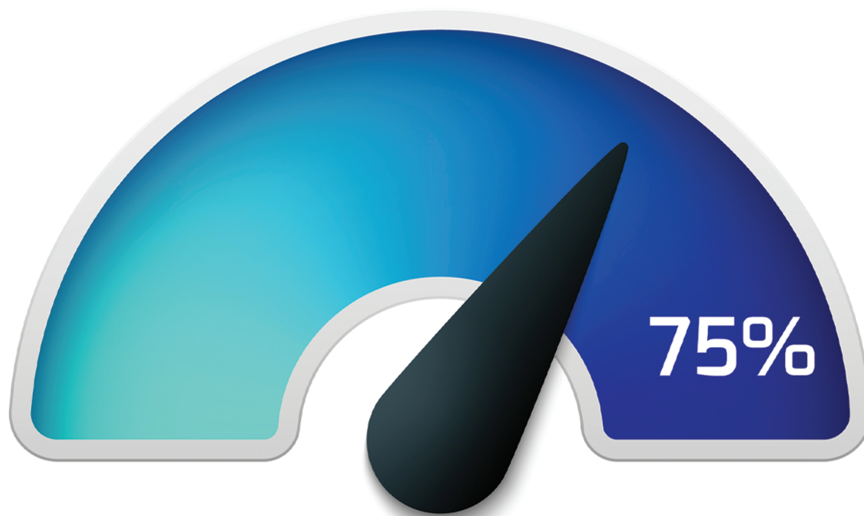
JAVA APPLICATION PERFORMANCE AND MEMORY MANAGEMENT

JAVA VERSIONS 8, 11 AND 17



A practical guide to improving performance for JVM applications

Matt Greencroft

Chapter 2 – How the Java Virtual Machine executes code

The history of Java

Before you can start to optimise an application, you need to have at least a basic understanding of what the JVM actually does when it runs your code. It can be helpful to start with an overview of the history of the language.

Prior to the creation of Java, every programming language could be categorised as being either an interpreted language or a compiled language. Interpreted languages are ones where the code needs a specific piece of software (an interpreter) for the user to run it. For example php, a popular language for the creation of websites, needs a webserver such as Apache. Javascript can be run in a web browser. For these languages it's the raw code that is provided to the user.

The alternative, the compiled languages, are those which go through an extra step before they can be distributed – the process of compilation. This results in the user being given an executable file – for example in Windows that would be a file with the .exe extension. The user doesn't get the raw code, and can run the application without needing any additional software.

In the early 1980s, when the internet was in its infancy, there were advantages and disadvantages to each approach. The main advantage of compiled languages were that the code would run faster than for an interpreted equivalent, and for businesses who had invested a lot of intellectual property into their code, they did not need to expose that to users, who would be able to copy and use their proprietary algorithms.

However the process of compilation is taking the code and compiling it for a specific operating system. So code which has been compiled to run on a Windows computer, won't run on a Mac or Linux machine (at least without additional software). Further, because the architectures of different operating systems differ significantly, it is normally the case that the code-base needed for a Windows program will be different to that needed for a Mac application. So a company wishing to sell software to both Windows and Mac users might need to double the effort of code creation and management.

The main benefit of interpreted languages is that the same code can be run on any device, as long as there is an interpreter available for that device. For example if I provide you with some Javascript, you should be able to run it on any computer you like that has a

modern browser – and in fact you can choose the interpreter (which browser) as well. This concept is known as "write once run anywhere" – and has the obvious benefit of lowering the duplication of effort required to create code suitable for multiple different systems. The disadvantage of interpreted languages is that you are providing the client (and therefore potentially your competitors) with open access to your code.

In 2022, this seems like a strange idea to claim as being a disadvantage. With the prevalence of open-source code, and software as a service, meaning that you can now manage the deployment of your code and sell access to it to the user, keeping interpreted code private is less of an issue. But when the usual way to distribute applications was using CD-Roms, as it was in the early 1980s, this was an important factor for businesses selling software.

One of the goals with the creation of Java (more strictly the virtual machine) was to try and provide a "best of both worlds" scenario. It was to be a language which can be written once and run anywhere, as long as there is a JVM for the operating system in question. At the same time it would go through a compilation process, and so gain at least some of the speed benefits of this process.

That's the history lesson complete, now let's look at where we are today:

Just In time compilation

You're hopefully aware that the code we write in Java is compiled by the Java compiler into "bytecode". These compiled files have a .class extension, and are optionally packaged together into a jar, war or ear file.

When we actually execute our application, using the Java command, the bytecode is then run by the Java Virtual Machine (JVM). We can think of this as the JVM is the interpreter of bytecode.

The JVM however is not simply interpreting the bytecode. It contains a number of features and complex algorithms to make it more efficient than more traditional code interpreters. If you were writing code in a language such as PHP, which is not compiled but is interpreted at runtime by a web server, each line of code is only looked at, analysed, and the way to execute it determined, as it is needed. Within the JVM, it's a much more complicated process – some of that work can be done in advance, and we will explore some of the detail of what the JVM can do in this and the next chapter.

So the JVM is not being asked to run Java code, but rather to run bytecode. In fact any language which can be compiled to JVM compatible bytecode can be run on the JVM. As a result pretty much everything we talk about in this book will apply not just to Java, but to Scala, Kotlin, Groovy, Clojure and any other JVM language.

At the end of this book, in the final chapter, we are going to look at some real bytecode, and we will compare bytecode produced by Java with bytecode produced by another JVM language.

But that's a long way off, and right now we're going to focus on what happens when we ask the JVM to actually run some bytecode.

Initially the JVM acts like any other interpreter, running each line of code as it is needed. However by default, this would make the code execution somewhat slow, so to help get around this problem of slower execution in interpreted languages than compiled languages, the JVM has a feature called Just In Time Compilation or JIT compilation for short. The JVM will monitor what branches of your code are run the most often, which methods or parts of methods, specifically loops, are executed the most frequently. The JVM will then decide that if a particular code block[3] is being used a lot, code execution would be speeded up if that method was compiled to native machine code… and it will then do this extra step.

At this point, some of your application is being run in interpreted mode, and some is running as native machine code. The part that has been compiled to native machine code will run faster than the interpreted part. Just to be clear, by native machine code, we mean executable code that can be understood directly by your operating system. So if you are running this application on Windows, part of the bytecode has been compiled into code that can be understood natively by the Windows operating system. If you were running on a Mac, then the JVM would have compiled this to native Mac code. The native Windows code and the native Mac code would of course be different – they are not compatible. So the Windows JVM is able to compile bytecode into native Windows code, and the Mac JVM is able to compile bytecode into native Mac code.

This process of native compilation is completely transparent to the user, but it has an important implication – your code will generally run faster the longer it is left to run.

3 Any sequence of bytecode can be compiled to native machine code. From our point of view as programmers we can think of this as meaning any method or any code block. This is not a completely accurate definition, but it's good enough to allow us to understand what the JVM is doing.

That's because the JVM can profile your code and work out which bits to optimise by compiling them to native machine code. So a method run multiple times every minute is likely to be JIT compiled quite quickly, but a method run once a day might not ever be JIT compiled.

The process of compiling, converting bytecode to native machine code, is run in a separate thread – the JVM is of course a mutli-threaded application, so the threads responsible for executing the code won't be affected by the extra thread doing the JIT compiling… and it also doesn't stop or pause the application in its process of running. While compilation takes place, the JVM will continue to use the interpreted version. Once the compilation is complete, and the native machine code version is available, the JVM will then seamlessly switch to the compiled version.

If your application is heavy on processing, using all the available CPU resources, you could potentially see a temporary reduction in performance while JIT compilation is taking place, although it would only be in the most critical and high power processing applications that you might notice this, and even then, it's going to be worth the momentary slight dip in processing power to get the benefit of the native code version of your method in the future.

An impact of JIT compilation is that if you are assessing the performance of a particular piece of code, you actually need to think about when you do that assessment. Suppose you're trying to determine how long a process takes to run, and you want to see which of two alternative methods will run more quickly.

If you assess the performance as soon as your application starts, you might find that you get different results than if you assess it after your application has been running for a short while. You need to think about whether you are assessing the performance of the code before it has been natively compiled, or after. We'll actually come back and consider this point in some more detail in chapter 15, when we investigate how to actually measure performance.

Using JVM flags

As a programmer it can be interesting to know which methods or code blocks are being compiled to native machine code. There's a JVM flag we can use to find this out.

As this is the first time we have used the term "JVM flag" it requires a simple definition – this means an argument that we can supply to the Java Virtual Machine at runtime, which

will either provide a configuration setting, or ask the JVM to provide some additional output.

To see how a JVM flag can be used, and in particular how it can be used to determine which parts of an application are being compiled to native machine code, we'll use a simple Java Program. The code for this program can be found in the Github repository for this book in the "Chapter 02" folder.

This simple program will generate a sequence of prime numbers[4] of a desired length. This is absolutely not written in any kind of production standard way; it has been purposely created to not be optimal in terms of coding.

When we run this application, we will need to provide an integer command line argument. This number will be the volume of prime numbers we would like the application to generate.

The code being executed is contained in a class called PrimeNumbers. When the application runs, the generateNumbers method is called, and it receives as a parameter the desired number of primes provided as the command line argument.

This method instantiates an ArrayList, and then within a loop, determines each of the required prime numbers in ascending order, starting with the number 2. Each number, when calculated is added to the list.

There are 2 methods being called as part of this process: the isPrime method will check if a number is prime or not, and the getNextPrimeAbove method executes a loop until the next prime number is found.

I suggest that you run the application before we go any further, just to see it working. I recommend that you run it from the command line, as we will be entering command line arguments and JVM flags, which is going to be straightforward if we use a command prompt or terminal window. If you would rather use an IDE, you will find details of how to enter command line arguments and JVM flags in Appendix 2 at the end of this book.

Having opened a command prompt or terminal window and navigated to the folder where the .Java files for this project can be found, entering the following will compile the 2 Java files into .class files.

---

4 A prime number is any integer which is only exactly divisible by itself and the number 1.

```
> javac *
```

The application can then be executed using the following example to create 15 prime numbers:

```
> java Main 15
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

The PrintCompilation flag

We are now ready to use our first JVM flag. This is simply a command line argument to the Java application. The flag we are going to use in this first example is:

```
-XX:+PrintCompilation
```

Most of the flags we will be using in this book follow this standard format. They start with "-XX". This is to signify that it is an advanced option. Next there is a colon, and then either a "+" or a "-". This is to indicate whether the option is to be switched on ("+") or off ("-"). And then finally we get the name of the option.

We'll see quite a few of these flags as we proceed through the book. Supplying this flag means we want to tell the JVM to run our application with some feature or behaviour that is not going to be the default.

Let's run the code again with this flag on. It is important to note that:

- the flags are case sensitive and must not contain any spaces
- the flags must be entered immediately after the java command, and before you name the class containing the main method to be executed.

In the code output which follows, the values you get may not match the example provided here. The example provided is generated using Java 8. Running the code with other versions of Java will result in different output.

```
> java -XX:+PrintCompilation  Main 15
71   1    3   java.lang.String::hashCode (55 bytes)
74   2    3   java.lang.Object::<init> (1 bytes)
75   4    3   java.lang.String::charAt (29 bytes)
76  11  n 0   java.lang.System::arraycopy (native)   (static)
77   3    3   java.lang.AbstractStringBuilder::ensureCapacityInternal (27 bytes)
78  13    1   java.lang.Object::<init> (1 bytes)
79   2    3   java.lang.Object::<init> (1 bytes)   made not entrant
79  10    3   java.lang.AbstractStringBuilder::append (29 bytes)
81   5    3   java.lang.CharacterData::of (120 bytes)
81  17    4   java.lang.String::charAt (29 bytes)
82   8    3   java.lang.Character::toLowerCase (9 bytes)
83   4    3   java.lang.String::charAt (29 bytes)   made not entrant
84   9    3   java.lang.CharacterDataLatin1::toLowerCase (39 bytes)
85  15    3   java.io.WinNTFileSystem::isSlash (18 bytes)
86  16 s  3   java.lang.StringBuffer::append (13 bytes)
87  18    3   java.lang.String::indexOf (70 bytes)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
89  20    1   java.lang.Integer::intValue (5 bytes)
90  14    3   java.lang.StringBuilder::append (8 bytes)
```

This output might seem a little daunting at first. What has been provided is every process of compiling that has been carried out by the JVM as our application runs.

The first column is the time in milliseconds since the virtual machine started. Next is the order in which the method was compiled. The fact that some of these are out of order means that some parts took longer to compile than others. This could be, due to multi-threading prioritisation issues, code complexity, or length of code being compiled.

We then have some space (actually a few columns), most of which is blank, but we do have one containing the letter “n”, and one containing the letter “s”.

The “n” means that this is a native method. The “s” means it is a synchronised method. You will in other examples see “!” in this section, which refers to exception handling, or “%” which means that the code has been native compiled and is now running in a special part of memory called the code cache. A “%” next to a method means it is running in the most optimal way.

The next column has a number from 0 to 4. This refers to the compilation tier that has taken place, which we’ll explore next.

And then finally is the line of code that has been compiled.

Executing the application with just 15 numbers being calculated, has resulted in a relatively low level of compilation. All the compilation is being carried out on core Java library methods (such as the String's charAt method). None of our code is appearing here.

If we run the application again using a much larger set of numbers to calculate, such as 5000, we'll get some more interesting output – you may wish to comment out the line of code which prints the output to the console this time. Note again that your exact output may differ from the version shown here but the key parts will be very similar.

```
>java -XX:+PrintCompilation  Main 5000
...
86   22     3    PrimeNumbers::isPrime (35 bytes)
87   20     1    java.lang.Integer::intValue (5 bytes)
87   27 %   4    PrimeNumbers::isPrime @ 2 (35 bytes)
88   25     3    java.lang.Integer::<init> (10 bytes)
89   23     3    java.lang.Number::<init> (5 bytes)
90   26     3    java.lang.Integer::valueOf (32 bytes)
91   28     4    PrimeNumbers::isPrime (35 bytes)
91   21     1    java.lang.Boolean::booleanValue (5 bytes)
92   24     1    java.util.ArrayList::size (5 bytes)
93   22     3    PrimeNumbers::isPrime (35 bytes)   made not entrant
96   30     3    PrimeNumbers::getNextPrimeAbove (43 bytes)
98   31     3    java.lang.Boolean::valueOf (14 bytes)
98   29     3    java.util.ArrayList::add (29 bytes)
99   32     3    java.util.ArrayList::ensureCapacityInternal (13 bytes)
100  33     3    java.util.ArrayList::calculateCapacity (16 bytes)
101  34     3    java.util.ArrayList::ensureExplicitCapacity (26 bytes)
102  14     3    java.lang.StringBuilder::append (8 bytes)
103  19     3    java.lang.String::indexOf (7 bytes)
165  35     4    PrimeNumbers::getNextPrimeAbove (43 bytes)
171  30     3    PrimeNumbers::getNextPrimeAbove (43 bytes)   made not entrant
412  37     3    java.lang.String::getChars (62 bytes)
413  48     1    java.lang.String::length (6 bytes)
...
```

I have excluded some of the output from the version provided here, to allow us to focus on the more interesting parts. Within the column that contains the method names we now have some of our methods appearing – the isPrime method and the getnextPrimeAbove method. And we have a % symbol appearing here too.

The column containing the numbers between 0 and 4 is the compilation tier. It tells us what kind of compiling has taken place. A zero means no compilation, the code has just been interpreted. The numbers 1 to 4 means that a deeper level of compiling has happened.

The JVM's compilers

There are actually 2 compilers built into the JVM, called C1 and C2. The C1 compiler is able to do 3 levels of compilation, each progressively more complex than the last one. The C2 compiler can undertake the 4th 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 when executed.

For any method which has a number 1 to 3, the code has been compiled using the C1 compiler. The higher the number the more "profiled" the code has been. If the code is called enough, then we reach level 4 and the C2 compiler is used instead – and this is more optimised than the C1 compiler.

The higher the compilation tier level, the more optimised the compiled code should be. The JVM doesn't just optimise everything to tier 4 because there is a trade off – it only optimises the most frequently called code, and if that code is not doing anything complex, there may be no benefit from a deeper level of compilation.

In our example, the isPrime method first got compiled at tier 3, then later it got compiled to tier 4. At this point we also see a "%", meaning that as well as being compiled in the most performant way, this has been placed in the code cache for even quicker access. The virtual machine determined that this method is being executed so much, and is so important to our application, that it needed to be compiled with the C2 compiler and placed in code cache for the best possible performance.

This first flag is outputting the information to the console. There is an alternative flag which will output the information to a file, and this gives even more information.

The LogCompilation flag

This flag is -XX:+LogCompilation. However to use this you also need to specify the flag -XX:+UnlockDiagnosticVMOptions, and that flag must come first in the list.

If you do not specify a filename, the data will be written to a file called hotspot.log in the current folder. Alternatively you can provide a file name with –XX:LogFile=fileName.log.

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

The log file is in XML format. The OpenJDK project provide a description of the format of the file (which they state is subject to change)[5]. For users of Java 15 and above, they also provide a tool to help read the file.

Having an understanding about how the JVM runs our code provides us with our first opportunity to improve performance.

When code has been compiled to tier 4 using the C2 compiler, as we have seen it is placed in the code cache. This is an efficient area of memory. But this code cache has a limited size and if there are lots of methods that could be potentially compiled to this level, then some may need to be removed to make space for a new method to be added. That initial method might subsequently be recompiled and re-added later on.

In other words in larger applications, with lots of methods that could be compiled to tier 4, over time some methods might be moved into the code cache, then moved out, then moved back in again later on. When this happens, the default code cache size might not be sufficient, and increasing the size could lead to an improvement in your application's performance. You might even see logging messages in the following format as your application runs:

```
VM warning: CodeCache is full. Compiler has been disabled.
```

This is telling us that the code would run better if another part of it could be compiled to native machine code, but there is insufficient room for it in the cache. Furthermore all the code that is in the cache is actively being used, so no other part of the cache can be easily cleaned up. This is a warning message – it doesn't stop your application running, but it does mean that it's not running in an optimal way.

The PrintCodeCache flag

We can find out about the size of the code cache using the -XX:+PrintCodeCache flag.

```
>java -XX:+PrintCodeCache Main 5000
CodeCache: size=245760Kb used=1150Kb max_used=1162Kb free=244609Kb
bounds [0x000001e49e140000, 0x000001e49e3b0000, 0x000001e4ad140000]
total_blobs=289 nmethods=51 adapters=152
```

5 https://wiki.openjdk.java.net/display/HotSpot/LogCompilation

```
compilation: enabled
```

In the example application, with 5000 prime numbers, on my computer, we are told that the code cache is approximately 24MB, and under 1MB has been used. There is definitely nothing to worry about here, but clearly in a more complex application, if you see the max_used number approaching the size, then a tweak might be helpful.

Tuning the code cache

The default maximum code cache size is normally 48MB. There are 3 flags we can set to alter this. -XX:InitialCodeCacheSize is the size of the code cache when the application starts. This is normally quite small – the default size varies based on your computer's available memory but is often just 160kb.

-XX:ReservedCodeCacheSize is the maximum size of the code cache. It can be allowed to grow up to this size. As we have already said the default is normally 48MB.

-XX:CodeCacheExpansionSize indicates how quickly the code cache should grow – as it gets full, how much extra space should we add to the code cache each time it is expanded.

Growing the size of the codecache can be a slightly slow process, so setting the initial code cache size to be the same as the reserved code cache size will probably mean a slightly slower startup but then no subsequent pauses, while the application is running, for the code cache to be resized.

Although we don't need to do it for our application, let's see what this would look like, as these flags have a slightly different format to the ones we have used so far:

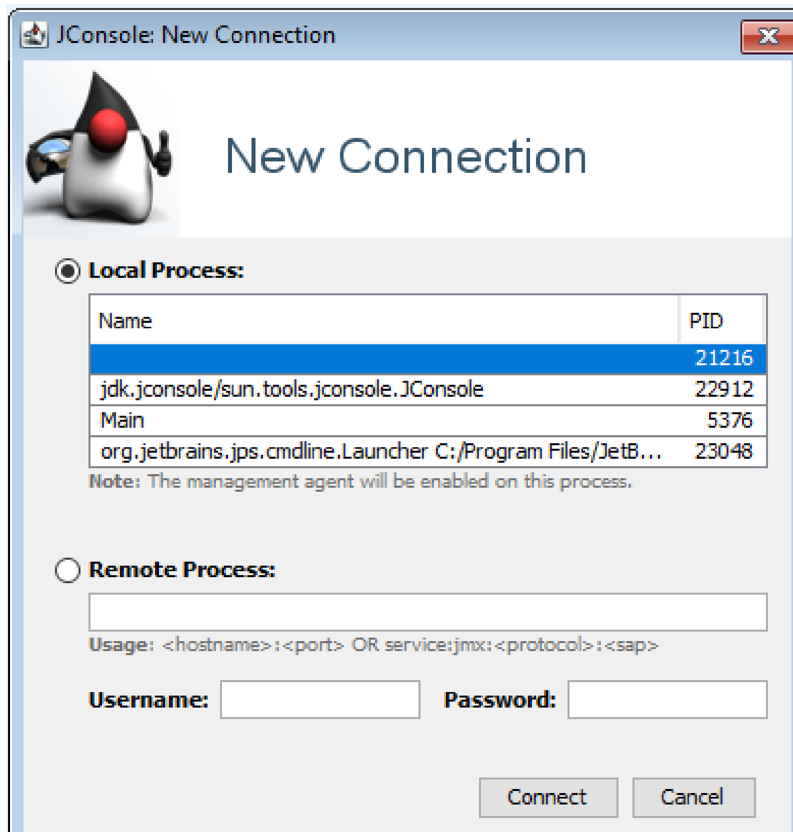
```
> java -XX:ReservedCodeCacheSize=20M -XX:+PrintCodeCache Main 5000
```

We don't use the "+" or "–" to indicate whether this flag is to be turned on or off. Rather we provide the flag name, then an equals sign then a value. If we just put a number in here, that means bytes. We can follow this with a K, to mean kilobytes, an M to mean megabytes, or a G to mean gigabytes. Actually we can't use a G in this example – the maximum size of a code cache is 48M. The letters K or M can be entered in upper or lower case.

There are some further flags that relate to the tuning of the code cache which you will find in the Java reference documentation[6].

Monitoring the code cache

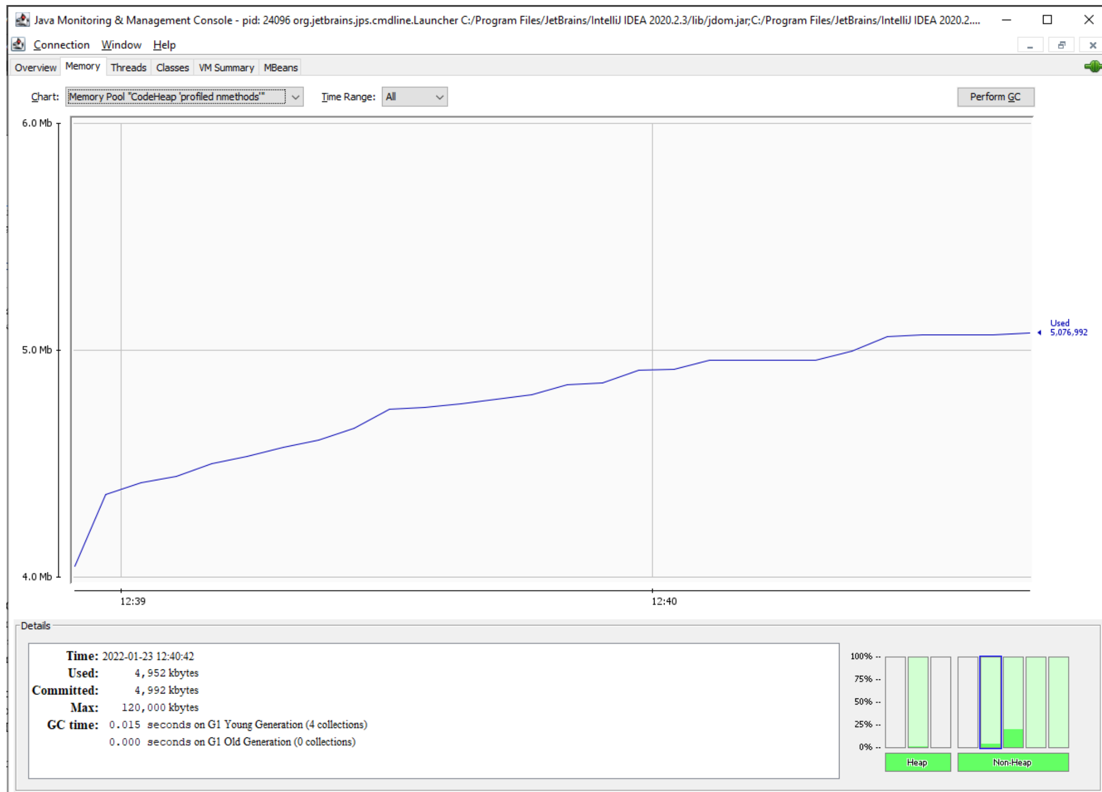
One of the tools that comes with the JDK can be used to monitor the code cache size graphically over time – the tool is called jConsole, and can be found in the bin folder of the JDK. Simply double click the jConsole file to run it.



When jConsole first starts, we need to connect it to an application to monitor. If you are using an IDE this will be a running Java application that you can connect to. You may receive a warning that the connection is insecure, which is perfectly safe to ignore if you are connecting to a process on your local machine.

6 https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/codecache.htm

After connecting to any application, select the memory tab and then change the selected chart to either “Memory pool Code Cache” if you are using Java 8, or “Memory pool Code Heap profiled methods" if you are using Java 11+.



If you run the project now from within the IDE (refer to Appendix 2 if you are not sure how to enter command line arguments within an IDE), we should see the code cache changing. Remember we’re monitoring our IDE here, not our code, so this is our IDE’s use of the code cache.

This tool might give you a useful way to monitor that an application isn’t growing its code cache out of control over time, as you can potentially leave this tool running. However be aware that doing this, connecting to a running application with jConsole, actually means that we are running a tool that is interacting with the JVM, and this will alter the performance of your application. In fact it requires the JVM to run extra code to communicate with jConsole, and in turn some of this code will be compiled… to see what I mean, let’s connect jConsole to our application rather than our IDE.

To do this I need to have our application run for long enough to make this meaningful, so I'm going to make it pause for 20 seconds when it first starts, before it commences doing any real work. This should give us enough time to connect to the application before it starts calculating prime numbers. The code to pause is currently commented out in the main method – simply uncomment it to execute in this way.

We can then run the application with the -XX:+PrintCompilation flag and after starting it, connect to it from jConsole.

```
> java -XX:+PrintCompilation Main 5000
```

As soon as you connect to the application with jConsole, you will see a large number of additional lines of compilation information output on the console. This is the compilation work that the virtual machine is doing by running the extra code needed to communicate with jConsole.

What we have seen in this chapter is not quite the complete story. In the next chapter, we'll see that the way the 32 bit and 64 bit JVMs work is a little different, and that we can also tune the compiler that we have been learning about in this chapter to work differently…. This will give us some more opportunities to optimise the performance of our applications at run time.