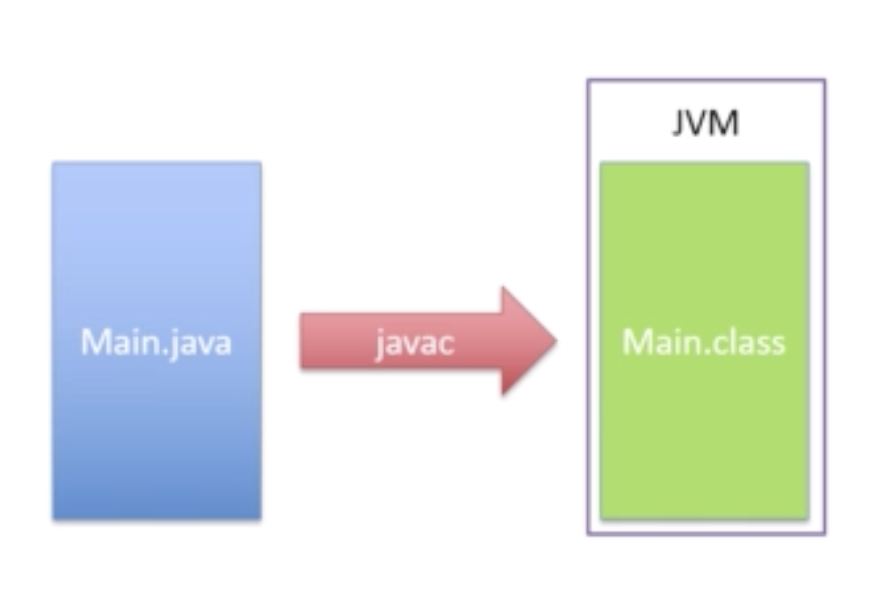
**Just In Time Compilation and the Code Cache**

* **What is bytecode?**

**Chart, waterfall chart

Description automatically generated**

****

**A picture containing diagram

Description automatically generated**

* **The concept of “Just In Time Compilation”**

**A picture containing chart

Description automatically generated**

**Chart

Description automatically generated**

**Chart

Description automatically generated**

**Chart

Description automatically generated**

Bytecode:

Bytecode is a code generated by Java compiler which only the JVM can run.

Once the JVM reads the bytecode, it converts/interprets that to the machine code, this usually makes Java’s performance slow, as it has to interpret at the runtime.

To address this issue, Java uses JIT (Compilation), this analyses the bytecode which are frequently used, once it identifies such a block, it will convert only those lines/blocks to a native code, so that it will run faster.

Native code means there is no interpretation required, so it will run faster.

The process which compiles the bytecode to native code runs in a separate thread.

Whenever the thread compiles the bytecode to native code, it will not affect the running of program, JVM will continue running the program using bytecode until the native code is compiled.

Graphical user interface, text, application

Description automatically generated

To measure the performance of an application with JIT, if you test it by running and stopping the application immediately or frequently, then you won’t find the difference at all. As the application has to run for sometime to identify the block or lines of code for JIT.

* **Introducing the first example project:**

There is one sample project given(chapter-02) in the resources section of chapter-01, import that to IDE.

* **Finding out which methods are being compiled in our applications:**

**Graphical user interface, text, application, chat or text message

Description automatically generated**

For all other additional option we give in future it will follow the same pattern:

XX 🡪 Stands for it is an advanced setting.

+ 🡪 Indicates we want to add option whatever comes after +

Minus (-) 🡪 indicates we want to remove option whatever comes after (-)

Note: The flags are case sensitive

Once the above flag is set, run the program. Apart from the normal output it will print few extra information. In the output if you see the 3 and 4th column, some entries will have “n” and some entries will have “s”.

n 🡪 Indicates “Native”

s 🡪 Indicates “Synchronized”

! 🡪 Indicates some exception handling going on.

% 🡪 Indicates the code is compiled natively and running on special block called “Code cache”. The method is now running in the most optimal way possible.

Text

Description automatically generated

There is another column after n, which prints number between 0 to 4

0 🡪 Means no compilation the code has just been interpreted.

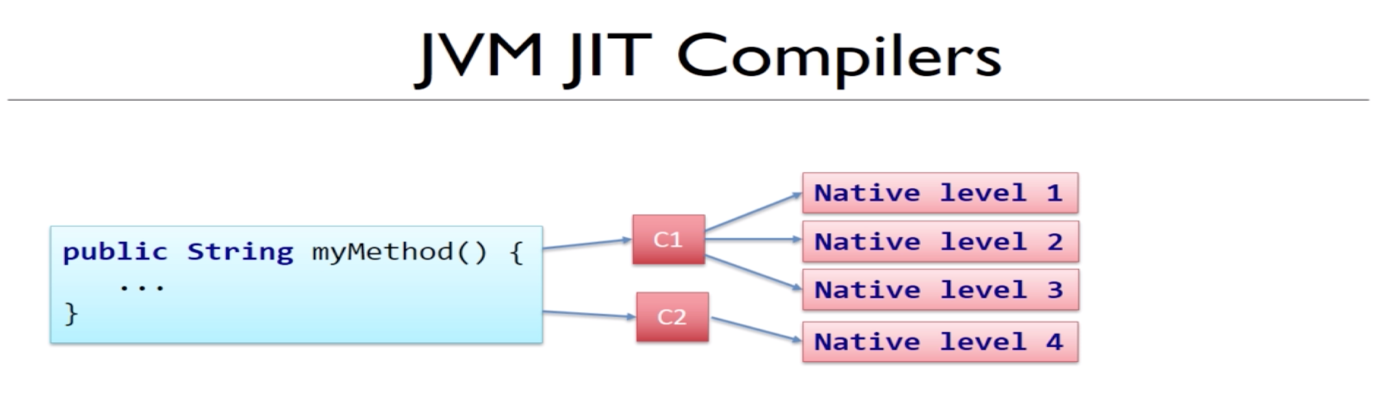
1 to 4 🡪 Means that deeper level of compilation has happened.

If you check the below output, our prime method was compiled at the highest possible level of compilation.

Text

Description automatically generated

* **The C1 & C2 compilers and logging the compilation activity:**

****

Now there are actually two compilers built into the Java Virtual Machine called C1 & C2.

C1 is able to do the first three levels of compilation. Each has progressively more complex than the last one.

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***”.

Code cache – Special area of memory because that will be the quickest way for it to be accessible.

The flag we used above prints the compilation details to console, there is an another way to collect logs as a file. And that file contains more information when compared with the details logged directly to console.

java -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation Main 5000

Once the code is executed you can find a log file.

* **Tuning the code cache:**

If a given class or application has identified many methods to be level4 compilation by JVM, the space we get for “code cache” is limited, in such cases we have to remove the method which is not used for the code is running, again we have to add the removed code whenever it is required.

So in case if the level4 compilation is more, we have to increase the size of “Code cache” which will improve the performance of the application.

We will get below error whenever the code cache is full.

Graphical user interface, text

Description automatically generated

We can find the size of “***Code cache***” by below JVM flag.

Graphical user interface

Description automatically generated

See the sample output below:

Text

Description automatically generated

Max size of the code cache is decided based on the java version we are using.

Less than Java1.7 version 🡪 32(32Bit JVM) to 48(64Bit JVM) MB

Java 1.8 or above and 64 bit JVM 🡪 248 MB

We can increase code cache size by setting following three flags.

Graphical user interface, text, application

Description automatically generated

IntialCodeCacheSize 🡪 Size at initial stage.

ReservedCodeCacheSize 🡪 Maximum size.

CodeCacheExpansionSize 🡪 How much extra space should we add

each time we grow the code cache.

Note: Each of above values can be defined in bytes, kilo and mega bytes.

java -XX:ReservedCodeCacheSize=28m -XX:+PrintCodeCache Main 5000

Now you can see the newly set code cache size.

* **Remotely monitoring the code cache with JConsole:**

There is an application used to monitor the code cache our application uses and which is running remotely.

The application name is JConsole and it comes with part of JDK.