A Report on

JAVA VIRTUAL MACHINE

Submitted by: Bhuvan.M.S 12IT16

Kusum Vanwani 12IT37

Madhavi Srinivasan 12IT39

Siddharth Jain 12IT78

Vinay Rao 12IT94

Table Of Contents

JVM Architecture

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java byte-code can be executed. JVM is:

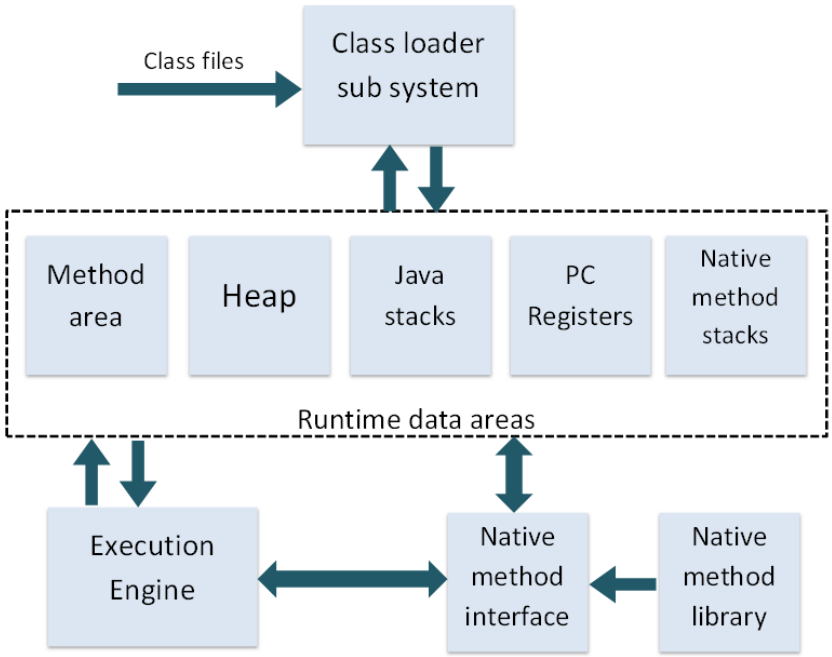
1. **A specification:** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
2. **An implementation:** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance:** Whenever you write java command on the command prompt to run the java class, and instance of JVM is created.

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Exception Handling
- Garbage-collected heap



Method Area

Inside a Java virtual machine instance, information about loaded types is stored in a logical area of memory called the method area. When the Java virtual machine loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area.

Features:

- Thread safe
  All threads share the same method area, so access to the method area's data structures must be designed to be thread-safe. If two threads are attempting to find a class named A, for example, and A has not yet been loaded, only one thread should be allowed to load it while the other one waits.

- Variable Size
  The size of the method area need not be fixed. As the Java application runs, the virtual machine can expand and contract the method area to fit the application's needs. Also, the memory of the method area need not be contiguous. It could be allocated on a heap--even on the virtual machine's own heap. Implementations may allow users or programmers to specify an initial size for the method area, as well as a maximum or minimum size.

- Support for Garbage collection
  The method area can also be garbage collected. Because Java programs can be dynamically extended via user-defined class loaders, classes can become "unreferenced" by the application. If a class becomes unreferenced, a Java virtual machine can unload the class (garbage collect it) to keep the memory occupied by the method area at a minimum.

- Stores type information
  For each type it loads, a Java virtual machine must store the following kinds of information in the method area:

  The fully qualified name of the type
  The fully qualified name of the type's direct superclass (unless the type is an interface or class java.lang.Object, neither of which have a superclass)
  Whether or not the type is a class or an interface
  The type's modifiers (some subset of` public, abstract, final)
  An ordered list of the fully qualified names of any direct superinterfaces

An Example of Method Area Use

As an example of how the Java virtual machine uses the information it stores in the method area, consider these classes:

```
// On CD-ROM in file jvm/ex2/Lava.java
class Lava {

   private int speed = 5; // 5 kilometers per hour

   void flow() {
   }
}

// On CD-ROM in file jvm/ex2/Volcano.java
class Volcano {

   public static void main(String[] args) {
      Lava lava = new Lava();
      lava.flow();
   }
}
```

To run the Volcano application, you give the name "Volcano" to a Java virtual machine in an implementation-dependent manner. Given the name Volcano, the virtual machine finds and reads in file Volcano.class. It extracts the definition of class Volcano from the binary data in the imported class file and places the information into the method area. The virtual machine then invokes the main() method, by interpreting the bytecodes stored in the method area. As the virtual machine executes main(), it maintains a pointer to the constant pool (a data structure in the method area) for the current class (class Volcano).

Note that this Java virtual machine has already begun to execute the bytecodes for main() in class Volcano even though it hasn't yet loaded class Lava. Like many (probably most) implementations of the Java virtual machine, this implementation doesn't wait until all classes used by the application are loaded before it begins executing main(). It loads classes only as it needs them.

main()'s first instruction tells the Java virtual machine to allocate enough memory for the class listed in constant pool entry one. The virtual machine uses its pointer into Volcano's constant pool to look up entry one and finds a symbolic reference to class Lava. It checks the method area to see if Lava has already been loaded.

The symbolic reference is just a string giving the class's fully qualified name: "Lava". Here you can see that the method area must be organized so a class can be located--as quickly as possible--given only the class's fully qualified name. Implementation designers can choose whatever algorithm and data structures best fit their needs--a hash table, a search tree, anything. This same mechanism can be used by the static forName() method of class Class, which returns a Class reference given a fully qualified name.

When the virtual machine discovers that it hasn't yet loaded a class named "Lava," it proceeds to find and read in file Lava.class. It extracts the definition of class Lava from the imported binary data and places the information into the method area.

The Java virtual machine then replaces the symbolic reference in Volcano's constant pool entry one, which is just the string "Lava", with a pointer to the class data for Lava. If the virtual machine ever has to use Volcano's constant pool entry one again, it won't have to go through the relatively slow process of searching through the method area for class Lava given only a symbolic reference, the string "Lava". It can just use the pointer to more quickly access the class data for Lava. This process of replacing symbolic references with direct references (in this case, a native pointer) is called constant pool resolution.

Class Loader

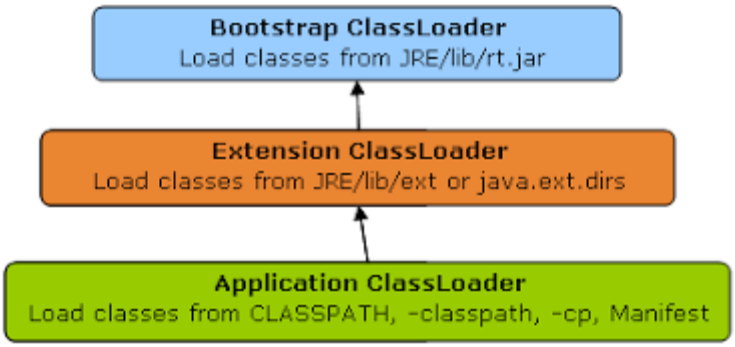
Class loading is done by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or lazy load the class until a need of class initialization occurs. If Class is loaded before it is actually being used it can sit inside before being initialized.This may vary from JVM to JVM.

ClassLoader in Java is a class which is used to load class files in Java. Java code is compiled into class file by javac compiler and JVM executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source. There are three default class loader used in Java, Bootstrap or Primordial , Extension and System or Application class loader. Every class loader has a predefined location, from where they loads class files.

1) Bootstrap ClassLoader - JRE/lib/rt.jar

2) Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs

3) Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.



Java class loaders are used to load classes at runtime. ClassLoader in Java works on three principle: delegation, visibility and uniqueness.

Delegation principle

Delegation principle forward request of class loading to parent class loader and only loads the class, if parent is not able to find or load class.

Suppose we have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from Classpath in Java.

Visibility Principle

Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader can not see classes loaded by child.

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which means if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either java.lang.ClassNotFoundException. as shown in below Example

```
package test;

import java.util.logging.Level;
```

```
import java.util.logging.Logger;

public class ClassLoaderTest {

   public static void main(String args[]) {

      try {

         //printing ClassLoader of this class

         System.out.println("ClassLoaderTest.getClass().getClassLoader() : "

                           + ClassLoaderTest.class.getClassLoader());

         //trying to explicitly load this class again using Extension class loader

         Class.forName("test.ClassLoaderTest", true

                        ,  ClassLoaderTest.class.getClassLoader().getParent());

      } catch (ClassNotFoundException ex) {

         Logger.getLogger(ClassLoaderTest.class.getName()).log(Level.SEVERE, null, ex);

      }

   }

}
```

Output:

```
ClassLoaderTest.getClass().getClassLoader() :
sun.misc.Launcher$AppClassLoader@601bb1

16/08/2012 2:43:48 AM test.ClassLoaderTest main

SEVERE: null

java.lang.ClassNotFoundException: test.ClassLoaderTest
```

Uniqueness Principle

Uniqueness principle allows to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent.

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial.

Garbage Collection

One of the benefits JAVA provides to the users is the benefit of garbage collection. This is basically automatic memory management. We know that in JAVA we can allocate memory in heap for an object using the "new" keyword. When an object is no longer referenced by the program, the heap space it occupies can be recycled so that the space is made available for subsequent new objects. Instead of doing it manually, JAVA does this job for us. **Garbage Collection in Java** is carried by a daemon thread called ***Garbage Collector***. This thread invokes the finalize method of the object before removing it to perform any last clean up actions if required. So basically, this is reclaiming the unused memory in the heap. We cannot force garbage collection in JAVA. It will trigger only if JVM wants depending upon the heap size. Now the question comes – Why automatic memory management? The answer is simple! It makes programs simpler; Removes need for explicit declaration, prevents memory leaks, enables proper encapsulation and prevents the problem of dangling pointers.

How Garbage Collection works

The general misconception behind garbage collector is that it finds the dead objects and deallocates their memory. But what actually happens is that it actually tracks only live objects and rest all is declared as dead! All objects are allocated on the heap area managed by the JVM. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory. Every object tree has one or more root objects. Special objects called garbage-collection roots are always reachable and so is any object that has a garbage-collection root at its own root.To determine which objects are no longer in use, the JVM intermittently runs what is very aptly called a mark-and-sweep algorithm. As you might intuit, it's a straightforward, two-step process:

1. The algorithm traverses all object references, starting with the GC roots and marks every found object as alive.

2. All of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept free of unused objects.

Generally if the number of live objects is large when garbage collector operates, garbage collection cycle can take a lot of time and hence the application execution time increased. This removal leads to fragmented memory which also slows down the process. To remove this, JVM performs compaction. But the downside is an even longer garbage collection cycle, and since most JVMs suspend the application execution during compaction, the performance impact can be considerable.

Reducing Garbage-Collection Pause Time

Generally two ways to reduce garbage collection –

1. Execute garbage collection parallel, using multiple CPUs.Application is still suspended from execution.
2. Running it concurrently with the application

This led to development of 3 garbage collection strategies – serial, parallel and concurrent. The serial collector suspends the application and executes the mark-and-sweep algorithm in a single thread. It is the simplest and oldest form of garbage collection in Java and is still the default in the Oracle HotSpot JVM. The parallel collector uses multiple threads to do its work. It can therefore decrease the GC pause time by leveraging multiple CPUs. It is often the best choice for throughput applications. The concurrent collector does the majority of its work concurrent with the application execution. It has to suspend the application for only very short amounts of time. This has a big benefit for response-time–sensitive applications, but is not without drawbacks.

Making Garbage Collector Faster

This can be done only if fewer objects are to be marked.

The Generation Conflict – Young vs. Old

In a typical application most objects are very short-lived. On the other hand, some objects last for a very long time and even until the application is terminated. When using generational garbage collection, the heap area is divided into two areas—a young generation and an old generation—that are garbage-collected via separate strategies.

Objects are usually created in the young area. Once an object has survived a couple of GC cycles it is tenured to the old generation. After the application has completed its initial start-up phase (most applications allocate caches, pools, and other permanent objects during start-up), most allocated objects will not survive their first or second GC cycle. The number of live objects that need to be considered in each cycle should be stable and relatively small. Allocations in the old generation should be infrequent, and in an ideal world would not happen at all after the initial start-up phase. If the old generation is not growing and therefore not running out of space, it requires no garbage-collection at all. There will be unreachable objects in the old generation, but as long as the memory is not needed, there is no reason to reclaim it. One of the disadvantages is that there is a lot of allocation and deallocation in young generation leading to fragmentation. To resolve this, JVM uses a strategy called copy collection. Copy garbage collection divides the heap into two (or more) areas, only one of which is used for allocations. When this area is full, all live objects are copied to the second area, and then the first area is simply declared empty. This avoids fragmentation completely but is effective only if most objects die. The whole idea of generational heaps also poses

some serious threats to the execution time if the size of young generation is too small or too big.

Garbage First (G1)

Oracle's Java 7 implements G1 garbage-collection, using what is known as a garbage first algorithm. The underlying principle is very simple, and it is expected to bring substantial performance improvements. Here's how it works.

The heap is divided into a number of fixed subareas. A list with references to objects in the area, called a remember set, is kept for each subarea. If a garbage collection is requested, then areas containing the most garbage are swept first, hence garbage first. If no living object in an area, its simply defined as free and the pause time and overhead time can also be controlled by defining a target time. The collector then sweeps only as many areas as it can in the prescribed interval. This is used in Oracle's Hotspot JVM. G1 can be called a generational GC but with a lot more flexibility

The IBM WebSphere JVM

As of JAVA 5, it has added this generational configuration to just mark and sweep algorithm.

The WebSphere nursery is equivalent to the HotSpot young generation, and the WebSphere tenured space is equivalent to the HotSpot old generation. The nursery is divided into two parts of equal size—allocate and survivor areas. Objects are always allocated in the nursery and copy garbage collection is used to copy surviving objects to the survivor area. It differentiates between small and large objects by allocating large objects in a specific non-generational heap or directly in the tenured area. Unlike the HotSpot JVM, the WebSphere JVM treats classes like any other object, placing them in the normal heap. There is no permanent generation and so classes are subjected to garbage collection every time. Under certain circumstances when classes are repeatedly reloaded, this can lead to performance problems.

So we can say that different JVM use different garbage collection methods although the base for each is mostly same. In future versions, these might be improved or new systems may be developed for garbage collection.

Native Method Support

An implementation of the Java Virtual Machine may use conventional stacks, colloquially called "C stacks," to support `native` methods (methods written in a language other than the Java programming language). Native method stacks may also be used by the implementation of an interpreter for the Java Virtual Machine's instruction set in a language such as C. Java Virtual Machine implementations that cannot load `native` methods and that do not themselves rely on conventional stacks need not supply native method stacks. If supplied, native method stacks are typically allocated per thread when each thread is created.

This specification permits native method stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the native method stacks are of a fixed size, the size of each native method stack may be chosen independently when that stack is created.

A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the native method stacks, as well as, in the case of varying-size native method stacks, control over the maximum and minimum method stack sizes

The following exceptional conditions are associated with native method stacks:

- If the computation in a thread requires a larger native method stack than is permitted, the Java Virtual Machine throws a *StackOverflowError*.
- If native method stacks can be dynamically expanded and native method stack expansion is attempted but insufficient memory can be made available, or if insufficient memory can be made available to create the initial native method stack for a new thread, the Java Virtual Machine throws an *OutOfMemoryError*.

Java Native Interface (JNI):

The use of native methods limits the portability of an application, because it involves system-specific code. It may have to interoperate with the Java virtual machine where it runs. The Java Native Interface (JNI) facilitates this interoperability in a platform-neutral way.

JNI is a programming framework that enables Java code running in a JVM to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly.

The JNI framework lets a native method use Java objects in the same way that Java code uses these objects. A native method can create Java objects and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code.

Exceptions

Exceptions allow you to smoothly handle unexpected conditions that occur as your programs run. To demonstrate the way the Java virtual machine handles exceptions, consider a class named *CheckedDivision* which throws a Checked Exception called *DivideByZeroException* which is defined by simply extending *java.lang.Exception class*. Consider an Exception Handling method in the class:

```
static int remainder(int dividend, int divisor) throws DivideByZeroException {
        try {
                return dividend % divisor;
            }
        catch (ArithmeticException e) {
                throw new DivideByZeroException();
        }
}
```

The *remainder* operation throws an ArithmeticException if the divisor of the remainder operation is a zero. This method catches this ArithmeticException and throws a DivideByZeroException. DivideByZeroException is a *checked* exception and the ArithmeticException is *unchecked*. Because the ArithmeticException is unchecked (RuntimeException), a method need not declare this exception in a throws clause even though it might throw it. Any exceptions that are subclasses of either Error or RuntimeException are unchecked. By catching ArithmeticException and then throwing DivideByZeroException, the remainder method forces its clients to deal with the possibility of a divide-by-zero exception, either by catching it or declaring DivideByZeroException in their own throws clauses. This is because checked exceptions, such as DivideByZeroException, thrown within a method must be either caught by the method or declared in the method's throws clause.

javac generates the following bytecode sequence for the remainder method:

- The main bytecode sequence for remainder:

```
0 iload_0          // Push local variable 0 (arg passed as divisor)
1 iload_1          // Push local variable 1 (arg passed as dividend)
2 irem             // Pop divisor, pop dividend, push remainder
3 ireturn          // Return int on top of stack (the remainder)
```

- The bytecode sequence for the catch (ArithmeticException) clause:

```
4 pop              // Pop the reference to the ArithmeticException
5 new #5 <Class DivideByZeroException>
                   // Create and push reference to new DivideByZeroException
```

- DivideByZeroException

```
6 dup              // Duplicate the reference to the new object on the top of the stack.
7 invokenonvirtual #9 <Method DivideByZeroException.<init>()V>
        // Call the constructor for the DivideByZeroException to initialize it (i.e. pop ()).
8 athrow           // Pop the reference to a Throwable object, in this
                   // case the DivideByZeroException, and throw the exception.
```

The bytecode sequence of the `remainder` method has two separate parts. The first part is the normal path of execution for the method. This part goes from pc offset 0 through 3. The second part is the catch clause, which goes from pc offset 4 through 8.

The *irem* instruction in the main bytecode sequence may throw an `ArithmeticException`. If this occurs, the Java virtual machine knows to jump to the bytecode sequence that implements the catch clause by looking up and finding the exception in a table. Each method that catches exceptions is associated with an exception table that is delivered in the class file along with the bytecode sequence of the method. The exception table has one entry for each exception that is caught by each try block. Each entry has four pieces of information: the start and end points, the pc offset within the bytecode sequence to jump to, and a constant pool index of the exception class that is being caught. The exception table for the `remainder` method:

Exception table:

from	to	target	type
0	4	4	<Class java.lang.ArithmeticException>

The above exception table indicates that from pc offset zero through three, inclusive, `ArithmeticException` is caught. The try block's endpoint value, listed in the table under the label "to", is always one more than the last pc offset for which the exception is caught. In this case the endpoint value is listed as four, but the last pc offset for which the exception is caught is three. This range, zero to three inclusive, corresponds to the bytecode sequence that implements the code inside the try block of `remainder`. The target listed in the table is the pc offset to jump to if an `ArithmeticException` is thrown between the pc offsets zero and three, inclusive.

If an exception is thrown during the execution of a method, the JVM searches through the exception table for a matching entry. An exception table entry matches if the current program counter is within the range specified by the entry, and if the exception class thrown is the exception class specified by the entry (or its subclass). The Java virtual machine searches through the exception table in the order in which the entries appear in the table. When the first match is found, the JVM sets the program counter to the new pc offset location and continues execution there. If no match is found, the JVM pops the current stack frame and re-throws the same exception. When the JVM pops the current stack frame, it effectively aborts execution of the current method and returns to the calling method. But it throws the same exception in that method, and so on, until it finds a match or it is thrown out of main method.

A Java programmer can throw an exception with a throw statement. The *athrow* instruction pops the top word from the stack and expects it to be a reference to an object that is a subclass of `Throwable` (or `Throwable` itself). The exception thrown is of the type defined by the popped object reference.

Security

Java's security model is one of the language's key architectural features that makes it an appropriate technology for networked environments. Security is important because networks provide a potential avenue of attack to any computer hooked to them. This concern becomes especially strong in an environment in which software is downloaded across the network and executed locally, as is done with Java applets, for example. The class files are automatically downloaded from web page in a browser and hence a user may encounter applets form untrusted sources also. Thus, Java's security mechanisms help make Java suitable for networks because they establish a proper secure system.

Java's security model is focused on protecting users from hostile programs downloaded from untrusted sources across a network. To accomplish this goal, Java provides a customizable "sandbox" in which Java programs run. A Java program must play only inside its sandbox. It can do anything within the boundaries of its sandbox, but it can't take any action outside those boundaries. The sandbox for untrusted Java applets, for example, prohibits many activities, including:

- Reading or writing to the local disk
- Making a network connection to any host, except the host from which the applet came
- Creating a new process
- Loading a new dynamic library and directly calling a native method

What is a Sandbox ?

In the traditional system, security was achieved by user by using softwares only from trusted sources and by regularly scanning for viruses, which is a tardy process. But once software was installed, it had full control, if it was malicious, it can adversely affect the whole system configurations. The sandbox security model makes it easier to work with software that comes from sources you don't fully trust. Instead of security being established by requiring you to prevent any code you don't trust from ever making its way onto your computer, the sandbox model lets you welcome code from any source. But as it's running, the sandbox restricts code from untrusted sources from taking any actions that could possibly harm your system. The advantage is you don't need to figure out what code you can and can't trust, and you don't need to scan for viruses. The sandbox itself prevents any viruses or other malicious code you may invite into your computer from doing any damage. To understand the sandbox, we must look at several different parts of Java's architecture and understand how they work together.

The fundamental components responsible for Java's sandbox are:

- Safety features built into the Java virtual machine
- The class loader architecture

- The class file verifier
- The security manager and the Java API

One of the greatest strengths of Java's security model is that two factors shown in the above list, the class loader and the security manager are customizable. To customize a sandbox, you write a class that descends from **java.lang.SecurityManager**. In this class, you override methods declared in the superclass that decide whether or not to allow particular actions, such as writing to the local disk.

JVM's Safety Features

Some built-in security mechanisms are present as bytecodes of JVM. The mechanisms are:

- Type-safe reference casting
- Structured memory access (no pointer arithmetic)
- Automatic garbage collection (can't explicitly free allocated memory)
- Array bounds checking
- Checking references for null(Throws an exception)

These safety features let the programs access memory in only safe, organised way which makes the Java program robust and also safe. How does it protect? For instance, if someone could learn where in memory a class loader is stored, they could assign a pointer to that memory and manipulate the class loader's data. And hence by enforcing structured access to memory, the Java virtual machine yields programs that are safe.

Unspecified memory layout is another feature of JVM providing security. The information about Java stacks, method area, garbage collectible heap is not present in the class file. If we look into the class file, we cannot find any memory addresses. When the Java virtual machine loads a class file, it decides where in its internal memory to put the bytecodes and other data it parses from the class file. When the Java virtual machine starts a thread, it decides where to put the Java stack it creates for the thread. When it creates a new object, it decides where in memory to put the object. Thus its not possible to predict, where in the memory the data is being stored just by looking at class file and also JVm specification does not tell anything about memory layout, making our data secure.

The prohibition on unstructured memory access is intrinsic to the bytecode instruction set. SO even if bytecode was written directly by a programmer, it won't allow unstructured access. But using native methods, one can beat around the bytecode and can cause a security breach.

When a thread invokes a native method, that thread leaps outside the sandbox. The security model for native methods therefore is the same traditional approach to computer security described earlier: You have to trust a native method before you call it. So we can say that calling a native method turns our sandbox protection into dust! Native methods don't go

through the Java API (native methods provide a means of going around the Java API), so the security manager isn't checked before a native method attempts to do something that could be potentially damaging. Of course, this is often how the Java API itself gets anything done. Many Java API methods may be implemented as native methods, but the native methods used by the Java API are "trusted." As long as a thread continues to execute the native method, the security policies applied by JVM will not be applicable to it anymore. Dynamic libraries are necessary for invoking native methods and hence the security manager includes a method that decides whether or not a program can load these libraries. If untrusted code is allowed to load a dynamic library, that code could maliciously invoke native methods that wreak havoc with the local system. If a piece of untrusted code is prevented by the security manager from loading a dynamic library, it won't be able to invoke an untrusted native method. Its malicious intent will be thwarted. Applets, for example, aren't allowed to load a new dynamic library and therefore can't install their own new native methods. They can, however, call methods in the Java API, methods that may be native but that are always trusted.

One final mechanism that is built into the Java virtual machine and that contributes to security is structured error handling with exceptions. Because of its support for exceptions, the JVM has something structured to do when a violation occurs. Instead of crashing, the JVM can throw an exception or an error, which may result in the death of the offending thread but shouldn't crash the system. Basically programs begin execution again from the point where the exception was handled.

References

JVM Architecture

http://www.javatpoint.com/internal-details-of-jvm

http://www.javaservletsjspweb.in/2012/02/java-virtual-machine-jvm-architecture.html#.Uz2aJYaQak0

Method Area

https://www.artima.com/insidejvm/ed2/jvm5.html

Class Loader

http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html

Garbage Collection

http://javabook.compuware.com/content/memory/how-garbage-collection-works.aspx

Native Method Support

http://en.wikipedia.org/wiki/Java_Native_Interface http://docs.oracle.com/javase/specs/jvms/se7/html/

Exceptions

http://www.javaworld.com/article/2076868/learn-java/how-the-java-virtual-machine-handles-exceptions.html

Security

http://www.javaworld.com/article/2076989/core-java/java-s-security-architecture.html