<http://coding-geek.com/jvm-memory-model/>

<https://dzone.com/articles/understanding-the-java-memory-model-and-the-garbag>

<https://dzone.com/articles/java-memory-model-programer%E2%80%99s>

<https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java>

<https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>

<https://blog.codecentric.de/en/2010/01/the-java-memory-architecture-1-act/>

<https://dzone.com/articles/java-memory-architecture-model-garbage-collection>

<https://blog.codecentric.de/en/2010/01/the-java-memory-architecture-1-act/>

# JAVA ARCHITECTURE

## Explain Java architecture?

Java's architecture arises out of four distinct but interrelated technologies:

• the Java programming language

• the Java class file format

• the Java Application Programming Interface

• the Java virtual machine

When you write and run a Java program, you are tapping the power of these four technologies.

You express the program in source files written in the Java programming language, compile the source to Java class files,

and run the class files on a Java virtual machine. When you write your program,

you access system resources (such as I/O, for example) by calling methods in the classes that implement the Java Application Programming

Interface, or Java API. As your program runs, it fulfills your program's Java API calls by invoking methods in class files that implement the Java API

Java virtual machine supports all three prongs of Java's network-oriented architecture: platform independence, security, and network-mobility.

Java virtual machines must be able to execute Java bytecodes, they may use any technique to execute them.

Also, the specification is flexible enough to allow a Java virtual machine to be implemented either completely in software or to varying degrees in hardware.

The flexible nature of the Java virtual machine's specification enables it to be implemented on a wide variety of computers and devices.

Java virtual machine's main job is to load class files and execute the bytecodes they contain. Java virtual machine contains a class loader,

which loads class files from both the program and the Java API.

Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine.

The bytecodes are executed in an execution engine.

The execution engine is one part of the virtual machine that can vary in different implementations.

On a Java virtual machine implemented in software, the simplest kind of execution engine just interprets the bytecodes one at a time.

Another kind of execution engine, one that is faster but requires more memory, is a just-in-time compiler.

In this scheme, the bytecodes of a method are compiled to native machine code the first time the method is invoked.

The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked.

A third type of execution engine is an adaptive optimizer. In this approach, the virtual machine starts by interpreting bytecodes,

but monitors the activity of the running program and identifies the most heavily used areas of code.

As the program runs, the virtual machine compiles to native and optimizes just these heavily used areas.

The rest of the of code, which is not heavily used, remain as bytecodes which the virtual machine continues to interpret.

This adaptive optimization approach enables a Java virtual machine to spend typically 80 to 90% of its time executing highly optimized native code,

while requiring it to compile and optimize only the 10 to 20% of the code that really matters to performance.

Lastly, on a Java virtual machine built on top of a chip that executes Java bytecodes natively, the execution engine is actually embedded in the chip.

Sometimes the Java virtual machine is called the Java interpreter; however, given the various ways in which bytecodes can be executed, this term can be misleading.

While "Java interpreter" is a reasonable name for a Java virtual machine that interprets bytecodes, virtual machines also use other techniques

(such as just-in-time compiling) to execute bytecodes.

Therefore, although all Java interpreters are Java virtual machines, not all Java virtual machines are Java interpreters.

When running on a Java virtual machine that is implemented in software on top of a host operating system,

a Java program interacts with the host by invoking native methods. In Java, there are two kinds of methods: Java and native.

A Java method is written in the Java language, compiled to bytecodes, and stored in class files.

A native method is written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular processor.

Native methods are stored in a dynamically linked library whose exact form is platform specific.

While Java methods are platform independent, native methods are not.

When a running Java program calls a native method, the virtual machine loads the dynamic library that contains the native method and invokes it.

Native methods are the connection between a Java program and an underlying host operating system.

Java gives you a choice. If you want to access resources of a particular host that are unavailable through the Java API,

you can write a platform-specific Java program that calls native methods.

If you want to keep your program platform independent, however, you must access the system resources of the underlying operating system only through the Java API.

---------Class loaders and network security

A Java application can use two types of class loaders: a "bootstrap" class loader and user-defined class loaders.

The bootstrap class loader (there is only one of them) is a part of the Java virtual machine implementation.

For example, if a Java virtual machine is implemented as a C program on top of an existing operating system,

then the bootstrap class loader will be part of that C program.

For each class it loads, the Java virtual machine keeps track of which class loader--whether bootstrap or user-defined--loaded the class.

When a loaded class first refers to another class, the virtual machine requests the referenced class from the same class loader

that originally loaded the referencing class.

For example, if the virtual machine loads class Volcano through a particular class loader,

it will attempt to load any classes Volcano refers to through the same class loader.

If Volcano refers to a class named Lava, perhaps by invoking a method in class Lava, the virtual machine will request Lava from the class loader that loaded Volcano.

The Lava class returned by the class loader is dynamically linked with class Volcano.

Because the Java virtual machine takes this approach to loading classes, classes can by default only see other classes that were loaded by the same class loader.

In this way, Java's architecture enables you to create multiple name-spaces inside a single Java application.

Each class loader in your running Java program has its own name-space, which is populated by the names of all the classes it has loaded

The Java application started by the web browser usually creates a different user-defined class loader for each location on the network

from which it retrieves class files. As a result, class files from different sources are loaded by different user-defined class loaders.

This places them into different name-spaces inside the host Java application.

Because the class files for applets from different sources are placed in separate name- spaces,

the code of a malicious applet is restricted from interfering directly with class files downloaded from any other source.

By allowing you to instantiate user-defined class loaders that know how to download class files across a network,

Java's class loader architecture supports network-mobility.

It supports security by allowing you to load class files from different sources through different user-defined class loaders.

This puts the class files from different sources into different name-spaces, which allows you to restrict or prevent access between code loaded from different sources

--------------JAVA vs C++

When you compile and link a C++ program, the executable binary file you get is specific to a particular target hardware platform and operating system

because it contains machine language specific to the target processor.

A Java compiler, by contrast, translates the instructions of the Java source files into bytecodes, the "machine language" of the Java virtual machine.

In Java, there is no way to directly access memory by arbitrarily casting pointers to a different type or by using pointer arithmetic, as there is in C++.

Java requires that you strictly obey rules of type when working with objects.

If you have a reference (similar to a pointer in C++) to an object of type Mountain, you can only manipulate it as a Mountain.

You can't cast the reference to type Lava and manipulate the memory as if it were a Lava.

Another way Java prevents you from inadvertently corrupting memory is through automatic garbage collection.

A third way Java protects the integrity of memory at run-time is array bounds checking.

One final example of how Java ensures program robustness is by checking object references, each time they are used,

to make sure they are not null. In C++, using a null pointer usually results in a program crash.

In Java, using a null reference results in an exception being thrown.

The Java API is set of runtime libraries that give you a standard way to access the system resources of a host computer.

The class files of the Java API are inherently specific to the host platform.

The API's functionality must be implemented expressly for a particular platform before that platform can host Java programs.

To access the native resources of the host, the Java API calls native methods.

In addition to facilitating platform independence, the Java API contributes to Java's security model.

The methods of the Java API, before they perform any action that could potentially be harmful (such as writing to the local disk), check for permission.

In Java releases prior to 1.2, the methods of the Java API checked permission by querying the security manager.

The security manager is a special object that defines a custom security policy for the application.

A security manager could, for example, forbid access to the local disk.

If the application requested a local disk write by invoking a method from the pre-1.2 Java API, that method would first check with the security manager.

Upon learning from the security manager that disk access is forbidden, the Java API would refuse to perform the write.

In Java 1.2, the job of the security manager was taken over by the access controller,

a class that performs stack inspection to determine whether the operation should be allowed.

(For backwards compatibility, the security manager still exists in Java 1.2.)

By enforcing the security policy established by the security manager and access controller,

the Java API helps to establish a safe environment in which you can run potentially unsafe code.

The Java programming language reflects Java's platform independence in one principal way: the ranges and behavior of its primitive types are defined by the language.

In languages such as C or C++, the range of the primitive type int is determined by its size, and its size is determined by the target platform.

The size of an int in C or C++ is generally chosen by the compiler to match the word size of the platform for which the program is compiled.

This means that a C++ program might have different behavior when compiled for different platforms merely

because the ranges of the primitive types are not consistent across the platforms.

For example, no matter what underlying platform might be hosting the program, an int in Java behaves as a signed 32-bit two's complement number.

A float adheres to the 32-bit IEEE 754 floating point standard. This consistency is also reflected in the internals of the Java virtual machine,

which has primitive data types that match those of the language, and in the class file, where the same primitive data types appear.

By guaranteeing that primitive types behave the same on all platforms,

the Java language itself promotes the platform independence of Java programs.

Security:

http://www.artima.com/insidejvm/ed2/security.html

Network Mobility

http://www.artima.com/insidejvm/ed2/netmob.html

# JVM INTERNALS

## Loader vs initializer vs linking

There are actually three steps in preparing a class for use:

1.Loading, which is performed by the class loader.

This finds the bytecodes (usually, but not necessarily, on your disk in your classpath) and creates a Class object from those bytecodes.

2.Linking. The link phase verifies the bytecodes in the class, allocates storage for static fields, and if necessary,

resolves all references to other classes made by this class.

3.Initialization. If there’s a superclass, initialize that. Execute static initializers and static initialization blocks.

Initialization is delayed until the first reference to a static method (the constructor is implicitly static) or to a non-constant static field:

creating a reference to a Class object using ".class" doesn’t automatically initialize the Class object

If a static final value is a "compile-time constant," such as Initable.staticFinal, that value can be read without causing the Initable class to be initialized.

Making a field static and final, however, does not guarantee this behavior: it might be using some other classes method

If a static field is not final, accessing it always requires linking and initialization before it can be read

instanceof should be preferred whenever you know the kind of class you want to check against in advance. In those very rare cases where you do not,

use isInstance() instead.

## Explain JVM internals?

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

A runtime instance of the Java virtual machine has a clear mission in life: to run one Java application. When a Java application starts, a runtime instance is born.

When the application completes, the instance dies. If you start three Java applications at the same time, on the same computer, using the same concrete implementation,

you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.

Inside the Java virtual machine, threads come in two flavors: daemon and non- daemon.

A daemon thread is ordinarily a thread used by the virtual machine itself, such as a thread that performs garbage collection.

The application, however, can mark any threads it creates as daemon threads.

The initial thread of an application--the one that begins at main()--is a non- daemon thread.

A Java application continues to execute (the virtual machine instance continues to live) as long as any non-daemon threads are still running.

When all non-daemon threads of a Java application terminate, the virtual machine instance will exit.

If permitted by the security manager, the application can also cause its own demise by invoking the exit() metho class Runtime or System.

Eeach Java virtual machine has a class loader subsystem: a mechanism for loading types (classes and interfaces) given fully qualified names.

Each Java virtual machine also has an execution engine: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

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.

Although the same runtime data areas exist in some form in every Java virtual machine implementation, their specification is quite abstract.

Many decisions about the structural details of the runtime data areas are left to the designers of individual implementations.

Different implementations of the virtual machine can have very different memory constraints.

Some implementations may have a lot of memory in which to work, others may have very little.

Some implementations may be able to take advantage of virtual memory, others may not.

The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java virtual machine on a wide variety of computers

and devices.

Some runtime data areas are shared among all of an application's threads and others are unique to individual threads.

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

----------------------- Data Types

The data types can be divided into a set of primitive types and a reference type.

Variables of the primitive types hold primitive values, and variables of the reference type hold reference values.

Reference values refer to objects, but are not objects themselves. Primitive values, by contrast, do not refer to anything. They are the actual data themselves.

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 primitive types of the Java programming language other than boolean form the numeric types of the Java virtual machine.

The numeric types are divided between the integral types: byte, short, int, long, and char, and the floating- point types: float and double.

As with the Java programming language, the primitive types of the Java virtual machine have the same range everywhere.

A long in the Java virtual machine always acts like a 64-bit signed twos complement number, independent of the underlying host platform.

The Java virtual machine works with one other primitive type that is unavailable to the Java programmer: the returnAddress type.

This primitive type is used to implement finally clauses of Java programs. How ?????????????????????????????????

The reference type of the Java virtual machine is cleverly named reference. Values of type reference come in three flavors:

the class type, the interface type, and the array type. All three types have values that are references to dynamically created objects.

The class type's values are references to class instances.

The array type's values are references to arrays, which are full-fledged objects in the Java virtual machine.

The interface type's values are references to class instances that implement an interface.

One other reference value is the null value, which indicates the reference variable doesn't refer to any object.

byte 8-bit signed two's complement integer (-2p7 to 2p7 - 1, inclusive)

short 16-bit signed two's complement integer (-2p15 to 2p15 - 1, inclusive)

int 32-bit signed two's complement integer (-2p31 to 2p31 - 1, inclusive)

long 64-bit signed two's complement integer (-2p63 to 2p63 - 1, inclusive)

char 16-bit unsigned Unicode character (0 to 2p16 - 1, inclusive)

float 32-bit IEEE 754 single-precision float

double 64-bit IEEE 754 double-precision float

returnAddress address of an opcode within the same method

reference reference to an object on the heap, or null

------------------Word Size\*\*\*

The basic unit of size for data values in the Java virtual machine is the word--a fixed size chosen by the designer of each Java virtual machine implementation.

The word size must be large enough to hold a value of type byte, short, int, char, float, returnAddress, or reference.

Two words must be large enough to hold a value of type long or double.

An implementation designer must therefore choose a word size that is at least 32 bits,

but otherwise can pick whatever word size will yield the most efficient implementation.

The word size is often chosen to be the size of a native pointer on the host platform.

The specification of many of the Java virtual machine's runtime data areas are based upon this abstract concept of a word.

For example, two sections of a Java stack frame--the local variables and operand stack-- are defined in terms of words.

These areas can contain values of any of the virtual machine's data types. When placed into the local variables or operand stack,

a value occupies either one or two words.

As they run, Java programs cannot determine the word size of their host virtual machine implementation.

The word size does not affect the behavior of a program. It is only an internal attribute of a virtual machine implementation.

-----------------Class Loader Subsystem

Java virtual machine contains two kinds of class loaders: a bootstrap class loader and user-defined class loaders.

The bootstrap class loader is a part of the virtual machine implementation, and user-defined class loaders are part of the running Java application.

Classes loaded by different class loaders are placed into separate name spaces inside the Java virtual machine.

The 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 a. Verification: ensuring the correctness of the imported type

b. Preparation: allocating memory for class variables and initializing the memory to default values

c. Resolution: transforming symbolic references from the type into direct references.

3. Initialization: invoking Java code that initializes class variables to their proper starting values.

---------------------------------User-Defined Class Loaders

Although user-defined class loaders themselves are part of the Java application, four of the methods in class ClassLoader are gateways into the Java virtual machine:

// Four of the methods declared in class java.lang.ClassLoader:

protected final Class defineClass(String name, byte data[],

int offset, int length);

protected final Class defineClass(String name, byte data[],

int offset, int length, ProtectionDomain protectionDomain);

protected final Class findSystemClass(String name);

protected final void resolveClass(Class c);

Any Java virtual machine implementation must take care to connect these methods of class ClassLoader to the internal class loader subsystem.

The two overloaded defineClass() methods accept a byte array, data[], as input.

Starting at position offset in the array and continuing for length bytes,

class ClassLoader expects binary data conforming to the Java class file format--binary data that represents a new type for the running application -- with

the fully qualified name specified in name. The type is assigned to either a default protection domain, if the first version of defineClass() is used,

or to the protection domain object referenced by the protectionDomain parameter.

Every Java virtual machine implementation must make sure the defineClass() method of class ClassLoader can cause a new type to be imported into the method area.

The findSystemClass() method accepts a String representing a fully qualified name of a type.

When a user-defined class loader invokes this method in version 1.0 and 1.1, it is requesting that the virtual machine attempt to load the named type

via its bootstrap class loader.

If the bootstrap class loader has already loaded or successfully loads the type, it returns a reference to the Class object representing the type.

If it can't locate the binary data for the type, it throws ClassNotFoundException.

In version 1.2, the findSystemClass() method attempts to load the requested type from the system class loader.

Every Java virtual machine implementation must make sure the findSystemClass() method can invoke the bootstrap (if version 1.0 or 1.1)

or system (if version 1.2 or later) class loader in this way.

The resolveClass() method accepts a reference to a Class instance.

This method causes the type represented by the Class instance to be linked (if it hasn't already been linked).

The defineClass() method, described previous, only takes care of loading.

(See the previous section, "Loading, Linking, and Initialization" for definitions of these terms.)

When defineClass() returns a Class instance, the binary file for the type has definitely been located and imported into the method area,

but not necessarily linked and initialized.

Java virtual machine implementations make sure the resolveClass() method of class ClassLoader can cause the class loader subsystem to perform linking.

When to compare using getClass() and when getClass().getName()?

If you want to know whether two objects are of the same type you should use the equals method to compare the two classes -- the first option.

I can't imagine why you'd want to do this, but if you want to know whether two objects with different concrete types have types with the same fully qualified name,

then you could use the second

------------------The Method Area

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.

---------------------------------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.

------------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 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

--------------------Object Representation

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

Another design makes an object reference a native pointer to a bundle of data that contains the object's instance data and a pointer to the object's class data.

This approach requires dereferencing only one pointer to access an object's instance data, but makes moving objects more complicated.

When the virtual machine moves an object to combat fragmentation of this kind of heap, it must update every reference to that object anywhere in the runtime data areas.

The virtual machine needs to get from an object reference to that object's class data for several reasons.

When a running program attempts to cast an object reference to another type,

the virtual machine must check to see if the type being cast to is the actual class of the referenced object or one of its supertypes. .

It must perform the same kind of check when a program performs an instanceof operation.

In either case, the virtual machine must look into the class data of the referenced object.

When a program invokes an instance method, the virtual machine must perform dynamic binding: it must choose the method to invoke based not on the type of the reference

but on the class of the object. To do this, it must once again have access to the class data given only a reference to the object

No matter what object representation an implementation uses, it is likely that a method table is close at hand for each object.

Method tables, because they speed up the invocation of instance methods, can play an important role in achieving good overall performance

for a virtual machine implementation. Method tables are not required by the Java virtual machine specification and may not exist in all implementations.

Implementations that have extremely low memory requirements, for instance, may not be able to afford the extra memory space method tables occupy.

If an implementation does use method tables, however, an object's method table will likely be quickly accessible given just a reference to the object

The special structure has two components:

• A pointer to the full the class data for the object

• The method table for the object The method table is an array of pointers to the data for each instance method that can be invoked on objects of that class.

The method data pointed to by method table includes:

• The sizes of the operand stack and local variables sections of the method's stack

• The method's bytecodes

• An exception table

-----------------------Array Representation

In Java, arrays are full-fledged objects. Like objects, arrays are always stored on the heap. Also like objects,

implementation designers can decide how they want to represent arrays on the heap.

Arrays have a Class instance associated with their class, just like any other object. All arrays of the same dimension and type have the same class.

The length of an array (or the lengths of each dimension of a multidimensional array) does not play any role in establishing the array's class.

For example, an array of three ints has the same class as an array of three hundred ints. The length of an array is considered part of its instance data.

The name of an array's class has one open square bracket for each dimension plus a letter or string representing the array's type.

For example, the class name for an array of ints is "[I". The class name for a three-dimensional array of bytes is "[[[B".

The class name for a two-dimensional array of Objects is "[[Ljava.lang.Object".

https://www.artima.com/insidejvm/ed2/jvm6.html

----------------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, a

nd 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

# MEMORY MODEL

<http://coding-geek.com/jvm-memory-model/>

<https://dzone.com/articles/understanding-the-java-memory-model-and-the-garbag>

<https://dzone.com/articles/java-memory-model-programer%E2%80%99s>

<https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java>

<https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>

<https://blog.codecentric.de/en/2010/01/the-java-memory-architecture-1-act/>

<https://dzone.com/articles/java-memory-architecture-model-garbage-collection>

<https://blog.codecentric.de/en/2010/01/the-java-memory-architecture-1-act/>

## what are data storage type?

Runtime data area in JVM can be divided as below,

1) Method Area : Storage area for compiled class files.

(One per JVM instance)

The runtime constant pool is a subset of the method area which "stores per-class structures such as the runtime constant pool, field and method data,

and the code for methods and constructors, including the special methods used in class and instance initialization and interface type initialization".

2) Heap : Storage area for Objects.

(One per JVM instance)

3) Java stack: Storage are for local variables, results of intermediate operations and refrence variables

(One per thread)

4) PC Register : Stores the address of the next instruction to be executed if the next instruction is native method then the value in pc register will be undefined.

(One per thread)

5) Native method stacks : Helps in executing native methods( methods written in languages other than java).

(One per thread)

## What is difference between Heap and Stack Memory?

Major difference between Heap and Stack memory are as follows:

* + Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
  + Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
  + Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally.

For a detailed explanation with a sample program, read [Java Heap vs Stack Memory](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory).