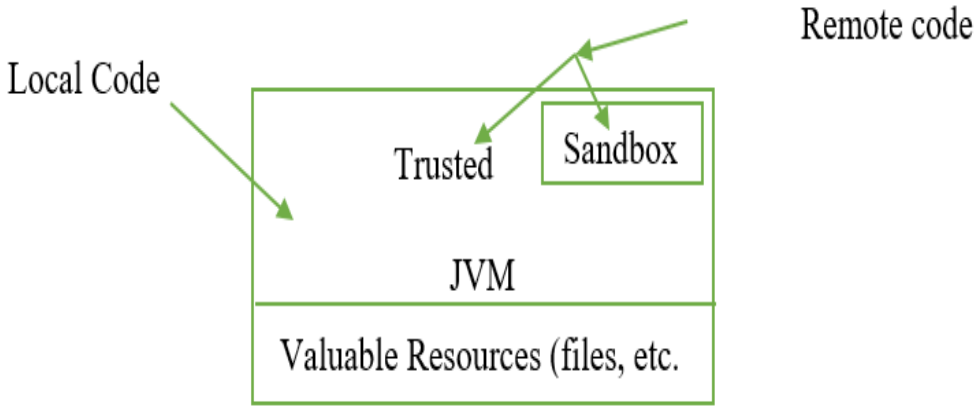


Fig: JDK 1.1 Sandbox Model

Evolving the Sandbox Model

The new Java SE Platform Security Architecture, illustrated in the figure below, is introduced primarily for the following purposes.

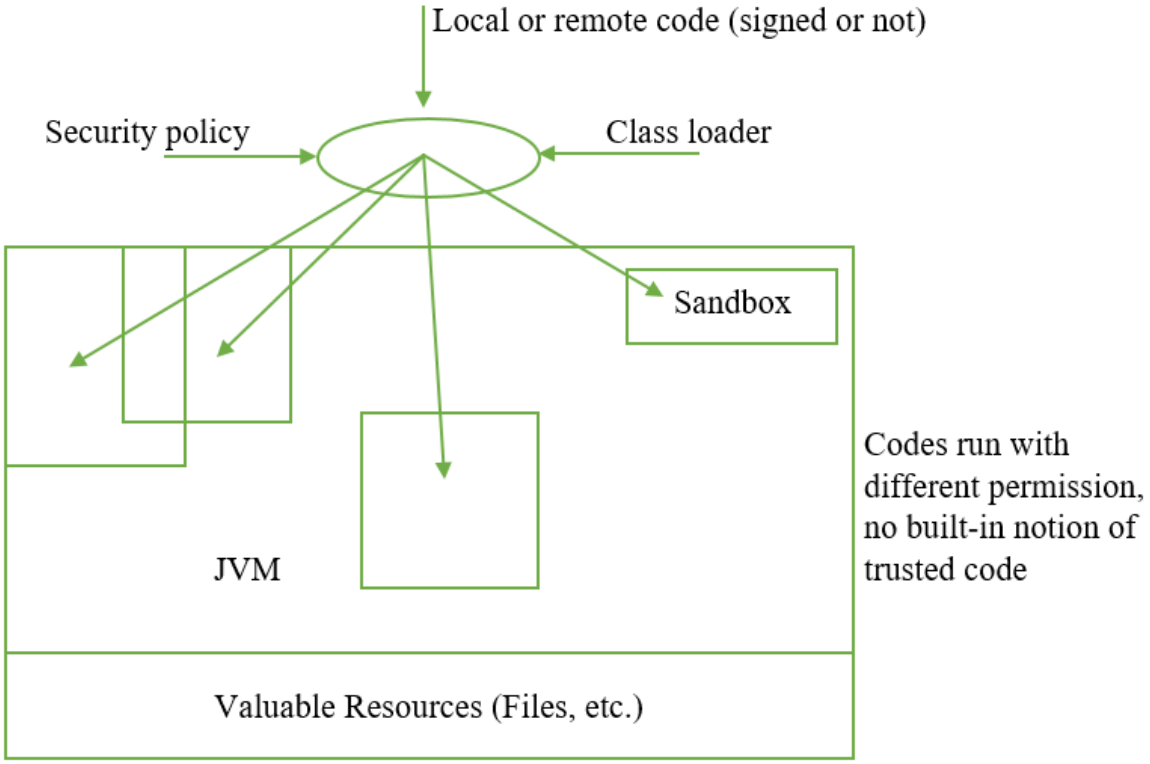


Fig 19: Java SE Platform Security Architecture

Fine-grained access control.

This capability existed in the JDK from the beginning, but to use it, the application writer had to do substantial programming (e.g., by subclassing and customizing the SecurityManager and ClassLoader classes). However, such programming is extremely security-sensitive and requires sophisticated skills and in-depth knowledge of computer security. The new architecture will make this exercise simpler and safer.

Easily configurable security policy.

Once again, this capability existed previously in the JDK but was not easy to use. Moreover, writing security code is not straightforward, so it is desirable to allow application builders and users to configure security policies without having to program.

Easily extensible access control structure.

Up to JDK 1.1, in order to create a new access permission, you had to add a new check method to the SecurityManager class. The new architecture allows typed permissions (each representing an access to a system resource) and automatic handling of all permissions (including yet-to-be-defined permissions) of

the correct type. No new method in the SecurityManager class needs to be created in most cases. (In fact, we have so far not encountered a situation where a new method must be created.)

Extension of security checks to all Java programs, including applications as well as applets.
There is no longer a built-in concept that all local code is trusted. Instead, local code (e.g., non-system code, application packages installed on the local file system) is subjected to the same security control as applets, although it is possible, if desired, to declare that the policy on local code (or remote code) be the most liberal, thus enabling such code to effectively run as totally trusted. The same principle applies to signed applets and any Java application.

Finally, an implicit goal is to make internal adjustment to the design of security classes (including the SecurityManager and ClassLoader classes) to reduce the risks of creating subtle security holes in future programming.