What is the JVM? Introducing the Java Virtual Machine

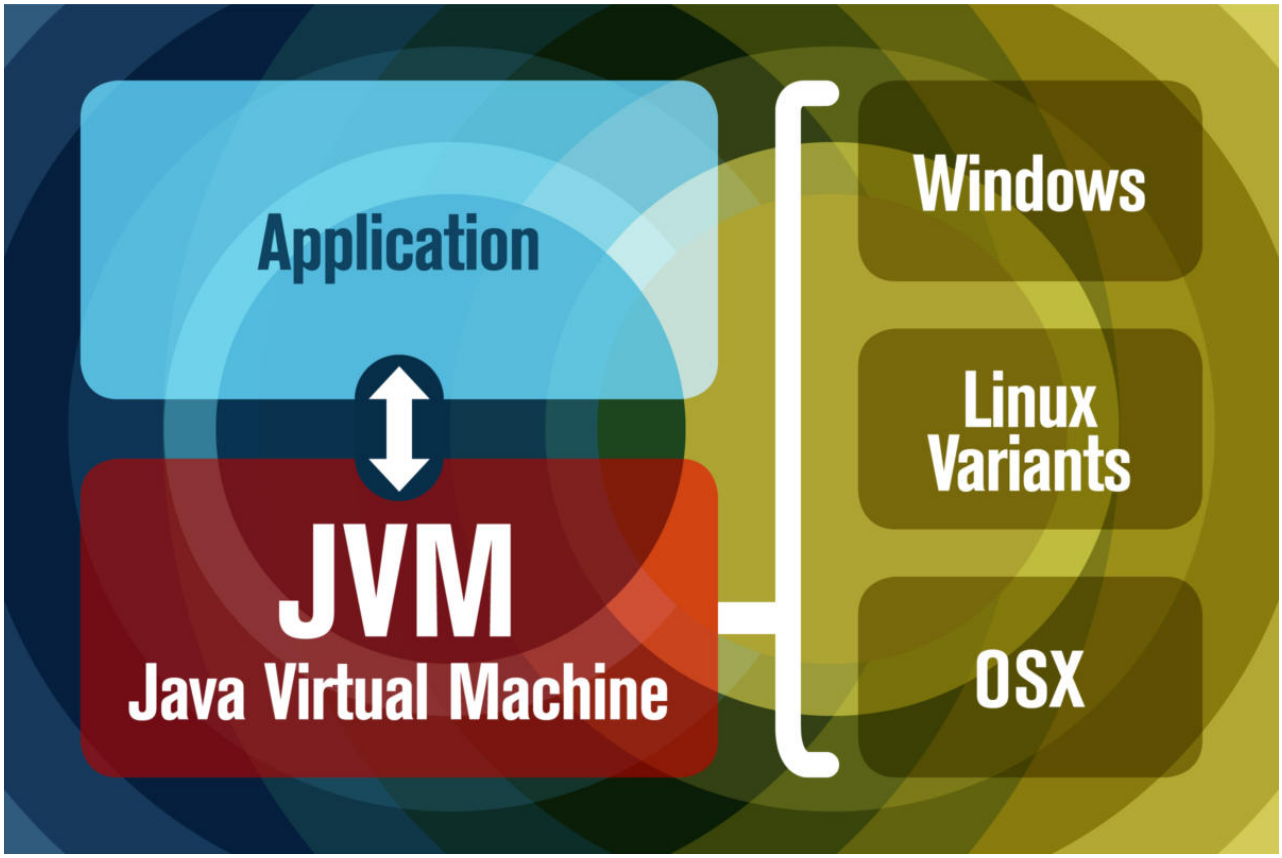
javaworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html

Matthew Tyson

The Java Virtual Machine is a program whose purpose is to execute other programs. It's a simple idea that also stands as one of our greatest examples of coding *kung fu*. The JVM upset the status quo for its time, and continues to support programming innovation today.

What the JVM is used for

The JVM has two primary functions: to allow Java programs to run on any device or operating system (known as the "Write once, run anywhere" principle), and to manage and optimize program memory. When Java was released in 1995, all computer programs were written to a specific operating system, and program memory was managed by the software developer. So the JVM was a revelation.



JavaWorld / IDG

Figure 1: High-level view of the JVM

Having a technical definition for the JVM is useful, and there's also an everyday way that software developers think about it. Let's break those down:

- **Technical definition**: The JVM is the specification for a software program that executes code and provides the runtime environment for that code.
- **Everyday definition**: The JVM is how we run our Java programs. We configure the JVM's settings and then rely on it to manage program resources during execution.

When developers talk about the JVM, we usually mean the process running on a machine, especially a server, that represents and controls resource usage for a Java app. Contrast this to the *JVM specification*, which describes the requirements for building a program that performs these tasks.

Who develops and and maintains the JVM?

The JVM is widely deployed, heavily used, and maintained by some very bright programmers, both corporate and open source. The OpenJDK project is the offspring of the Sun Microsystems decision to open-source Java. OpenJDK has continued through Oracle's stewardship of Java, with much of the heavy lifting these days done by Oracle engineers.

Memory management in the JVM

The most common interaction with a running JVM is to check the memory usage in the heap and stack. The most common adjustment is tuning the JVM's memory settings.

Garbage collection

Before Java, all program memory was managed by the programmer. In Java, program memory is managed by the JVM. The JVM manages memory through a process called *garbage collection*, which continuously identifies and eliminates unused memory in Java programs. Garbage collection happens inside a running JVM.

In the early days, Java came under a lot of criticism for not being as "close to the metal" as C++, and therefore not as fast. The garbage collection process was especially controversial. Since then, a variety of algorithms and approaches have been proposed and used for garbage collection. With consistent development and optimization, garbage collection has vastly improved.

What does 'close to the metal' mean?

When programmers say a programming language or platform is "close to the metal," we mean the developer is able to programmatically (by writing code) manage an operating system's memory. In theory, programmers can wring more performance out of our programs by stipulating how much is used and when to discard it. In most cases, delegating memory management to a highly refined process like the JVM yields better performance and fewer errors than doing it yourself.

The JVM in three parts

It could be said there are three aspects to the JVM: specification, implementation and instance. Let's consider each of these.

1. The JVM specification

First, the JVM is a software specification. In a somewhat circular fashion, the JVM spec highlights that its implementation details are *not* defined within the spec, in order to allow for maximum creativity in its realization:

> "To implement the Java virtual machine correctly, you need only be able to read the `class` file format and correctly perform the operations specified therein."

J.S. Bach once described creating music similarly:

> "All you have to do is touch the right key at the right time."

So, all the JVM has to do is run Java programs correctly. Sounds simple, might even look simple from outside, but it is a massive undertaking, especially given the power and flexibility of the Java language.

The JVM as a virtual machine

The JVM is a *virtual machine* that runs Java class files in a portable way. Being a virtual machine means the JVM is an abstraction of an underlying, actual machine--such as the server that your program is running on. Regardless of what operating system or hardware is actually present, the JVM creates a predictable environment for programs to run within. Unlike a true virtual machine, however, the JVM doesn't create a virtual operating system. It would be more accurate to describe the JVM as a *managed runtime environment*, or a *process virtual machine*.

2. JVM implementations

Implementing the JVM specification results in an actual software program, which is a JVM implementation. In fact, there are many JVM implementations, both open source and proprietary. OpenJDK's HotSpot JVM is the reference implementation, and remains one of the most thoroughly tried-and-tested codebases in the world. HotSpot is also the most commonly used JVM.

Almost all licensed JVM's are created as forks off the OpenJDK and the HotSpot JVM, including Oracle's licensed JDK. Developers creating a licensed fork off the OpenJDK are often motivated by the desire to add OS-specific performance improvements. Typically, you download and install the JVM as a bundled part of a Java Runtime Environment (JRE).

3. A JVM instance

After the JVM spec has been implemented and released as a software product, you may download and run it as a program. That downloaded program is an instance (or instantiated version) of the JVM.

Most of the time, when developers talk about "the JVM," we are referring to a JVM instance running in a software development or production environment. You might say, "Hey Anand, how much memory is the JVM on that server using?" or, "I can't believe I created a circular call and a stack overflow error crashed my JVM. What a newbie mistake!"

What is a software specification?

A *software specification* (or spec) is a human-readable design document that describes how a software system should operate. The purpose of a specification is to create a clear description and requirements for engineers to code to.

Loading and executing class files in the JVM

We've talked about the JVM's role in running Java applications, but how does it perform its function? In order to run Java applications, the JVM depends on the Java class loader and a Java execution engine.

The Java class loader in the JVM

Everything in Java is a class, and all Java applications are built from classes. An application could consist of one class or thousands. In order to run a Java application, a JVM must load compiled .class files into a context, such as a server, where they can be accessed. A JVM depends on its class loader to perform this function.

The Java class loader is the part of the JVM that loads classes into memory and makes them available for execution. Class loaders use techniques like lazy-loading and caching to make class loading as efficient as it can be. That said, class loading isn't the epic brain-teaser that (say) portable runtime memory management is, so the techniques are comparatively simple.

Every Java Virtual Machine includes a class loader. The JVM spec describes standard methods for querying and manipulating the class loader at runtime, but JVM implementations are responsible for fulfilling these capabilities. From the developer's perspective, the underlying class loader mechanisms are typically a black box.

The execution engine in the JVM

Once the class loader has done its work of loading classes, the JVM begins executing the code in each class. The *execution engine* is the JVM component that handles this function. The execution engine is essential to the running JVM. In fact, for all practical purposes, it is the JVM instance.

Executing code involves managing access to system resources. The JVM execution engine stands between the running program--with its demands for file, network and memory resources--and the operating system, which supplies those resources.

How the execution engine manages system resources

System resources can be divided into two broad categories: memory and everything else.

Recall that the JVM is responsible for disposing of unused memory, and that garbage collection is the mechanism that does that disposal. The JVM is also responsible for allocating and maintaining the *referential structure* that the developer takes for granted. As an example, the JVM's execution engine is responsible for taking something like the `new` keyword in Java, and turning it into an OS-specific request for memory allocation.

Beyond memory, the execution engine manages resources for file system access and network I/O. Since the JVM is interoperable across operating systems, this is no mean task. In addition to each application's resource needs, the execution engine must be responsive to each OS environment. That is how the JVM is able to handle in-the-wild demands.

JVM evolution: Past, present, future

In 1995, the JVM introduced two revolutionary concepts that have since become standard fare for modern software development: "Write once, run anywhere" and automatic memory management. Software interoperability was a bold concept at the time, but few developers today would think twice about it. Likewise, whereas our engineering forebears had to manage program memory themselves, my generation grew up with garbage collection.

We could say that James Gosling and Brendan Eich invented modern programming, but thousands of others have refined and built on their ideas over the following decades. Whereas the Java Virtual Machine was originally just for Java, today it has evolved to support many scripting and programming languages, including Scala, Groovy, and Kotlin. Looking forward, it's hard to see a future where the JVM isn't a prominent part of the development landscape.