An analysis of the Java Virtual Machine

Pablo Acereda García and David E. Craciunescu

Universidad de Alcala

Abstract. In the world of programming languages, one of the most famous is the *virtual-machine based* paradigm. This way of designing code compilation and execution has all the benefits of a fast compilation time, while keeping the most prominent features of *virtual-machines*, like the ability to run on any platform and the incomparable level of customization such a framework enables. In this paper, the most prominent features of the Java Virtual Machine (JVM) will be explored. The reader is advised, this paper does not treat the inner architectural mechanisms of the JVM itself, rather it explores the different processes it carries out to create an executable program, i.e. the processing and creation of the *class* file, and it analyzes advantages and disadvantages of the framework itself.

Introduction

When analyzing the Java framework, one wouldn't need much time to realize that the central point of it is the Java Virtual Machine (JVM). Some of Java's most prominent features are due to this very piece of technology. From hardware and operating system independence, to the small size of its compiled code, the JVM is involved in one way or another. The JVM can be classified as an abstract computing machine. Just as any other computing machine, it has a specific instruction set and it manipulates different memory areas during runtime execution. Nowadays, it isn't uncommon to implement a programming language using a virtual machine. In fact, some of the best performing programming languages out there are implemented using virtual machines as well. The most obvious example is the P-Code machine of UCSD Pascal.

The first time the Java Virtual Machine was implemented successfully, it barely emulated the JVM instruction set in software hosted by a PDA. Since then, the most modern implementations have been able to run on desktop, server, and even mobile devices. It is important to note that the Java Virtual Machine does not assume any specific implementation technology or host hardware. It is non-inherently interpreted, but can just as well be implemented by compiling the instruction set to a silicon CPU.

Although the cornerstone of the Java framework, the JVM actually knows nothing of the Java programming language itself, it only knows about the *class* file format, which only contains JVM instructions[1], a symbol table and some other useful information.

[1] Also known as bytecodes.

The JVM enforces strong syntactic and structural constraints on the code of the *class* file. Thanks to this, any programming language that can be expressed as a valid *class* file is able to be hosted on the Java Virtual Machine. The general philosophy of the JVM is to be an available and machine-independent delivery vehicle for the programming languages that want to make use of it. [1]

Instruction set and the class file

The JVM is both a stack machine and a register machine. Each frame for a method call has an "operand stack" and an array of "local variables". The operand stack is used for operand to computations and for receiving the return value of a called method, while local variables serve the same purpose as registers and are also used to pass method arguments. The maximum size of the operand stack and local variable array, computed by the compiler, is part of the attributes of each method. Each can independently size from *0* to $2^{32} - 1$, where each value is 32 bits.
long and *double* types, which are 64 bits, take up to two consecutive local variables (which need not be 64-bit aligned in the local variables array) or one value in the operand stack (but are counted as two units in the depth of the stack).

The Java bytecode format

As previously mentioned, *bytecodes* are the machine language of the JVM. When a JVM loads a *class* file, it gets one stream of bytecodes from each individual method in the class and stores them in the method area. These streams are then executed when invoked while running the program. They can be executed by interpretation, just-in-time compilation, or any other way that was chosen by the designer of the JVM in which they are being executed in particular.
A method's *bytecode* stream is the sequence of instructions the Java Virtual Machine must follow exactly. Each instruction is made up of a one-byte *opcode* followed by zero or mode *operands*, and extra information, if required by the specified instructions the JVM were executing at that time.
The *bytecode* instruction set was designed to be extremely compact. For that reason, the vast majority of instructions are aligned on the boundaries of a byte. Also, the total number of opcodes is small enough so that it considerably reduces the size of files before being loaded by the JVM.
Even though the JVM is a stack machine and a register machine at the same time, it has no registers that allow it to store arbitrary values, and everything must be pushed onto the stack before it can be used in a calculation. The JVM was designed to be hybrid, but at the same time stack-based, in order to facilitate efficient implementation of (now discontinued) architectures known to be register-lacking in nature. Therefore, *bytecode* instructions operate primarily on the stack.

Types

Just like the Java programming language, the JVM operates on two different kinds of types: *primitive types* and *reference types*.
The JVM is programmed to expect that nearly all type checking is done before run time, most normally by a compiler, and it is not the JVM's responsibility to do it itself. Values of primitive types don't have to be tagged or marked for their types to be determined at run time. What the JVM does is create different sets of operations that distinguish the operand types and behave differently according to those types. For example, even though the abstract process for the addition operation is the same one for any numeric value, the JVM has different instructions for the addition operation of each data type. [2]
The Java Virtual Machine also has support for objects. An object is one of two things: a dynamically allocated class instance or an array. A reference to an object is considered to be of type *reference*[2].

Numeric types The numeric types consist of the *integral types* and the *floating point types*.
The integral types are the following:
- ***byte***, whose values are 8-bit signed two's complement integers, and whose default value is zero. The range of accepted values is from -2^7 to 2^7 inclusive.
- ***short***, whose values are 16-bit signed two's complement integers, and whose default value is zero. The range of accepted values is from -2^{15} to 2^{15} inclusive.
- ***int***, whose values are 32-bit signed two's complement integers, and whose default value is zero. The range of accepted values is from -2^{31} to 2^{31} inclusive.
- ***long***, whose values are 64-bit signed two's complement integers, and whose default value is zero. The range of accepted values is from -2^{63} to 2^{63} inclusive.
- ***char***, whose values are 16-bit unsigned integers representing *Unicode* code points in the Basic Multilingual Plane, encoded with *UTF-16*, and whose default value is the null code point (u0000). The range of accepted values is from 0 to 65535 inclusive.
The floating points are:
- ***float***, whose values are elements of the float value set or, where supported, the float-extended-exponent value set, and whose default value is positive zero.
- ***double***, whose values are elements of the double value set or, where supported, the double-extended-exponent value set, and whose default value is positive zero.

[2] The *reference* type is no more than a classic pointer with a different name

The creation of the executable program

During the execution of the Java compiler, a *class* file is created. That file, unfortunately, is far from being ready to be executed in a machine, which makes the class file totally dependent on the JVM environment. Thanks to the JVM, the *class* file has an execution environment and an underlying platform to use as a sandbox when executing the program in question. The middleman the Java Virtual Machine is, does not only provide the playground for the *class* file, it is also the intermediary for the exchange process of services and resources. When breaking down the main processes the JVM carries out in the early stages of program development, three main ones stand apart. These are called *loading, linking and initialization.*

Loading

The process of loading can be defined in many different ways. The *Official Java Virtual Machine Specification* will be used henceforth, and any mention or reference related to it will be described as being part of the *Official Specification.* The aforementioned specification states that the process of loading is *"the process of finding the binary representation of a class or interface type with a particular name and creating a class or interface from the binary representation".*
There are two main types of class loaders that JVM provides: (1) the *bootstrap class loader* and (2) the *user-defined class loader.* The bootstrap class loader is the default class loader, strictly defined and specified in the JVM, and loads class files accordingly to the specification. On the other hand, the user-defined class loader is designed to be able to implement vendor-specific implementations and can load classes in a custom way via the *java.lang.Class* instance.
Normally, a class loader stores the binary representation of the type at load time, in advance, or in relation to a specific group of classes. If any problem is encountered in the early stages of the loading process, the JVM waits until that specific class is called upon to report the error, and only reports it if referenced, even if the error remains.
Therefore, the loading process can be boiled down to these main functions: (1) create a binary stream of data from the class file and (2) parse the binary data according to the internal data structure.[3]

[3] Even though the JVM is able to carry out much more in the linking phase, these two processes are considered the most important ones.

Linking

In the same fashion as the last section, according to the *Official Specification*, the process of linking is the *"process of taking a class or interface and combining it into the run-time state of the JVM so that it can be executed"*.
This linking process starts with a phase called *verification* of the class. Here is where the JVM makes sure that the code follows the semantic rules of the language and its addition does not create any disruption in the JVM environment. Even though this process is well defined by the standard implementation of the JVM, the specification is intentionally flexible enough for vendor-specific JVM implementers to decide when the linking activities take place of the specific processes to follow.
Many little processes happen during the linking phase. First, there is a list of exceptions specified by the JVM to throw under specific circumstances. The JVM completes these checks right from the beginning, to make sure parsed binary data into the internal data structure does not make the program crash. Also, checking is done to make sure the structure of the binary data aligns with the format it expects. Even though multiple verifications take place at multiple stages, it is generally considered that the official verification begins with the linking process.
Once the verification is complete, JVM allocates the memory for the class variables and initializes them according to their respective types. The actual initialization (with user-defined values) does not occur until the next initialization phase. This process has the name of *preparation*.
Finally[4], in the *resolution* phase, JVM locates the references of any classes, fields, methods, etc. in the *constant pool*[5] and determines their real value. Just like many features of the JVM, the Java Symbolic reference resolution is open to vendor-specific implementation. To put it in layman's terms, the verification process checks that the binary representation of a class has the correct structure.
Therefore, the linking process involves the following functions: (1) verification, (2) preparation, and (3) resolution.

Initialization

As per the *Official Specification*, the *"initialization of a class or interface consists of executing its class or interface initialization method"*.
After the class goes through all the previous processes and stages, the initialization phase makes the class ready for its real use. The proces starts with the initialization of the class variables with the expected starting value. Initialization means that the class variables are initialized via some initialization routine described by the programmer and initialize the class's direct superclass if it has not already been initialized. The simple initialization process could be boiled down to two main processes: (1) initialize class variables with a programmer-specific routine and (2) initialize super classes if not already initialized.

[4] And optionally.
[5] Also called symbol table.

Advantages and disadvantages

As mentioned in the first part of this paper, the use of a *virtual-machine* based compilation and execution method has both its advantages and its drawbacks. While many consider the benefits far outweigh the disadvantages, there are still those that believe this scheme to be a misguided one. In this section, the most prominent advantages and disadvantages of the Java Virtual Machine can be found. [3]

Platform Independent

One of the biggest pros of the Java Virtual Machine is its platform independence. The JVM has the ability to be installed on any operating system, such as Windows, Linux, etc. It will transform the programs to bytecode regardless of the hardware of Operating System to be executed. Nowadays, the JVM can run on pretty much any modern operating system, and a wide variety of platforms, ranging from embedded systems to mobile phones.
According to the *Official Java Website* [1] itself, there are between 5 and 10 billion Java devices in the world. This would mean that there may very well be more devices running Java than people there are on planet Earth. The reader can easily see that platform independence is one of the biggest factors influencing such widespread use.

Security

The Java Virtual Machine is designed in such a way to provide security to the host computer in terms of their data and program. The JVM performs verification of the bytecode before running it to prevent the program from performing unsafe operations such as branching to incorrect locations, which may contain data rather than instructions. It also allows the JVM to enforce runtime constraints such as array bounds checking. This means that Java Programs are significanly less likely to suffer from memory safety flaws such as buffer overflow than programs written in languages such as C which do not provide such memory safety guarantees.
The platform does not allow programs to perform certain unsafe operations such as pointer arithmetic or unchecked type casts. It also does not allow manual control over memory allocation and deallocation; users are required to rely on the automatic garbage collector provided by the platform. This also contributes to type safety and memory safety.

Speed

Given the fact that Java code is not executed as native code and needs to be run on a virtual machine for it to work, Java programs will usually take longer to execute in comparison to C programs, for example.
In the early days of Java, there were many criticisms of its performance. Java has been demonstrated to run at a speed comparable with optimised

native code, and modern JVM implementations are regularly benchmarked as one of the fastest language platforms available.
Java bytecode can either be interpreted at run time by the virtual machine, or it can be compiled at load time or runtime into native code which runs directly on the computer's hardware. Interpretation is slower than native execution, and compilation at load time or runtime has an initial performance penalty for the compilation. Modern performant JVM implementations all use the compilation approach, so after the initial startup time the performance is similar to native code.

Cannot experience platform specific features

The Java Virtual Machine has been developed to be compatible with multiple Operating Systems. These systems have specific and special features and that cannot really be changed by a program running on them e.g. the JVM. Therefore, even if the JVM is a sandbox for Java Code, it still has its limitations when it comes to the Operating System itself. A clear example can be that of Graphical User Interfaces. Even though the JVM may specify exactly the look and feel of the program, the operating system can easily override that specification and enforce its own rules.

Correctness

A program that performs, as expected, is said to be correct. Since a Java program relies on the Java Virtual Machine to execute it, the JVM must be free of errors for the program to operate correctly. This reliance on the Java Virtual Machine introduces a possible point of failure for the program. Luckily, the Java Virtual Machine software is produced with very high standards, and therefore it isn't likely to ship with any errors. Regardless, a failure in the Java Virtual Machine is a possibility to be considered.

Conclusions

The Java Virtual Machine has been an invaluable tool and integral part of the Java language framework. Thanks to its different very well designed features it receives the worldspread use and attention it does nowadays. Even though many detractors of the JVM claim its flaws far outweigh its benefits, it is clear that the global community still uses the language and is content enough with it every day.

References

[1] *Java Official Page*. URL: https://go.java/index.html?intcmp=gojava-banner-java-com.
[2] A Taivalsaari. *Virtual Machine Design*. 2003.
[3] *The Java Virtual Machine Specification*. URL: https://docs.oracle.com/javase/specs/jls/se12/html/index.html.