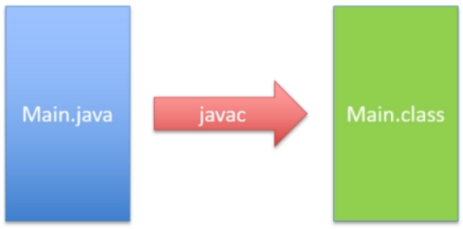
JIT compilation

The java code we write is compiled by java compiler into *byte code*. These are the *.class* files that are created by java and may be packaged together into a *.jar* or a *.war* file. When we run our application using the java command the byte code is then run by the *Java Virtual Machine (JVM)*, i.e., JVM is interpreting the byte code.

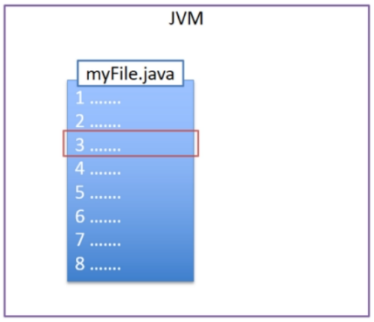


This gives one of the advantages of Java. We can write code once and run it with consistent results on any hardware that has a Java Virtual Machine. So, these same Java code can be run on a Mac, Windows, Linux, or any other operating system for which a JVM exists.

Traditional Interpreted Languages

If you were writing code in a language such as PHP which is not compiled but is interpreted at runtime using an interpreter such as the Apache Web Server. Each line of PHP code is only looked at analyzed and the way to execute it determined as it goes through each line (not the case from PHP 8.0 and later though. They bring in JIT)

JVM Bytecode interpretation

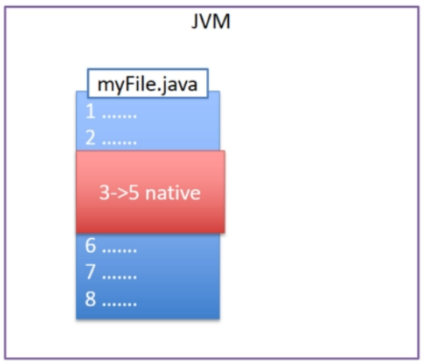


So initially the JVM acts like any other interpreter running each line of code as it is needed. However, this would make the code execution somewhat slow, if you compared this to code written in a language like C which would be compiled to native machine code.

Since C is directly compiled to native code, the operating system can comprehend directly it doesn't need any additional software to interpret or run it. This makes it quick to run compared to interpreted languages. But using these kinds of languages that are compiled natively means that we lose the right once run anywhere feature of Java.

So, to help get around this problem of slower execution in interpreted languages than compiled languages the Java Virtual Machine has a feature called Just In Time compilation (JIT). The JVM will monitor which branches of code are run the most often which methods or parts of methods specifically loops are executed the most frequently and then the virtual machine can decide for example that a particular method is being used a lot and so code execution would be speeded up if that method was compiled to native machine code and it will then do so.

So, at this point some of our application is being run in interpretive mode as byte code and some is no longer byte code but is running as compiled native machine code. The part that has been compiled to native machine code will run faster than the byte code. So if you're running this application on Windows, part of the byte code has been compiled into code that can be natively understood by the Windows OS.



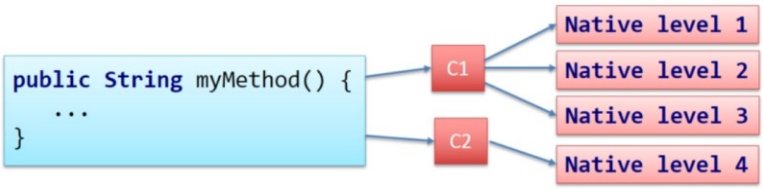
Similarly, if we were running on a Mac then the JVM would have compiled this to native MAC code. Of course, the native Windows code and a native MAC code would be different. In fact the compilation even depends on the architecture of the CPU, i.e., the JIT code compiled for x64 systems would be different for ARM systems even if they have same Windows OS.

With JIT, your code will generally run faster the longer it is left to run. That's because the virtual machine can profile your code and work out which bits of it could be optimized. So a method that runs multiple times every minutes is very likely to be JIT compiled quite quickly but a method that might run once a day might not ever be checked compiled.

The process of compiling converting the byte code to native machine code is run in a separate thread. The virtual machine is of course a multi threaded application itself so the threads within the virtual machine responsible for running the code that is interpreting the byte code and executing the byte code won't be affected by the thread doing JIT compiling.

Hence application will run in interpreted mode until JIT compilation is complete, and native machine code is available. Once it is available, JVM will seamlessly switch to the compiled code for the parts that are compiled and will still run interpreted code for the parts that are not compiled.

JVM JIT compilers



Now there are two compilers built into the Java Virtual Machine called C1 and C2. C1 compiler can do the *first three levels* of compilation. Each has progressively more complex than the last one and the C2 compiler can do the *fourth* level. The virtual machine decides which level of compilation to apply to a particular block of code based on how often it is being run and how complex or time consuming it is. This is called profiling the code. So, for any method which has a number of 1 to 3 the code has been compiled using the C1 compiler and the higher the number the more profile the code has been. If the code has been called enough times, then we reach Level 4 and the C2 compiler will be used instead. Once the code is compiled with C2 at level 4, JVM can then decide to put the code in a code cache. *Code cache* is a special area of memory because that will be the quickest way for it to be accessible and therefore highly compiled code is put there for faster access. This is often called *compilation tier*.