**Class Loader In java**

Java class loaders are used to load classes at runtime. ClassLoader in Java works on three principle: **delegation, visibility and uniqueness**. 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. 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. 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. Correct understanding of class loader is must to resolve issues like NoClassDefFoundError in Java and java.lang.ClassNotFoundException, which are related to class loading. ClassLoader is also an important topic in advanced Java Interviews, where good knowledge of working of Java ClassLoader and How classpath works in Java is expected from Java programmer. I have always seen questions like, Can one class be loaded by two different ClassLoader in Java on various Java Interviews. In this Java programming tutorial, we will learn what is ClassLoader in Java, How ClassLoader works in Java and some specifics about Java ClassLoader.

**What is ClassLoader in Java**

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 , Extension and System or Application class loader.** Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java. Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw NullPointerException in Java. Bootstrap class loader is also known as Primordial ClassLoader in Java. Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by sun.misc.Launcher$ExtClassLoader. Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from CLASSPATH environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except Bootstrap class loader, which is implemented in native language mostly in C, all Java class loaders are implemented using java.lang.ClassLoader.

In short here is the location from which Bootstrap, Extension and Application ClassLoader load 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.

**How ClassLoader works in Java**

**Delegation principles**

As discussed on when a class is loaded and initialized in Java, a class is loaded in Java, when its needed. Suppose you 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. Remember Classpath is used to load class files while PATH is used to locate executable like javac or java command.

**Visibility Principle**

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean 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

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);

}

}

}

**Uniqueness Principle**

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. You should follow all class loader principle while writing your own ClassLoader.

**How to load class explicitly in Java**

Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drives we have seen in Java program to Connect Oracle database. As shown in above example you can pass name of ClassLoader which should be used to load that particular class along with binary name of class. Class is loaded by calling loadClass() method of java.lang.ClassLoader class which calls findClass() method to locate bytecodes for corresponding class. In this example Extension ClassLoader uses java.net.URLClassLoader which search for class files and resources in JAR and directories. any search path which is ended using "/" is considered directory. If findClass() does not found the class than it throws java.lang.ClassNotFoundException and if it finds it calls defineClass() to convert bytecodes into a .class instance which is returned to the caller.

**Where to use ClassLoader in Java**

ClassLoader in Java is a powerful concept and used at many places. One of the popular example of ClassLoader is AppletClassLoader which is used to load class by Applet, since Applets are mostly loaded from internet rather than local file system, By using separate ClassLoader you can also loads same class from multiple sources and they will be treated as different class in JVM. J2EE uses multiple class loaders to load class from different location like classes from WAR file will be loaded by Web-app ClassLoader while classes bundled in EJB-JAR is loaded by another class loader. Some web server also supports hot deploy functionality which is implemented using ClassLoader. You can also use ClassLoader to load classes from database or any other persistent store.

That's all about What is ClassLoader in Java and How ClassLoader works in Java. We have seen delegation, visibility and uniqueness principles which is quite important to debug or troubleshoot any ClassLoader related issues in Java. In summary knowledge of How ClassLoader works in Java is must for any Java developer or architect to design Java application and packaging.

http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html

**The ClassLoader**

All classes in a Java application are loaded using some subclass of java.lang.ClassLoader. Loading classes dynamically must therefore also be done using a java.lang.ClassLoader subclass.

When a class is loaded, all classes it references are loaded too. This class loading pattern happens recursively, until all classes needed are loaded. This may not be all classes in the application. Unreferenced classes are not loaded until the time they are referenced.

**The ClassLoader Hierarchy**

Class loaders in Java are organized into a hierarchy. When you create a new standard Java ClassLoader you must provide it with a parent ClassLoader. If a ClassLoader is asked to load a class, it will ask its parent class loader to load it. If the parent class loader can't find the class, the child class loader then tries to load it itself.

**Class Loading**

The steps a given class loader uses when loading classes are:

Check if the class was already loaded.

If not loaded, ask parent class loader to load the class.

If parent class loader cannot load class, attempt to load it in this class loader.

When you implement a class loader that is capable of reloading classes you will need to deviate a bit from this sequence. The classes to reload should not be requested loaded by the parent class loader. More on that later.

**Dynamic Class Loading**

Loading a class dynamically is easy. All you need to do is to obtain a ClassLoader and call its loadClass() method. Here is an example:

public class MainClass {

public static void main(String[] args){

ClassLoader classLoader = MainClass.class.getClassLoader();

try {

Class aClass = classLoader.loadClass("com.jenkov.MyClass");

System.out.println("aClass.getName() = " + aClass.getName());

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

**Dynamic Class Reloading**

Dynamic class reloading is a bit more challenging. Java's builtin Class loaders always checks if a class is already loaded before loading it. Reloading the class is therefore not possible using Java's builtin class loaders. To reload a class you will have to implement your own ClassLoader subclass.

Even with a custom subclass of ClassLoader you have a challenge. Every loaded class needs to be linked. This is done using the ClassLoader.resolve() method. This method is final, and thus cannot be overridden in your ClassLoader subclass. The resolve() method will not allow any given ClassLoader instance to link the same class twice. Therefore, everytime you want to reload a class you must use a new instance of your ClassLoader subclass. This is not impossible, but necessary to know when designing for class reloading.

**Designing your Code for Class Reloading**

As stated earlier you cannot reload a class using a ClassLoader that has already loaded that class once. Therefore you will have to reload the class using a different ClassLoader instance. But this poses som new challenges.

Every class loaded in a Java application is identified by its fully qualified name (package name + class name), and the ClassLoader instance that loaded it. That means, that a class MyObject loaded by class loader A, is not the same class as the MyObject class loaded with class loader B. Look at this code:

MyObject object = (MyObject)

myClassReloadingFactory.newInstance("com.jenkov.MyObject");

Notice how the MyObject class is referenced in the code, as the type of the object variable. This causes the MyObject class to be loaded by the same class loader that loaded the class this code is residing in.

If the myClassReloadingFactory object factory reloads the MyObject class using a different class loader than the class the above code resides in, you cannot cast the instance of the reloaded MyObject class to the MyObject type of the object variable. Since the two MyObject classes were loaded with different class loaders, the are regarded as different classes, even if they have the same fully qualified class name. Trying to cast an object of the one class to a reference of the other will result in a ClassCastException.

It is possible to work around this limitation but you will have to change your code in either of two ways:

Use an interface as the variable type, and just reload the implementing class.

Use a superclass as the variable type, and just reload a subclass.

Here are two coresponding code examples:

MyObjectInterface object = (MyObjectInterface)

myClassReloadingFactory.newInstance("com.jenkov.MyObject");

MyObjectSuperclass object = (MyObjectSuperclass)

myClassReloadingFactory.newInstance("com.jenkov.MyObject");

Either of these two methods will work if the type of the variable, the interface or superclass, is not reloaded when the implementing class or subclass is reloaded.

To make this work you will of course need to implement your class loader to let the interface or superclass be loaded by its parent. When your class loader is asked to load the MyObject class, it will also be asked to load the MyObjectInterface class, or the MyObjectSuperclass class, since these are referenced from within the MyObject class. Your class loader must delegate the loading of those classes to the same class loader that loaded the class containing the interface or superclass typed variables.

**ClassLoader Load / Reload Example**

The text above has contained a lot of talk. Let's look at a simple example. Below is an example of a simple ClassLoader subclass. Notice how it delegates class loading to its parent except for the one class it is intended to be able to reload. If the loading of this class is delegated to the parent class loader, it cannot be reloaded later. Remember, a class can only be loaded once by the same ClassLoader instance.

As said earlier, this is just an example that serves to show you the basics of a ClassLoader's behaviour. It is not a production ready template for your own class loaders. Your own class loaders should probably not be limited to a single class, but a collection of classes that you know you will need to reload. In addition, you should probably not hardcode the class paths either.

public class MyClassLoader extends ClassLoader{

public MyClassLoader(ClassLoader parent) {

super(parent);

}

public Class loadClass(String name) throws ClassNotFoundException {

if(!"reflection.MyObject".equals(name))

return super.loadClass(name);

try {

String url = "file:C:/data/projects/tutorials/web/WEB-INF/" +

"classes/reflection/MyObject.class";

URL myUrl = new URL(url);

URLConnection connection = myUrl.openConnection();

InputStream input = connection.getInputStream();

ByteArrayOutputStream buffer = new ByteArrayOutputStream();

int data = input.read();

while(data != -1){

buffer.write(data);

data = input.read();

}

input.close();

byte[] classData = buffer.toByteArray();

return defineClass("reflection.MyObject",

classData, 0, classData.length);

} catch (MalformedURLException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

}

Below is an example use of the MyClassLoader.

public static void main(String[] args) throws

ClassNotFoundException,

IllegalAccessException,

InstantiationException {

ClassLoader parentClassLoader = MyClassLoader.class.getClassLoader();

MyClassLoader classLoader = new MyClassLoader(parentClassLoader);

Class myObjectClass = classLoader.loadClass("reflection.MyObject");

AnInterface2 object1 =

(AnInterface2) myObjectClass.newInstance();

MyObjectSuperClass object2 =

(MyObjectSuperClass) myObjectClass.newInstance();

//create new class loader so classes can be reloaded.

classLoader = new MyClassLoader(parentClassLoader);

myObjectClass = classLoader.loadClass("reflection.MyObject");

object1 = (AnInterface2) myObjectClass.newInstance();

object2 = (MyObjectSuperClass) myObjectClass.newInstance();

}

Here is the reflection.MyObject class that is loaded using the class loader. Notice how it both extends a superclass and implements an interface. This is just for the sake of the example. In your own code you would only have to one of the two - extend or implement.

public class MyObject extends MyObjectSuperClass implements AnInterface2{

//... body of class ... override superclass methods

// or implement interface methods

}

http://viralpatel.net/blogs/java-dynamic-class-loading-java-reflection-api/

One of the reason why Java language has been so useful and used widely is the set of APIs that comes with the language (and 3rd party APIs like iText etc). Using these APIs one do a whole lot unimaginable stuff.

Java Reflection API are one of such APIs that extend the horizon of a Java programmer and enables him to code some really great stuffs.

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

Dynamic Java Class loading is an important feature of the Java Virtual Machine because it provides the Java platform with the ability to install software components at run-time. It has a number of unique characteristics. First of all, lazy loading means that classes are loaded on demand and at the last moment possible. Second, dynamic class loading maintains the type safety of the Java Virtual Machine by adding link-time checks, which replace certain run-time checks and are performed only once. Moreover, programmers can define their own class loaders that, for example, specify the remote location from which certain classes are loaded, or assign appropriate security attributes to them. Finally, class loaders can be used to provide separate name spaces for various software components. For example, a browser can load applets from different web pages using separate class loaders, thus maintaining a degree of isolation between those applet classes. In fact, these applets can contain classes of the same name — these classes are treated as distinct types by the Java Virtual Machine.

Let us see an example of Dynamic class loading using Java Reflection API. Following is our DemoClass that needs to be loaded dynamically and method demoMethod() needs to be called.

class DemoClass {

public String demoMethod(String demoParam) {

System.out.println("Parameter passed: " + demoParam);

return DemoClass.class.getName();

}

}

So to load above class file dynamically following code can be used.

public class DynamicClassLoadingExample {

public static void main(String[] args) {

try {

ClassLoader myClassLoader = ClassLoader.getSystemClassLoader();

// Step 2: Define a class to be loaded.

String classNameToBeLoaded = "net.viralpatel.itext.pdf.DemoClass";

// Step 3: Load the class

Class myClass = myClassLoader.loadClass(classNameToBeLoaded);

// Step 4: create a new instance of that class

Object whatInstance = myClass.newInstance();

String methodParameter = "a quick brown fox";

// Step 5: get the method, with proper parameter signature.

// The second parameter is the parameter type.

// There can be multiple parameters for the method we are trying to call,

// hence the use of array

Method myMethod = myClass.getMethod("demoMethod",new Class[] { String.class });

// Step 6:

// Calling the real method. Passing methodParameter as

// parameter. You can pass multiple parameters based on

// the signature of the method you are calling. Hence

// there is an array.

String returnValue = (String) myMethod.invoke(whatInstance, new Object[] { methodParameter });

System.out.println("The value returned from the method is:"+ returnValue);

} catch (SecurityException e) {

e.printStackTrace();

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (NoSuchMethodException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

}

}

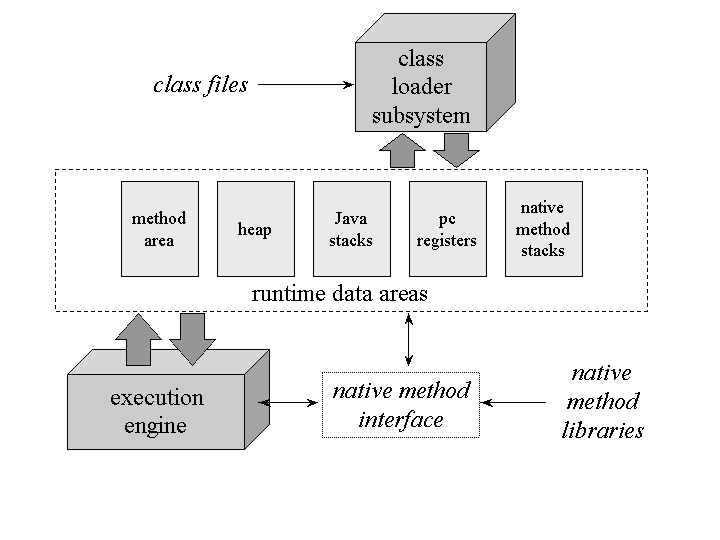
}

The above code is pretty much self explanatory. We have used ClassLoader.getSystemClassLoader() method to get instance of class java.lang.ClassLoader. We have loaded our demo class DemoClass using method loadClass() of ClassLoader and invoked the desired method.

**JVM Architecture and Class Loading – By Debadatta Mishra**

The behavior of a virtual machine instance is described in terms of subsystems, memory areas, data types, and instructions.

When a Java virtual machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas. Each instance of the Java virtual machine has one method area and one heap. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap. As each new thread comes into existence, it gets its own pc register (program counter) and Java stack. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A thread's Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in native method stacks, as well as possibly in registers or other implementation-dependent memory areas.



The Java stack is composed of stack frames (or frames). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

The Java virtual machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values. This approach was taken by Java's designers to keep the Java virtual machine's instruction set compact and to facilitate implementation on architectures with few or irregular general purpose registers. In addition, the stack-based architecture of the Java virtual machine's instruction set facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run-time in some virtual machine implementations. All the primitive types of the Java programming language are primitive types of the Java virtual machine. Although boolean qualifies as a primitive type of the Java virtual machine, the instruction set has very limited support for it. When a compiler translates Java source code into bytecodes, it uses ints or bytes to represent booleans. In the Java virtual machine, false is represented by integer zero and true by any non-zero integer. Operations involving boolean values use ints. Arrays of boolean are accessed as arrays of byte, though they may be represented on the heap as arrays of byte or as bit fields.

## The Class Loader Subsystem

he class loader subsystem involves many other parts of the Java virtual machine and several classes from the java.lang library. For example, user-defined class loaders are regular Java objects whose class descends from java.lang.ClassLoader. The methods of class ClassLoader allow Java applications to access the virtual machine's class loading machinery. Also, for every type a Java virtual machine loads, it creates an instance of class java.lang.Class to represent that type. Like all objects, user-defined class loaders and instances of class Class reside on the heap. Data for loaded types resides in the method area.

### Loading, Linking and Initialization

The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. These activities are performed in a strict order:

1. Loading: finding and importing the binary data for a type
2. Linking: performing verification, preparation, and (optionally) resolution
   1. Verification: ensuring the correctness of the imported type
   2. Preparation: allocating memory for class variables and initializing the memory to default values
   3. Resolution: transforming symbolic references from the type into direct references.
3. Initialization: invoking Java code that initializes class variables to their proper starting values.

### The Bootstrap Class Loader

Java virtual machine implementations must be able to recognize and load classes and interfaces stored in binary files that conform to the Java class file format. An implementation is free to recognize other binary forms besides class files, but it must recognize class files.

Every Java virtual machine implementation has a bootstrap class loader, which knows how to load trusted classes, including the classes of the Java API. The Java virtual machine specification doesn't define how the bootstrap loader should locate classes. That is another decision the specification leaves to implementation designers.

Given a fully qualified type name, the bootstrap class loader must in some way attempt to produce the data that defines the type. One common approach is demonstrated by the Java virtual machine implementation in Sun's 1.1 JDK on Windows98. This implementation searches a user-defined directory path stored in an environment variable named CLASSPATH. The bootstrap loader looks in each directory, in the order the directories appear in the CLASSPATH, until it finds a file with the appropriate name: the type's simple name plus ".class". Unless the type is part of the unnamed package, the bootstrap loader expects the file to be in a subdirectory of one the directories in the CLASSPATH. The path name of the subdirectory is built from the package name of the type. For example, if the bootstrap class loader is searching for class java.lang.Object, it will look for Object.class in the java\lang subdirectory of each CLASSPATH directory.

In 1.2, the bootstrap class loader of Sun's Java 2 SDK only looks in the directory in which the system classes (the class files of the Java API) were installed. The bootstrap class loader of the implementation of the Java virtual machine from Sun's Java 2 SDK does not look on the CLASSPATH. In Sun's Java 2 SDK virtual machine, searching the class path is the job of the system class loader, a user-defined class loader that is created automatically when the virtual machine starts up. More information on the class loading scheme of Sun's Java 2 SDK is given in Chapter 8, "The Linking Model."

## The 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. Memory for class (static) variables declared in the class is also taken from the method area.

The manner in which a Java virtual machine implementation represents type information internally is a decision of the implementation designer. For example, multi-byte quantities in class files are stored in big- endian (most significant byte first) order. When the data is imported into the method area, however, a virtual machine can store the data in any manner. If an implementation sits on top of a little-endian processor, the designers may decide to store multi-byte values in the method area in little-endian order.

The virtual machine will search through and use the type information stored in the method area as it executes the application it is hosting. Designers must attempt to devise data structures that will facilitate speedy execution of the Java application, but must also think of compactness. If designing an implementation that will operate under low memory constraints, designers may decide to trade off some execution speed in favor of compactness. If designing an implementation that will run on a virtual memory system, on the other hand, designers may decide to store redundant information in the method area to facilitate execution speed. (If the underlying host doesn't offer virtual memory, but does offer a hard disk, designers could create their own virtual memory system as part of their implementation.) Designers can choose whatever data structures and organization they feel optimize their implementations performance, in the context of its requirements.

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 Lava, for example, and Lava has not yet been loaded, only one thread should be allowed to load it while the other one waits.

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.

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. The unloading of classes--including the conditions under which a class can become "unreferenced"--is described in Chapter 7, "The Lifetime of a Type."

### 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

Inside the Java class file and Java virtual machine, type names are always stored as fully qualified names. In Java source code, a fully qualified name is the name of a type's package, plus a dot, plus the type's simple name. For example, the fully qualified name of class Object in package java.lang is java.lang.Object. In class files, the dots are replaced by slashes, as in java/lang/Object. In the method area, fully qualified names can be represented in whatever form and data structures a designer chooses.

In addition to the basic type information listed previously, the virtual machine must also store for each loaded type:

* The constant pool for the type
* Field information
* Method information
* All class (static) variables declared in the type, except constants
* A reference to class ClassLoader
* A reference to class Class

This data is described in the following sections.

### The Constant Pool

For each type it loads, a Java virtual machine must store a constant pool. A constant pool is an ordered set of constants used by the type, including literals (string, integer, and floating point constants) and symbolic references to types, fields, and methods. Entries in the constant pool are referenced by index, much like the elements of an array. Because it holds symbolic references to all types, fields, and methods used by a type, the constant pool plays a central role in the dynamic linking of Java programs. The constant pool is described in more detail later in this chapter and in Chapter 6, "The Java Class File."

### Field Information

For each field declared in the type, the following information must be stored in the method area. In addition to the information for each field, the order in which the fields are declared by the class or interface must also be recorded. Here's the list for fields:

* The field's name
* The field's type
* The field's modifiers (some subset of public, private, protected, static, final, volatile, transient)

### Method Information

For each method declared in the type, the following information must be stored in the method area. As with fields, the order in which the methods are declared by the class or interface must be recorded as well as the data. Here's the list:

* The method's name
* The method's return type (or void)
* The number and types (in order) of the method's parameters
* The method's modifiers (some subset of public, private, protected, static, final, synchronized, native, abstract)

In addition to the items listed previously, the following information must also be stored with each method that is not abstract or native:

* The method's bytecodes
* The sizes of the operand stack and local variables sections of the method's stack frame (these are described in a later section of this chapter)
* An exception table (this is described in Chapter 17, "Exceptions")

### Class Variables

Class variables are shared among all instances of a class and can be accessed even in the absence of any instance. These variables are associated with the class--not with instances of the class--so they are logically part of the class data in the method area. Before a Java virtual machine uses a class, it must allocate memory from the method area for each non-final class variable declared in the class.

Constants (class variables declared final) are not treated in the same way as non-final class variables. Every type that uses a final class variable gets a copy of the constant value in its own constant pool. As part of the constant pool, final class variables are stored in the method area--just like non-final class variables. But whereas non-final class variables are stored as part of the data for the type that declares them, final class variables are stored as part of the data for any type that uses them. This special treatment of constants is explained in more detail in Chapter 6, "The Java Class File."

### A Reference to Class ClassLoader

For each type it loads, a Java virtual machine must keep track of whether or not the type was loaded via the bootstrap class loader or a user-defined class loader. For those types loaded via a user-defined class loader, the virtual machine must store a reference to the user-defined class loader that loaded the type. This information is stored as part of the type's data in the method area.

The virtual machine uses this information during dynamic linking. When one type refers to another type, the virtual machine requests the referenced type from the same class loader that loaded the referencing type. This process of dynamic linking is also central to the way the virtual machine forms separate name spaces. To be able to properly perform dynamic linking and maintain multiple name spaces, the virtual machine needs to know what class loader loaded each type in its method area. The details of dynamic linking and name spaces are given in Chapter 8, "The Linking Model."

### A Reference to Class Class

An instance of class java.lang.Class is created by the Java virtual machine for every type it loads. The virtual machine must in some way associate a reference to the Class instance for a type with the type's data in the method area.

Your Java programs can obtain and use references to Class objects. One static method in class Class, allows you to get a reference to the Class instance for any loaded class:

// A method declared in class java.lang.Class:

public static Class forName(String className);

If you invoke forName("java.lang.Object"), for example, you will get a reference to the Class object that represents java.lang.Object. If you invoke forName("java.util.Enumeration"), you will get a reference to the Class object that represents the Enumeration interface from the java.util package. You can use forName() to get a Class reference for any loaded type from any package, so long as the type can be (or already has been) loaded into the current name space. If the virtual machine is unable to load the requested type into the current name space, forName() will throw ClassNotFoundException.

An alternative way to get a Class reference is to invoke getClass() on any object reference. This method is inherited by every object from class Object itself:

// A method declared in class java.lang.Object:

public final Class getClass();

If you have a reference to an object of class java.lang.Integer, for example, you could get the Class object for java.lang.Integer simply by invoking getClass() on your reference to the Integer object.

Given a reference to a Class object, you can find out information about the type by invoking methods declared in class Class. If you look at these methods, you will quickly realize that class Class gives the running application access to the information stored in the method area. Here are some of the methods declared in class Class:

// Some of the methods declared in class java.lang.Class:

public String getName();

public Class getSuperClass();

public boolean isInterface();

public Class[] getInterfaces();

public ClassLoader getClassLoader();

These methods just return information about a loaded type. getName() returns the fully qualified name of the type. getSuperClass() returns the Class instance for the type's direct superclass. If the type is class java.lang.Object or an interface, none of which have a superclass, getSuperClass() returns null. isInterface() returns true if the Class object describes an interface, false if it describes a class. getInterfaces() returns an array of Class objects, one for each direct superinterface. The superinterfaces appear in the array in the order they are declared as superinterfaces by the type. If the type has no direct superinterfaces, getInterfaces() returns an array of length zero. getClassLoader() returns a reference to the ClassLoader object that loaded this type, or null if the type was loaded by the bootstrap class loader. All this information comes straight out of the method area.

### Method Tables

The type information stored in the method area must be organized to be quickly accessible. In addition to the raw type information listed previously, implementations may include other data structures that speed up access to the raw data. One example of such a data structure is a method table. For each non-abstract class a Java virtual machine loads, it could generate a method table and include it as part of the class information it stores in the method area. A method table is an array of direct references to all the instance methods that may be invoked on a class instance, including instance methods inherited from superclasses. (A method table isn't helpful in the case of abstract classes or interfaces, because the program will never instantiate these.) A method table allows a virtual machine to quickly locate an instance method invoked on an object. Method tables are described in detail in Chapter 8, "The Linking Model."

### 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();

}

}

The following paragraphs describe how an implementation might execute the first instruction in the bytecodes for the main() method of the Volcano application. Different implementations of the Java virtual machine can operate in very different ways. The following description illustrates one way--but not the only way--a Java virtual machine could execute the first instruction of Volcano's main() method.

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. The symbolic reference is resolved into a direct reference by searching through the method area until the referenced entity is found, loading new classes if necessary.

Finally, the virtual machine is ready to actually allocate memory for a new Lava object. Once again, the virtual machine consults the information stored in the method area. It uses the pointer (which was just put into Volcano's constant pool entry one) to the Lava data (which was just imported into the method area) to find out how much heap space is required by a Lava object.

A Java virtual machine can always determine the amount of memory required to represent an object by looking into the class data stored in the method area. The actual amount of heap space required by a particular object, however, is implementation-dependent. The internal representation of objects inside a Java virtual machine is another decision of implementation designers. Object representation is discussed in more detail later in this chapter.

Once the Java virtual machine has determined the amount of heap space required by a Lava object, it allocates that space on the heap and initializes the instance variable speed to zero, its default initial value. If class Lava's superclass, Object, has any instance variables, those are also initialized to default initial values. (The details of initialization of both classes and objects are given in Chapter 7, "The Lifetime of a Type.")

The first instruction of main() completes by pushing a reference to the new Lava object onto the stack. A later instruction will use the reference to invoke Java code that initializes the speed variable to its proper initial value, five. Another instruction will use the reference to invoke the flow() method on the referenced Lava object.

## The Heap

Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap. As there is only one heap inside a Java virtual machine instance, all threads share it. Because a Java application runs inside its "own" exclusive Java virtual machine instance, there is a separate heap for every individual running application. There is no way two different Java applications could trample on each other's heap data. Two different threads of the same application, however, could trample on each other's heap data. This is why you must be concerned about proper synchronization of multi-threaded access to objects (heap data) in your Java programs.

The Java virtual machine has an instruction that allocates memory on the heap for a new object, but has no instruction for freeing that memory. Just as you can't explicitly free an object in Java source code, you can't explicitly free an object in Java bytecodes. The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application. Usually, a Java virtual machine implementation uses a garbage collector to manage the heap.

### Garbage Collection

A garbage collector's primary function is to automatically reclaim the memory used by objects that are no longer referenced by the running application. It may also move objects as the application runs to reduce heap fragmentation.

A garbage collector is not strictly required by the Java virtual machine specification. The specification only requires that an implementation manage its own heap in some manner. For example, an implementation could simply have a fixed amount of heap space available and throw an OutOfMemory exception when that space fills up. While this implementation may not win many prizes, it does qualify as a Java virtual machine. The Java virtual machine specification does not say how much memory an implementation must make available to running programs. It does not say how an implementation must manage its heap. It says to implementation designers only that the program will be allocating memory from the heap, but not freeing it. It is up to designers to figure out how they want to deal with that fact.

No garbage collection technique is dictated by the Java virtual machine specification. Designers can use whatever techniques seem most appropriate given their goals, constraints, and talents. Because references to objects can exist in many places--Java Stacks, the heap, the method area, native method stacks--the choice of garbage collection technique heavily influences the design of an implementation's runtime data areas. Various garbage collection techniques are described in Chapter 9, "Garbage Collection."

As with the method area, the memory that makes up the heap need not be contiguous, and may be expanded and contracted as the running program progresses. An implementation's method area could, in fact, be implemented on top of its heap. In other words, when a virtual machine needs memory for a freshly loaded class, it could take that memory from the same heap on which objects reside. The same garbage collector that frees memory occupied by unreferenced objects could take care of finding and freeing (unloading) unreferenced classes. Implementations may allow users or programmers to specify an initial size for the heap, as well as a maximum and minimum size.

### Object Representation

The Java virtual machine specification is silent on how objects should be represented on the heap. Object representation--an integral aspect of the overall design of the heap and garbage collector--is a decision of implementation designers

The primary data that must in some way be represented for each object is the instance variables declared in the object's class and all its superclasses. Given an object reference, the virtual machine must be able to quickly locate the instance data for the object. In addition, there must be some way to access an object's class data (stored in the method area) given a reference to the object. For this reason, the memory allocated for an object usually includes some kind of pointer into the method area.

One possible heap design divides the heap into two parts: a handle pool and an object pool. An object reference is a native pointer to a handle pool entry. A handle pool entry has two components: a pointer to instance data in the object pool and a pointer to class data in the method area. The advantage of this scheme is that it makes it easy for the virtual machine to combat heap fragmentation. When the virtual machine moves an object in the object pool, it need only update one pointer with the object's new address: the relevant pointer in the handle pool. The disadvantage of this approach is that every access to an object's instance data requires dereferencing two pointers. This approach to object representation is shown graphically in Figure 5-5. This kind of heap is demonstrated interactively by the HeapOfFish applet, described in Chapter 9, "Garbage Collection."

## The Program Counter

Each thread of a running program has its own pc register, or program counter, which is created when the thread is started. The pc register is one word in size, so it can hold both a native pointer and a returnAddress. As a thread executes a Java method, the pc register contains the address of the current instruction being executed by the thread. An "address" can be a native pointer or an offset from the beginning of a method's bytecodes. If a thread is executing a native method, the value of the pc register is undefined.

## The Java Stack

When a new thread is launched, the Java virtual machine creates a new Java stack for the thread. As mentioned earlier, a Java stack stores a thread's state in discrete frames. The Java virtual machine only performs two operations directly on Java Stacks: it pushes and pops frames.

The method that is currently being executed by a thread is the thread's current method. The stack frame for the current method is the current frame. The class in which the current method is defined is called the current class, and the current class's constant pool is the current constant pool. As it executes a method, the Java virtual machine keeps track of the current class and current constant pool. When the virtual machine encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame.

When a thread invokes a Java method, the virtual machine creates and pushes a new frame onto the thread's Java stack. This new frame then becomes the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data.

A method can complete in either of two ways. If a method completes by returning, it is said to have normal completion. If it completes by throwing an exception, it is said to have abrupt completion. When a method completes, whether normally or abruptly, the Java virtual machine pops and discards the method's stack frame. The frame for the previous method then becomes the current frame.

All the data on a thread's Java stack is private to that thread. There is no way for a thread to access or alter the Java stack of another thread. Because of this, you need never worry about synchronizing multi- threaded access to local variables in your Java programs. When a thread invokes a method, the method's local variables are stored in a frame on the invoking thread's Java stack. Only one thread can ever access those local variables: the thread that invoked the method.

Like the method area and heap, the Java stack and stack frames need not be contiguous in memory. Frames could be allocated on a contiguous stack, or they could be allocated on a heap, or some combination of both. The actual data structures used to represent the Java stack and stack frames is a decision of implementation designers. Implementations may allow users or programmers to specify an initial size for Java stacks, as well as a maximum or minimum size.

## The Stack Frame

The stack frame has three parts: local variables, operand stack, and frame data. The sizes of the local variables and operand stack, which are measured in words, depend upon the needs of each individual method. These sizes are determined at compile time and included in the class file data for each method. The size of the frame data is implementation dependent.

When the Java virtual machine invokes a Java method, it checks the class data to determine the number of words required by the method in the local variables and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the Java stack.

### Local Variables

The local variables section of the Java stack frame is organized as a zero-based array of words. Instructions that use a value from the local variables section provide an index into the zero-based array. Values of type int, float, reference, and returnAddress occupy one entry in the local variables array. Values of type byte, short, and char are converted to int before being stored into the local variables. Values of type long and double occupy two consecutive entries in the array.

To refer to a long or double in the local variables, instructions provide the index of the first of the two consecutive entries occupied by the value. For example, if a long occupies array entries three and four, instructions would refer to that long by index three. All values in the local variables are word-aligned. Dual-entry longs and doubles can start at any index.

The local variables section contains a method's parameters and local variables. Compilers place the parameters into the local variable array first, in the order in which they are declared. Figure 5-9 shows the local variables section for the following two methods:

// On CD-ROM in file jvm/ex3/Example3a.java

class Example3a {

public static int runClassMethod(int i, long l, float f,

double d, Object o, byte b) {

return 0;

}

public int runInstanceMethod(char c, double d, short s,

boolean b) {

return 0;

}

}

### Operand Stack

Like the local variables, the operand stack is organized as an array of words. But unlike the local variables, which are accessed via array indices, the operand stack is accessed by pushing and popping values. If an instruction pushes a value onto the operand stack, a later instruction can pop and use that value.

The virtual machine stores the same data types in the operand stack that it stores in the local variables: int, long, float, double, reference, and returnType. It converts values of type byte, short, and char to int before pushing them onto the operand stack.

Other than the program counter, which can't be directly accessed by instructions, the Java virtual machine has no registers. The Java virtual machine is stack-based rather than register-based because its instructions take their operands from the operand stack rather than from registers. Instructions can also take operands from other places, such as immediately following the opcode (the byte representing the instruction) in the bytecode stream, or from the constant pool. The Java virtual machine instruction set's main focus of attention, however, is the operand stack.

The Java virtual machine uses the operand stack as a work space. Many instructions pop values from the operand stack, operate on them, and push the result. For example, the iadd instruction adds two integers by popping two ints off the top of the operand stack, adding them, and pushing the int result. Here is how a Java virtual machine would add two local variables that contain ints and store the int result in a third local variable:

### Frame Data

In addition to the local variables and operand stack, the Java stack frame includes data to support constant pool resolution, normal method return, and exception dispatch. This data is stored in the frame data portion of the Java stack frame.

Many instructions in the Java virtual machine's instruction set refer to entries in the constant pool. Some instructions merely push constant values of type int, long, float, double, or String from the constant pool onto the operand stack. Some instructions use constant pool entries to refer to classes or arrays to instantiate, fields to access, or methods to invoke. Other instructions determine whether a particular object is a descendant of a particular class or interface specified by a constant pool entry.

Whenever the Java virtual machine encounters any of the instructions that refer to an entry in the constant pool, it uses the frame data's pointer to the constant pool to access that information. As mentioned earlier, references to types, fields, and methods in the constant pool are initially symbolic. When the virtual machine looks up a constant pool entry that refers to a class, interface, field, or method, that reference may still be symbolic. If so, the virtual machine must resolve the reference at that time.

Aside from constant pool resolution, the frame data must assist the virtual machine in processing a normal or abrupt method completion. If a method completes normally (by returning), the virtual machine must restore the stack frame of the invoking method. It must set the pc register to point to the instruction in the invoking method that follows the instruction that invoked the completing method. If the completing method returns a value, the virtual machine must push that value onto the operand stack of the invoking method.

The frame data must also contain some kind of reference to the method's exception table, which the virtual machine uses to process any exceptions thrown during the course of execution of the method. An exception table, which is described in detail in Chapter 17, "Exceptions," defines ranges within the bytecodes of a method that are protected by catch clauses. Each entry in an exception table gives a starting and ending position of the range protected by a catch clause, an index into the constant pool that gives the exception class being caught, and a starting position of the catch clause's code.

When a method throws an exception, the Java virtual machine uses the exception table referred to by the frame data to determine how to handle the exception. If the virtual machine finds a matching catch clause in the method's exception table, it transfers control to the beginning of that catch clause. If the virtual machine doesn't find a matching catch clause, the method completes abruptly. The virtual machine uses the information in the frame data to restore the invoking method's frame. It then rethrows the same exception in the context of the invoking method.

In addition to data to support constant pool resolution, normal method return, and exception dispatch, the stack frame may also include other information that is implementation dependent, such as data to support debugging.

### Possible Implementations of the Java Stack

Implementation designers can represent the Java stack in whatever way they wish. As mentioned earlier, one potential way to implement the stack is by allocating each frame separately from a heap. As an example of this approach, consider the following class:

// On CD-ROM in file jvm/ex3/Example3c.java

class Example3c {

public static void addAndPrint() {

double result = addTwoTypes(1, 88.88);

System.out.println(result);

}

public static double addTwoTypes(int i, double d) {

return i + d;

}

}

Figure 5-11 shows three snapshots of the Java stack for a thread that invokes the addAndPrint() method. In the implementation of the Java virtual machine represented in this figure, each frame is allocated separately from a heap. To invoke the addTwoTypes() method, the addAndPrint() method first pushes an int one and double 88.88 onto its operand stack. It then invokes the addTwoTypes() method.

## Native Method Stacks

In addition to all the runtime data areas defined by the Java virtual machine specification and described previously, a running Java application may use other data areas created by or for native methods. When a thread invokes a native method, it enters a new world in which the structures and security restrictions of the Java virtual machine no longer hamper its freedom. A native method can likely access the runtime data areas of the virtual machine (it depends upon the native method interface), but can also do anything else it wants. It may use registers inside the native processor, allocate memory on any number of native heaps, or use any kind of stack.

Native methods are inherently implementation dependent. Implementation designers are free to decide what mechanisms they will use to enable a Java application running on their implementation to invoke native methods.

Any native method interface will use some kind of native method stack. When a thread invokes a Java method, the virtual machine creates a new frame and pushes it onto the Java stack. When a thread invokes a native method, however, that thread leaves the Java stack behind. Instead of pushing a new frame onto the thread's Java stack, the Java virtual machine will simply dynamically link to and directly invoke the native method. One way to think of it is that the Java virtual machine is dynamically extending itself with native code. It is as if the Java virtual machine implementation is just calling another (dynamically linked) method within itself, at the behest of the running Java program.

If an implementation's native method interface uses a C-linkage model, then the native method stacks are C stacks. When a C program invokes a C function, the stack operates in a certain way. The arguments to the function are pushed onto the stack in a certain order. The return value is passed back to the invoking function in a certain way. This would be the behavior of the of native method stacks in that implementation.

A native method interface will likely (once again, it is up to the designers to decide) be able to call back into the Java virtual machine and invoke a Java method. In this case, the thread leaves the native method stack and enters another Java stack.

Figure 5-13 shows a graphical depiction of a thread that invokes a native method that calls back into the virtual machine to invoke another Java method. This figure shows the full picture of what a thread can expect inside the Java virtual machine. A thread may spend its entire lifetime executin

## Execution Engine

At the core of any Java virtual machine implementation is its execution engine. In the Java virtual machine specification, the behavior of the execution engine is defined in terms of an instruction set. For each instruction, the specification describes in detail what an implementation should do when it encounters the instruction as it executes bytecodes, but says very little about how. As mentioned in previous chapters, implementation designers are free to decide how their implementations will execute bytecodes. Their implementations can interpret, just-in-time compile, execute natively in silicon, use a combination of these, or dream up some brand new technique.

Similar to the three senses of the term "Java virtual machine" described at the beginning of this chapter, the term "execution engine" can also be used in any of three senses: an abstract specification, a concrete implementation, or a runtime instance. The abstract specification defines the behavior of an execution engine in terms of the instruction set. Concrete implementations, which may use a variety of techniques, are either software, hardware, or a combination of both. A runtime instance of an execution engine is a thread.

Each thread of a running Java application is a distinct instance of the virtual machine's execution engine. From the beginning of its lifetime to the end, a thread is either executing bytecodes or native methods. A thread may execute bytecodes directly, by interpreting or executing natively in silicon, or indirectly, by just- in-time compiling and executing the resulting native code. A Java virtual machine implementation may use other threads invisible to the running application, such as a thread that performs garbage collection. Such threads need not be "instances" of the implementation's execution engine. All threads that belong to the running application, however, are execution engines in action.

### The Instruction Set

A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte opcode followed by zero or more operands. The opcode indicates the operation to be performed. Operands supply extra information needed by the Java virtual machine to perform the operation specified by the opcode. The opcode itself indicates whether or not it is followed by operands, and the form the operands (if any) take. Many Java virtual machine instructions take no operands, and therefore consist only of an opcode. Depending upon the opcode, the virtual machine may refer to data stored in other areas in addition to (or instead of) operands that trail the opcode. When it executes an instruction, the virtual machine may use entries in the current constant pool, entries in the current frame's local variables, or values sitting on the top of the current frame's operand stack.

The abstract execution engine runs by executing bytecodes one instruction at a time. This process takes place for each thread (execution engine instance) of the application running in the Java virtual machine. An execution engine fetches an opcode and, if that opcode has operands, fetches the operands. It executes the action requested by the opcode and its operands, then fetches another opcode. Execution of bytecodes continues until a thread completes either by returning from its starting method or by not catching a thrown exception.

From time to time, the execution engine may encounter an instruction that requests a native method invocation. On such occasions, the execution engine will dutifully attempt to invoke that native method. When the native method returns (if it completes normally, not by throwing an exception), the execution engine will continue executing the next instruction in the bytecode stream.

One way to think of native methods, therefore, is as programmer-customized extensions to the Java virtual machine's instruction set. If an instruction requests an invocation of a native method, the execution engine invokes the native method. Running the native method is how the Java virtual machine executes the instruction. When the native method returns, the virtual machine moves on to the next instruction. If the native method completes abruptly (by throwing an exception), the virtual machine follows the same steps to handle the exception as it does when any instruction throws an exception.

Part of the job of executing an instruction is determining the next instruction to execute. An execution engine determines the next opcode to fetch in one of three ways. For many instructions, the next opcode to execute directly follows the current opcode and its operands, if any, in the bytecode stream. For some instructions, such as goto or return, the execution engine determines the next opcode as part of its execution of the current instruction. If an instruction throws an exception, the execution engine determines the next opcode to fetch by searching for an appropriate catch clause.

Several instructions can throw exceptions. The athrow instruction, for example, throws an exception explicitly. This instruction is the compiled form of the throw statement in Java source code. Every time the athrow instruction is executed, it will throw an exception. Other instructions throw exceptions only when certain conditions are encountered. For example, if the Java virtual machine discovers, to its chagrin, that the program is attempting to perform an integer divide by zero, it will throw an ArithmeticException. This can occur while executing any of four instructions--idiv, ldiv, irem, and lrem--which perform divisions or calculate remainders on ints or longs.

Each type of opcode in the Java virtual machine's instruction set has a mnemonic. In the typical assembly language style, streams of Java bytecodes can be represented by their mnemonics followed by (optional) operand values.

## [Parts Of JVM And JVM Architecture Diagram?](http://middlewaremagic.com/weblogic/?p=4456)

# What Are these Different Parts?

## Eden Space:

Eden Space is a Part of Java Heap where the JVM initially creates any objects, where most objects die and quickly are cleanedup by the minor Garbage Collectors (Note: Full Garbage Collection is different from Minor Garbage Collection). Usually any new objects created inside a Java Method go into Eden space and the objects space is reclaimed once the method execution completes. Where as the Instance Variables of a Class usually lives longer until the Object based on that class gets destroyed. When Eden fills up it causes a minor collection, in which some surviving objects are moved to an older generation.

## Survivor Spaces:

Eden Sapce has two Survivor spaces. One survivor space is empty at any given time. These Survivor Spaces serves as the destination of the next copying collection of any living objects in eden and the other survivor space.

The parameter SurvivorRatio can be used to tune the size of the survivor spaces.

-XX:SurvivorRatio=6 sets the ratio between each survivor space and eden to be 1:6

If survivor spaces are too small copying collection overflows directly into the tenured generation.

## Young Generation: (-XX:MaxNewSize)

Till JDK1.3 and 1.4 we used to set the Young Generation Size using **-XX:MaxNewSize**. But from JDK1.4 onwards we set the YoungGeneration size using (**-Xmn**) JVM option.

Young Generation size is controlled by NewRatio.  It means setting -XX:NewRatio=3 means that the ratio between the Old Generation and the Young Generation is  1:3

.

Similarly -XX:NewRatio=8 means that 8:1 ratio of tenured and young generation.

**NewRatio:** NewRatio is actually the ratio between the (YoungGenaration/Old Generations) has default values of 2 on Sparc , 12 on client Intel, and 8 everywhere else.

**NOTE:** After JDK 1.4 The Young Generation Size can be set using  (**-Xmn**) as well.

## **Virtual Space-1: (MaxNewSize – NewSize)**

The First Virtual Space is actually shows the difference between the -XX:NewSize and -XX:MaxNewSize.  Or we can say that it is basically a difference between the Initial Young Size and the Maximum Young Size.

## Java Heap Area: (-Xmx and -Xms)

### Java Heap is a Memory area inside the Java Process which holds the java objects.  Java Heap is a combination of Young Generation Heap and Old Generation Heap. We can set the Initial Java Heap Size using -Xms JVM parameter similarly if we want to set the Maximum Heap Size then we can use -Xmx JVM parameter to define it.

**Example:**

**-Xmx1024m** —> Means Setting the Maximum limit of Heap as 1 GB

**-Xms512m** —> Means setting Java Heap Initial Size as 512m

.

**NOTE-1):** It is always recommended to set the Initial and the Maximum Heap size values as same for better performance.

**NOTE-2):** The Theoretical limitation of Maximum Heap size for a 32 bit JVM is upto 4GB. Because of the Memory Fragmentation, Kernel Space Addressing, Swap memory usages and the Virtual Machine Overheads are some factors JVM does not allow us to allocate whole 4GB memory for Heap in a 32 bit JVM. So usually on 32-bit Windows Operating Systems the Maximum can be from 1.4 GB to 1.6 GB.

.

If we want a Larger memory allocation according to our application requirement then we must choose the 64-bit operating systems with 64 bit JVM. 64-bit JVM provides us a larger address space. So we can have much larger Java Heap  with  the increased number of Threads allocation area. Based on the Nature of your Operating system in a 64 bit JVM you can even set the Maximum Heap size upto 32GB.

Example:        -Xms32g -Xmx32g -Xmn4g

## Virtual Space-2: (MaxHeapSize – InitialHeapSize)

The Second Virtual Space is actually the Difference between the Maximum Heap size (**-Xmx**)and the Initial Heap Size(**-Xms**). This is called as virtual space because initially the JVM will allocate the Initial Heap Size and then according to the requirement the Heap size can grow till the MaxHeapSize.

.

## **PermGen Space: (-XX:MaxPermSize)**

PermGen is a non-heap memory area where the Class Loading happens and the JVM allocates spaces for classes, class meta data,  java methods and the reference Objects here. The PermGen is independent from the Heap Area. It can be resized according to the requirement using -XX:MaxPermSize and -XX:PermSize  JVM Options. The Garbage collection happens in this area of JVM Memory as well. The Garbage collection in this area is called as “Class GC”. We can disable the Class Garbage Collection using the JVM Option -noclassgc. if  ”-noclassgc” Java Option is added while starting the Server. In that case the Classes instances which are not required will not be Garbage collected.

## **Native Area:**

Native Memory is an area which is usually used by the JVM for it’s internal operations and to execute the JNI codes. The JVM Uses Native Memory for Code Optimization and for loading the classes and libraries along with the intermediate code generation.

The Size of the Native Memory depends on the Architecture of the Operating System and the amount of memory which is already commited to the Java Heap. Native memory is an Process Area where the JNI codes gets loaded or JVM Libraries gets loaded or the native Performance packs and the Proxy Modules gets loaded.

There is no JVM Option available to size the Native Area. but we can calculate it approximately using the following formula:

**NativeMemory = (ProcessSize – MaxHeapSize – MaxPermSize)**

**JVM Architecture**

If you start learning Java, you would want to learn full details of how JVM really functioning. In this post, I'm going to explain JVM (Java Virtual Machine) architecture.

|  |
| --- |
| [http://2.bp.blogspot.com/-4g8GW68TQy4/T0J4DOqkE1I/AAAAAAAAJGE/k62CUFwPtRc/s640/JVM-arc1.png](http://2.bp.blogspot.com/-4g8GW68TQy4/T0J4DOqkE1I/AAAAAAAAJGE/k62CUFwPtRc/s1600/JVM-arc1.png) |
| JVM Architecture |

JVM has various sub components internally. You can see all of them from the above diagram.  
  
**1. Class loader sub system:** JVM's class loader sub system performs 3 tasks  
      a. It loads .class file into memory.  
      b. It verifies byte code instructions.  
      c. It allots memory required for the program.  
  
**2. Run time data area:** This is the memory resource used by JVM and it is divided into 5 parts  
      **a. Method area:** Method area stores class code and method code.  
      **b. Heap:** Objects are created on heap.  
      **c. Java stacks:** Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.  
      **d. Program counter registers:** The program counter registers store memory address of the instruction to be executed by the microprocessor.  
      **e. Native method stacks:** The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.  
  
**3. Native method interface:** Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.  
  
**4. Native method library:** holds the native libraries information.  
  
**5. Execution engine:** Execution engine contains interpreter and JIT compiler, which covert byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The HotSpot represent the block of code executed by JIT compiler.  
  
The above information is just very basic guide of what JVM consists of. Please refer "Inside JVM" book in order to learn more about JVM in detail.

**Internal Memory Details**

One of the biggest strength of the Java Platform is the implementation of an automatic memory management in the Java Virtual Maschine. Everybody who has programmed with languages like C/C++ knows about the problems of managing memory allocation and deallocation in the code. With Java problems like deallocating memory too early (corrupted pointer) or too late ([memory leak](http://de.wikipedia.org/wiki/Speicherleck)) cannot occur by specification. The question is: Why am I writing these blog entries?

The problem is that even with an implicit memory management integrated, Java cannot prevent application of being corrupt in sense of memory management, even it is not allowed to explicitly allocate memory in Java. The result of such wrongly programmed code normally is an exception of type: [java.lang.OutOfMemoryError](http://java.sun.com/javase/6/docs/api/java/lang/OutOfMemoryError.html" \o "OutOfMemoryError Javadoc" \t "_blank).

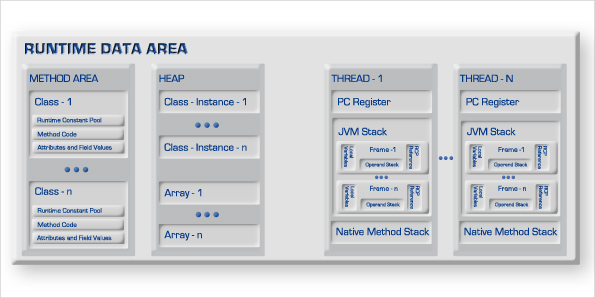
This part of the blog series about Java OutOfMemoryError, will introduce the Java Memory Architecture in detail and shows in which memory areas an java.lang.OutOfMemoryError can occur. Details about the cause of these errors and the tools and methods for analysis will be covered in later entries.

Lets start by looking at the Javadoc of java.lang.OutOfMemoryError:

*Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.*

This description copied from the actual Java API Documentation (Version 6) is not only very short, but in my point of view incomplete and therefore wrong. This description does only cover the heap of the JVM – as we will learn later, OutOfMemoryError can also occur in different areas of the JVMs memory. These errors are not mentioned in the Javadoc, but you can see them every day in real world applications.

The architecture of Java’s memory management is defined for all JVM implementations in the [Java Virtual Machine Specification](http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html)*.*Chapters *3.5 Runtime Data Areas* and *3.6 Frames*are the most relevant for memory architecture. For a better understanding, I’ve drawn the following picture as a summary of the chapters on memory areas in a JVM.

[](http://blog.codecentric.de/wp-content/uploads/2009/12/java-memory-architecture.jpg)

We can basically distinguish memory areas that are available for all threads in a JVM and those memory areas that are exclusively accessible from only one thread. The two areas that are available from all threads are the *Method Area*and the*Heap*.

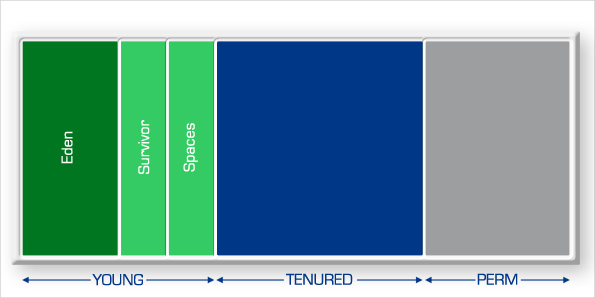
The method area is responsible for storing class information. The Class-Loader will load the bytecode of a class and will pass it to the JVM. The JVM will generate an internal class representation of the bytecode and store it in the method area. The internal representation of a class will have the following data areas:

* *Runtime Constant Pool*Numeric constants of the class of types int, long, float or double, String-constants and symbolic references to all methods, attributes and types of this class.
* *Method Code*The implementation (code) of all methods of this class including constructors etc.
* *Attributes*A list of all named attributes of this class.
* *Fields*Values of all fields of this class as references to the Runtime Constant Pool.

The method area can be part of the heap and will be created at runtime. The size of the method area can be static or dynamic and it does not have to provide a Garbage Collector.

The second memory area that is available for all threads inside the JVM is the *Heap.* The Java heap manages instances of classes (objects) and arrays at runtime. The heap will be created at JVM startup and the size can be static or dynamic. The JVM specification mandates a Garbage Collection mechanism for reclaiming the memory of an object on the Java heap. The implementation of the Garbage Collector is not specified, but it is not allowed to provide the programmer with an explicit mechanism for deallocating the memory of an object.

Lets have a look at the [Sun HotSpot](http://java.sun.com/javase/technologies/hotspot/) implementation as an example:

[](http://blog.codecentric.de/wp-content/uploads/2009/12/sun-hotspot-memory-1.jpg)

The heap is devided into two generations: The Young Generation and the Tenured Generation. The details of this “generational heap” are not relevant in the context of Java OutOfMemoryError as the design is driven by [optimizations of the Garbage Collection algorithm](http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html). The method area is implemented as a separated part: The Permanent Generation. All details about configuration and monitoring of these generations will be covered in the third part of this series: “JVM Monitoring and Configuration”.

This example of the Sun HotSpot JVM memory architecure shows that the JVM specification defines how the memory inside a JVM is organized in general, but leaves enough room for implementation specific optimizations.

In addition to the heap and method area, that are available for all threads of a JVM, every thread also has exclusive access to memory that is created for each thread:

* *PC Register*The Program Counter register. The register points to the current JVM instruction of the method the thread is executing, if the method is not a native method. If it is a native method the content of the PC register is not defined.
* *Java Virtual Machine Stack*Each thread gets its own stack on which so called *Frames* are pushed for each method the thread currently executed. This means that there can be many frames on the stack for nested method calls – but there is only one frame active at the same time for one thread. The frame contains the local variables of the method, a reference to the Runtime Constant Pool of the method’s class and an operand stack for the execution of JVM operations. (The JVM is a stack machine!)
* *Native Methode Stack*Native methods get its own stack, the so called „C-Stack“.

Until now you should have get an overview of the Java Memory Model including its different memory areas – this is essential, because now we will take a closer look at our java.lang.OutOfMemoryError. As mentioned before the Javadoc of this exception is not very meaningful, but the Java Virtual Machine specification defines exactly when and where Java OutOfMemoryError can occur. The difficulty is that theses errors can occur in every memory area I’ve described before. Let’s have a look at the Sun HotSpot JVM and its concrete implementation of OutOfMemoryError errors.

In the heap we get an OutOfMemoryError, if the garbage collector cannot reclaim enough memory for a new object. In such situation the Sun HotSpot JVM shows this error message:

Exception in thread "main": java.lang.OutOfMemoryError: Java heap space

A alternative for this is

Exception in thread "main": java.lang.OutOfMemoryError: Requested array size exceeds VM limit

if the application tries to create an array on the heap that is bigger than the total heap size.

If there is not enough memory in the method area for creating a new class, the Sun HotSpot implementation gets an error in the permanent generation:

Exception in thread "main": java.lang.OutOfMemoryError: PermGen space

Both kinds of OutOfMemoryError occur very often in real life and the reasons for them are very different and will be covered in later blog entries.

OutOfMemory errors in thread exclusive memory areas occur less frequently and are identified by the following error messages in the Sun HotSpot JVM:

Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread

Exception in thread "main": java.lang.OutOfMemoryError: <reason> <stacktrace> (Native method)

The first error is thrown if there are too many threads in the JVM and there is not enough memory left to create a new thread. I’ve seen this because the memory limits of a process have been reached (especially in 32bit operating systems, e.g. on Windows 32bit it is 2GB) or the maximum number of file handles for the user that executes the java process has been reached. The second error message indicates that a memory allocation error on a native stack (JNI method call) has occured.

It is also interesting that a memory allocation error on the JVM stack (too many frames on the stack) does not throw an Java OutOfMemory error but as the JVM specification mandates: [java.lang.StackOverflowError](http://java.sun.com/javase/6/docs/api/java/lang/StackOverflowError.html" \o "Apidocs" \t "_self).

The last variant of the OutOfMemoryError that I know of is

Exception in thread "main": java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?

This error is thrown if there is not enough memory left on the operating system level – which is normally true if other processes are using all of the available memory or the swap space is configured too small.

This first blog entry of the Java OutOfMemoryError series covered the basics of the Java Memory Architecture. In my point of view it is essential to know the different memory areas of the JVM and its functions if you want to understand why a java.lang.OutOfMemoryError occured in your application. I hope that I have made clear that there can be many variations of this error with totally different possible causes. There are a lot of open questions about when and why theses errors occur and how we can monitor and analyze memory problems in our applications. This is exactly what the next episodes ot this [Java OutOfMemoryError series](http://blog.codecentric.de/en/2010/01/java-outofmemoryerror-eine-tragodie-in-sieben-akten/) will cover.