**Virtual machines**

A machine (with respect to computing) is usually described in terms of its architecture (memory layout, registers, cache etc) and its instruction set.

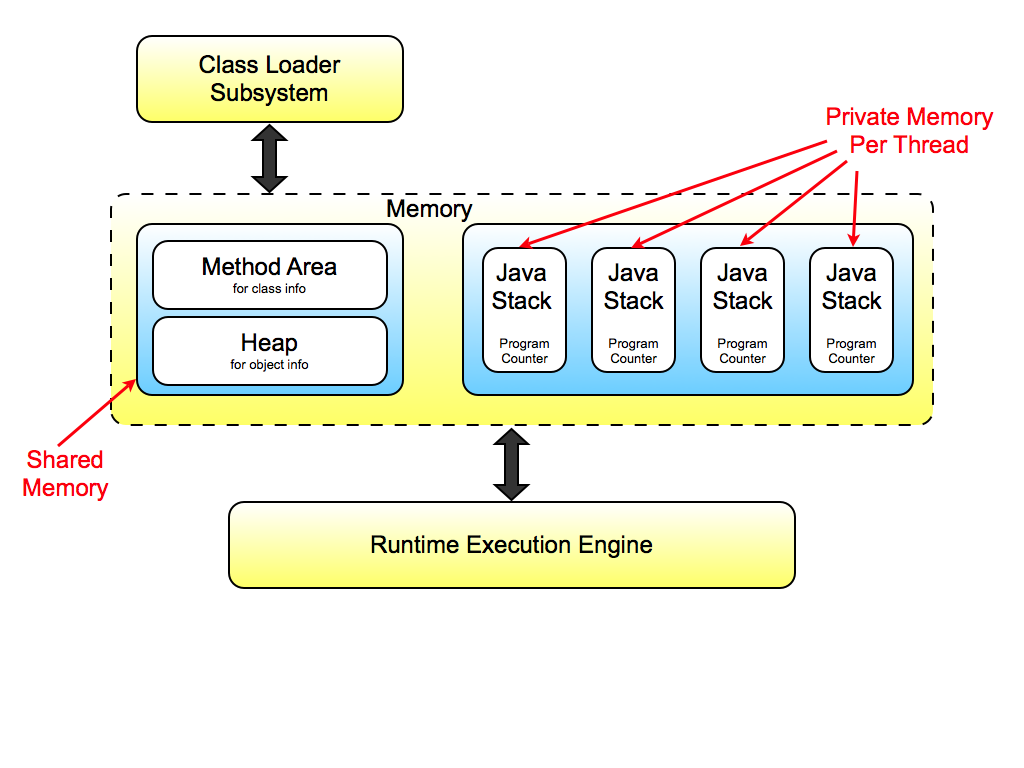
A virtual machine is similarly a machine with its own architecture and instruction set but the difference is that it is built in software rather than hardware. When we say virtual machine we may actually be talking about:

* A runtime instance of the software which represents the machine, or
* A description of this hypothetical machine’s architecture and instruction set, or
* A concrete implementation of this description.

**Architecture of the JVM**

The architecture of the JVM is a high-level abstract description of how a JVM implementation should behave. The behavior is expressed in terms of certain abstract components. The JVM spec does not really give detailed architectural design as this is left to the implementors.

Here is a graphical depiction of the JVM architecture:



Although there are only four threads’ memory areas depicted here, the JVM is not limited to this many threads.

**Points to note**

* The memory in the machine is organized into **shared** data areas – there is **one** heap and **one** method area per JVM instance.
* And **non-shared** data areas – each Thread has its own Java Stack and Program Counter.
* There are **no registers** for temporary data storage – the JVM has a **Stack** **Based** architecture.

**The Program Counter** holds the address of the next (bytecode) instruction to execute for each thread. **The Java Stack** holds state information for that thread – e.g. which methods have been called. It also is used for storing local variables, parameter and performing calculations.

The **Class Loader** subsystem is responsible for location your .class files, verifying that they represent well-formed (bytecode) classes, and loading them into the **Method Area** of the shared memory, linking, and initializing the class variables.

The **Execution Engine** is the part of the specification which describes how bytecode is actually executed and what the expected behavior should be. It details this behavior in terms of the JVM instruction set. Each Java Thread uses an instance of the Execution Engine to execute its code.

**Loading classes**

When the JVM needs to load a type (class):

* The **Class Loader** read the .class file for that type – this gives a steam of bytecode.
* The JVM extracts the type information from this bytecode and creates space in the **Method Area** of memory for it – this is shared memory because all threads may need to access this type. For each type the JVM will need to store:
  + The type’s name
  + Whether it is a class or interface
  + The type’s modifiers (public, abstract or final)
  + Superclass and/or superinterfaces
  + Field information (name, type and modifiers)
  + Method definitions (name, type, modifiers, bytecode, exception table, local memory info)
  + Class variables (i.e. static variables) – non-final class variables are stored with the class that declares them, final class variables are stored with each class that uses them.

The exact structure and organization of all this data is JVM implementation specific. It is not prescribed in the JVM spec. The structures used in any implementation must be efficient for method lookup. A common practice is to use method tables – each type has a table of pointers to code for all methods which can be invoked in objects of that type, including those inherited from superclasses.

**The constant pool**

When the **Class Loader** loads a type in the shared **Method Area**, for each type loaded, an area of the **Method Area** is set aside for what is called, the **Constant Pool** for that type.

The **Constant Pool** contains an ordered list of ‘constants’ used by the type; these are literals (string, integers, floats etc.) and symbolic references to types, fields, methods etc.

Think of this as a big array of all the entities that are referred to in the class definition for the type.

During execution, for non-literals, the symbolic references to the types are used initially to address the various entities. For example, if a class refers to another type Foo – the symbolic reference Foo is used. The execution engine would then see if Foo has been loaded already, if it has, there will be a memory address for it and the symbolic reference will be replaced with a pointer to that address.

The business of looking up symbolic references in the constant pool and replacing them with direct pointers helps with efficiency. It is part of Java’s dynamic linking model called constant pool resolution.

**The heap**

The other area of memory which is shared between all threads in a JVM is the heap.

This is the portion of memory in which object *instances* are stored.

The data that needs to be stored for each object instance is the values of all of the objects instance variables (non-static fields). These must include those inherited from superclasses and interfaces. The data values can be primitive values, or references i.e. pointers to other object instances in the heap.

Each object instance will also need a pointer to the object’s type in the Method Area of memory. This is so that when a method is invoked on an object instance, the correct method definition is invoked.

**Arrays in the heap**

Java treats **Arrays** as object instances and as such they are **stored** on the **heap**.

**Example of memory layout**

class Foo {

public final static fs = 42;

public int i = 0;

public String s = "hi";

public void a(){ ... }

}

class Bar extends Foo {

public static nfs = 10;

public boolean b = true;

Foo f;

public Bar(Foo f){ this.f = f; }

public void b(){ .. Foo.fs ... }

}

public class MemoryExample {

public static void main(String[] args){

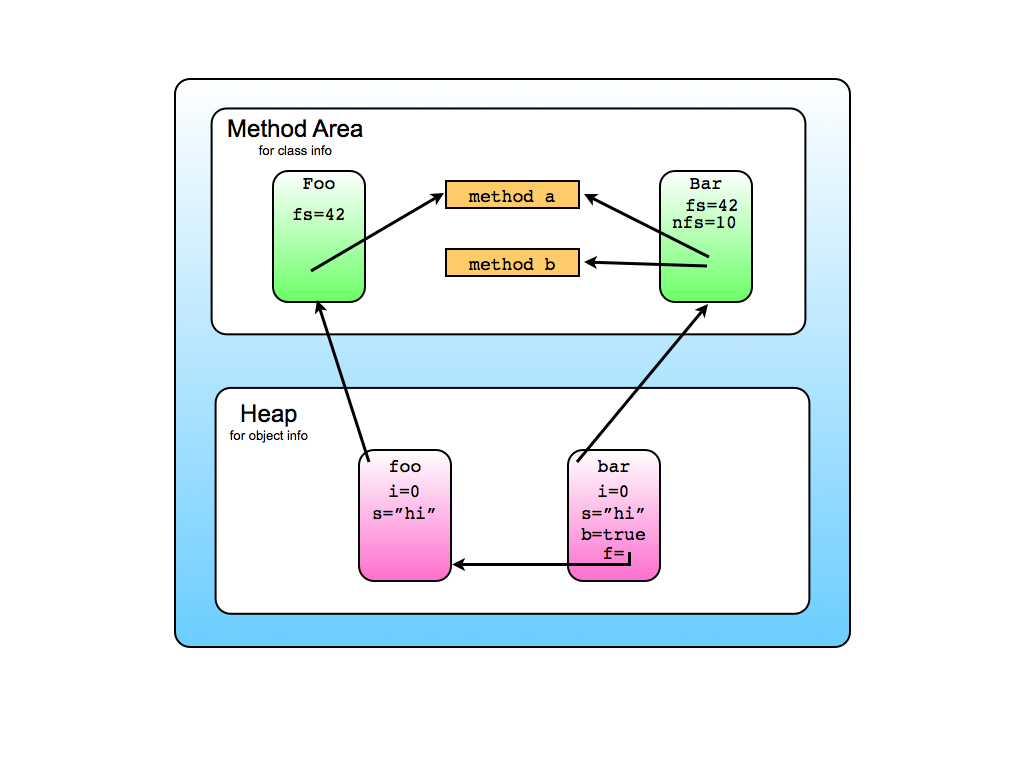
Foo foo = new Foo();

Bar bar = new Bar(foo);

}

}

How this class will be arranged in memory in the JVM. Omitting the main class, constructors and constant pool and just depicting foo and bar and their classes:



**Thread local memory**

So far we have looked at the shared memory areas of the JVM. The other important memory area is the non-shared memory which is private to **each** thread.

The thread local memory is essentially made up of the Program counter and Java Stack.

Each thread has a program counter, which contains a pointer to the Method Area to the instruction of bytecode that it is currently executing, or will next execute.

In addition to this, each thread has its own Stack-based storage facility. There are only two operations permitted on the Stack: push frame and pop frame.

Frames are the unit of operation for the Java Stack. They contain state information for the execution of the thread.

When a thread calls a method, a new frame is pushed in to the thread’s Java stack. All local variables to the method, parameters to the method and temporary calculation is done within this frame.

The method can terminate normally (completes by returning a value) or abruptly (throwing an exception). In both cases, when a method terminates, its frame is popper off the stack and discarded.

**Anatomy of a stack frame**

When a thread calls a method, a new frame is pushed on to the thread’s Java stack. All local variables to the method, parameters to the method and temporary calculation is done within this frame.

Stack frames contain the space needed to run the method they have been created for:

* Space for local variables
* Space for parameters
* Space needed for the frame data
* Space needed for intermediate computations

The size of each of these parts of a frame is method dependent and is determined at compile time and included with the class data in the Method Area. The JVM inspects this information at invocation time in order to create a big enough Stack Frame.

**Nested method calls**

We know that within Java, it is very common to make a call to another method, whilst on the middle of executing a method:

void aMethod(){

...

int i = o.anotherMethod();

...

}

So when a thread is executing aMethod(), the frame for this method is on the thread’s stack. When the thread reaches the point in the code in which anotherMethod() is called a new frame for that method is pushed on to the stack – on top of the frame for aMethod().

Of course, when the anotherMethod() terminates, its frame is popper and discarded so the top of the stack will then contain the frame for aMethod().

**Storing local variables in a stack frame**

A method’s parameters and local variables are organized within a stack frame as a zero-based array of words.

Bytecode instructions which access local variables do so simply using an index in to this array.

Entries in the array are the values of the primitive types *or* references – i.e. pointers to object instances in the heap.

The zero indexed entry in the local variables array of a non-static method is always a reference to the object upon which the method is invoked. This is used for the value of the ‘this’ keyword in method definitions.

**The operand stack (part of each stack)**

The operand stack is the space reserves for calculations.

Because the JVM has no registers, all bytecode instructions which need arguments to execute must first push these arguments on to the operand stack. Any resulting values is left on this stack and must be popped off to retrieve it.

Example: add two variables:

static void add(int x, int y){

int z = x + y;

...

}

Would generate bytecode like:

iload\_0 // push the value of x on to the operand stack

iload\_1 // push the value of y on to the operand stack

iadd // pop two ints, add them, push the result

istore\_2 // pop an int, store it in local variable z

**Frame data**

As well as local variables and space for calculations, stack frames contain certain (implementation dependent) data for helping with the execution of the method.

The frame data portion of the frame will contain:

* A pointer to the method’s defining type’s constant pool
* The address of the bytecode instruction to return to in the previous frame once the current method is terminated
* A pointer to the method’s exception table which the JVM uses to handle any exceptions thrown during the execution of this method.

When a method throws an exception, the JVM will use this exception table to look for a matching catch clause. If there is one, the program counter is updated to point to the handling code, otherwise, the method terminates abruptly and the previous method’s frame is restored. The JVM then rethrows the exception using the previous frame, looking for a handler.

**Looking at the java stack**

There is a useful static method called dumpStack(). This lets you force an Exception which causes a dump of the current Java Stack state to System.out. The dump lists the names of all of the method which currently have frames in the stack.

Here is an example which uses this method. It demonstrates a call stack for five nested method calls, and also the abrupt termination of these due to an exception. See how the exception cascades back to methodOne():

public class DumpExample {

public static void main(String[] args){

DumpExample de = new DumpExample();

de.methodOne();

}

private void methodOne(){

try { this.methodTwo(); }

catch (Exception e) {

Thread.dumpStack();

System.out.println("All but methodOne has

been popped from the stack\n\n\n");

}

}

private void methodTwo() throws Exception {

this.methodThree();

}

private void methodThree() throws Exception {

this.methodFour();

}

private void methodFour() throws Exception {

this.methodFive();

}

private void methodFive() throws Exception {

Thread.dumpStack();

System.out.println("All five methods have frames in the stack\n\n\n");

throw new Exception();

}

}

After running:

java.lang.Exception: Stack trace

at java.lang.Thread.dumpStack(Thread.java:1342)

at DumpExample.methodFive(DumpExample.java:30)

at DumpExample.methodFour(DumpExample.java:26)

at DumpExample.methodThree(DumpExample.java:22)

at DumpExample.methodTwo(DumpExample.java:18)

at DumpExample.methodOne(DumpExample.java:10)

at DumpExample.main(DumpExample.java:6)

All five methods have frames in the stack

java.lang.Exception: Stack trace

at java.lang.Thread.dumpStack(Thread.java:1342)

at DumpExample.methodOne(DumpExample.java:12)

at DumpExample.main(DumpExample.java:6)

All but methodOne has been popped from the stack

**Summary**

The JVM can be considered as an abstract specification of an architecture for implementing the Java language.

The main components of this architecture are:

* The class loader subsystem
* The execution engine
* Shared memory
* Thread local Java Stacks and program counters

The shared memory is subdivided in to the Method Area for storing class information and the Heap for storing object instances.

The Java Stacks store stack frames which contain the data needed to execute methods. Each method call causes a new stack frame to be pushed on to the stack. Method termination causes stack frames to be popped.