# How the Java Virtual Machine (JVM) Works

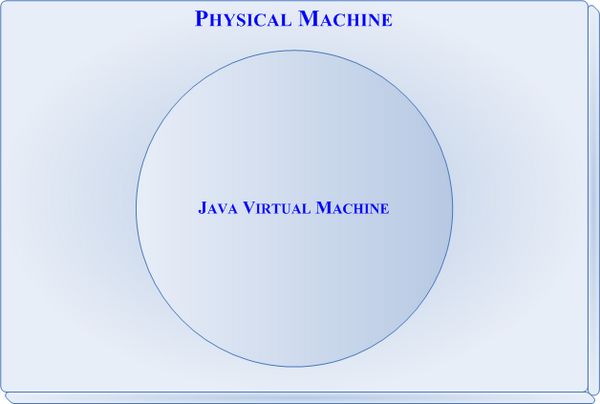
## **Introduction**

In high-level programming languages such as C and C++, we write a program in a human-readable format, and a program called a compiler translates it to a binary format called executable code that the computer can understand and execute. The executable code depends upon the computer machine that we use to execute our program; it is machine dependent. In Java, this process of writing to executing a program is very similar, but with one important difference that allows us to write Java programs that are machine independent.

Using an interpreter, all Java programs are compiled to an intermediate level called byte code. We can run the compiled byte code on any computer with the Java runtime environment installed on it. The runtime environment consists of a virtual machine and its supporting code.

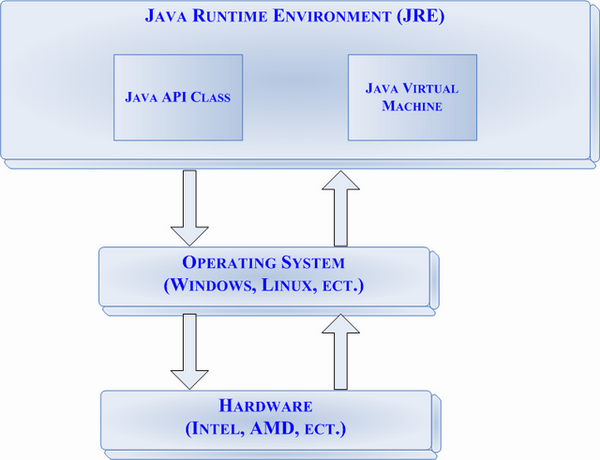
## **JVM is an Emulation**

The difficult part of creating Java byte code is that the source code is compiled for a machine that does not exist. This machine is called the Java Virtual Machine, and it exists only in the memory of our computer. Fooling the Java compiler into creating byte code for a nonexistent machine is only one-half of the ingenious process that makes the Java architecture neutral. The Java interpreter must also make our computer and the byte code file believe they are running on a real machine. It does this by acting as the intermediary between the Virtual Machine and our real machine. (See figure below.)



***Figure 1 - JVM emulation run on a physical machine***

The Java Virtual Machine is responsible for interpreting Java byte code and translating this into actions or Operating System calls. For example, a request to establish a socket connection to a remote machine will involve an Operating System call. Different Operating Systems handle sockets in different ways - but the programmer doesn't need to worry about such details. It is the responsibility of the JVM to handle these translations so that the Operating System and the CPU architecture on which the Java software is running is completely irrelevant to the developer. (See figure below.)



***Figure 2 - JVM handles translations***

## **The Basic Parts of the Java Virtual Machine**

Creating a Virtual Machine within our computer's memory requires building every major function of a real computer down to the very environment within which programs operate. These functions can be broken down into seven basic parts:

* A set of registers
* A stack
* An execution environment
* A garbage-collected heap
* A constant pool
* A method storage area
* An instruction set

### **Registers**

The **registers** of the Java Virtual Machine are similar to the registers in our computer. However, because the Virtual Machine is stack based, its registers are not used for passing or receiving arguments. In Java, registers hold the machine's state, and are updated after each line of byte code is executed, to maintain that state. The following four registers hold the state of the virtual machine:

* **Frame**, the reference frame, and contains a pointer to the execution environment of the current method.
* **Optop**, the operand top, and contains a pointer to the top of the operand stack, and is used to evaluate arithmetic expressions.
* **Pc**, the program counter, and contains the address of the next byte code to be executed.
* **Vars**, the variable register, and contains a pointer to local variables.

All these registers are 32 bits wide, and are allocated immediately. This is possible because the compiler knows the size of the local variables and the operand stack, and because the interpreter knows the size of the execution environment.

### **The Stack**

The Java Virtual Machine uses an **operand stack** to supply parameters to methods and operations, and to receive results back from them. All byte code instructions take operands from the stack, operate on them, and return results to the stack. Like registers in the Virtual Machine, the operand stack is 32 bits wide.

The operand stack follows the last-in first-out (LIFO) methodology, and expects the operands on the stack to be in a specific order. For example, the isub byte code instruction expects two integers to be stored on the top of the stack, which means that the operands must have been pushed there by the previous set of instructions. Isub pops the operands off the stack, subtracts them, and then pushes the results back onto the stack.

In Java, integers are a primitive data type. Each primitive data type has unique instructions that tell it how to operate on operands of that type. For example, the lsub byte code is used to perform long integer subtraction, the fsub byte code is used to perform floating-point subtraction, and the dsub byte code is used to perform long integer subtraction. Because of this, it is illegal to push two integers onto the stack and then treat them as a single long integer. However, it is legal to push a 64-bit long integer onto the stack and have it occupy two 32-bit slots.

Each method in our Java program has a stack frame associated with it. The **stack frame** holds the state of the method with three sets of data: the method's local variables, the method's execution environment, and the method's operand stack. Although the sizes of the local variable and the execution environment data sets are always fixed at the start of the method call, the size of the operand stack changes as the method's byte code instructions are executed. Because the Java stack is 32 bits wide, 64-bit numbers are not guaranteed to be 64-bit aligned.

### **The Execution Environment**

The **execution environment** is maintained within the stack as a data set, and is used to handle dynamic linking, normal method returns, and exception generation. To handle dynamic linking, the execution environment contains symbolic references to methods and variables for the current method and current class. These symbolic calls are translated into actual method calls through dynamic linking to a symbol table.

Whenever a method completes normally, a value is returned to the calling method. The execution environment handles normal method returns by restoring the registers of the caller and incrementing the program counter of the caller to skip the method call instruction. Execution of the program then continues in the calling method's execution environment.

If execution of the current method completes normally, a value is returned to the calling method. This occurs when the calling method executes a return instruction appropriate to the return type.

If the calling method executes a return instruction that is not appropriate to the return type, the method throws an exception or an error. Errors that can occur include dynamic linkage failure, such as a failure to find a class file, or runtime errors, such as a reference outside the bounds of an array. When errors occur, the execution environment generates an exception.

### **The Garbage-Collected Heap**

Each program running in the Java runtime environment has a **garbage-collected heap** assigned to it. Because instances of class objects are allocated from this heap, another word for the heap is **memory allocation pool**. By default, the heap size is set to 1MB on most systems.

Although the heap is set to a specific size when we start a program, it can grow, for example, when new objects are allocated. To ensure that the heap does not get too large, objects that are no longer in use are automatically deallocated or garbage-collected by the Java Virtual Machine.

Java performs automatic garbage collection as a background thread. Each thread running in the Java runtime environment has two stacks associated with it: the first stack is used for Java code; the second is used for C code. Memory used by these stacks is drawn from the total system memory pool. Whenever a new thread starts execution, it is assigned a maximum stack size for the Java code and for the C code. By default, on most systems, the maximum size of the Java code stack is 400KB, and the maximum size of the C code stack is 128KB.

If our system has memory limitations, we can force Java to perform more aggressive cleanup and thus reduce the total amount of memory used. To do this, reduce the maximum size of the Java and C code stacks. If our system has lots of memory, we can force Java to perform less aggressive cleanup, thus reducing the amount of background processing. To do this, increase the maximum size of the Java and C code stacks.

### **The Constant Pool**

Each class in the heap has a **constant pool** associated with it. Because constants do not change, they are usually created at compile time. Items in the constant pool encode all the names used by any method in a particular class. The class contains a count of how many constants exist, and an offset that specifies where a particular listing of constants begins within the class description.

All information associated with a constant follows a specific format based on the type of the constant. For example, **class-level constants** are used to represent a class or an interface, and have the following format:

CONSTANT\_Class\_info {

u1 tag;

u2 name\_index;

}

Where tag is the value of CONSTANT\_Class, and the name\_index provides the string name of the class. The class name for int[][] is [[I. The class name for Thread[] is [Ljava.lang.Thread;.

### **The Method Area**

Java's **method area** is similar to the compiled code areas of the runtime environments used by other programming languages. It stores byte code instructions that are associated with methods in the compiled code, and the symbol table the execution environment needs for dynamic linking. Any debugging or additional information that might need to be associated with a method is stored in this area as well.

### **The Byte Code Instruction Set**

Although programmers prefer to write code in a high-level format, our computer cannot execute this code directly, which is why we must compile Java programs before we can run them. Generally, compiled code is either in a machine-readable format called **machine language** or in an intermediate-level format such as the assembly language or Java byte code.

The byte code instructions used by the Java Virtual Machine resemble Assembler instructions. If you have ever used Assembler, you know that the instruction set is streamlined to a minimum for the sake of efficiency, and that tasks, such as printing to the screen, are accomplished using a series of instructions. For example, the Java language allows us to print to the screen using a single line of code, such as:

System.out.println("Hello world!");

At compile time, the Java compiler converts the single-line print statement to the following byte code:

0 getstatic #6 <Field java.lang.System.out Ljava/io/PrintStream;>

3 ldc #1 <String "Hello world!">

5 invokevirtual #7 <Method java.io.PrintStream.println(Ljava/lang/String;)V>

8 return

The JDK provides a tool for examining byte code called the **Java class file disassembler**. We can run the disassembler by typing javap at the command line.

Because the byte code instructions are in such a low-level format, our programs execute at nearly the speed of programs compiled to machine language. All instructions in machine language are represented by byte streams of 0s and 1s. In a low-level language, byte streams of 0s and 1s are replaced by suitable mnemonics, such as the byte code instruction isub. As with assembly language, the basic format of a byte code instruction is:

<operation> <operands(s)>

Therefore, an instruction in the byte code instruction set consists of a 1-byte opcode specifying the operation to be performed, and zero or more operands that supply parameters or data that will be used by the operation.