**Java Strings and Immutability**

**Background**

Java strings are immutable - when we allocate a string that the reference point to an object on the heap.

String are immutable and are allocated to the String pool

String abc = “abc” not the same as String abc = new String(abc);

It is possible for several String refences to point to the same heap reference value. When they are no longer being pointed at - what happens?

* In the old days, before Java 7, String data would build up in their String pool - and out of memory exception could happen.
* In Java 7 the String elements were moved to the heap - there is a distinct pool for the string objects, but this is Garbage collected

In our test framework we have been using String and reallocated them new values (e.g. with replace) – the concern was that this might cause in very long running scenarios (e.g. where we leave the framework running as a monitoring tool us to run out of memory).

This testing aimed to assess in Java 8, how much of problem this may be - it did show that StringBuilder and StringBuffer performed better than string.

Here is some theory discussion that someone has posted..

<https://stackabuse.com/string-vs-stringbuilder-vs-stringbuffer-in-java/>

This testing aligns with that theory, and proposes a migration for our test framework.

**Measuring Java Memory.**

To look at the heap size - stack size

java -XX:+PrintFlagsFinal -version | findstr /i "HeapSize ThreadStackSize"

We can measure the size for the string using the following

java -XX:+PrintFlagsFinal -version | findstr /i "String“

**Measuring things**

|  |  |
| --- | --- |
| 1 | -XX:+PrintFlagsFinal |
| 1 | -XX:+PrintStringTableStatistics |

There are some tools that part of the JDK - see your bin folder

* Jconsole
* Java VisualVM

**Limitations in the study**

It is assumed that the use of random number generated and the us a String array in arraylist is not distorting the results.

The allocation of data did not cause an out of memory exception to occur and JVM and garbage collection continued.

**Possible Solutions**

You can increase the size of the StringTableSize – use a large prime number ideally. If we want to increase the pool size in terms of buckets, we can use the StringTableSize JVM option:

* -XX:StringTableSize=16729
* Alternatively use StringBuilder and StringBuffer – (latter only if you need to be thread safe)
* Use of the .intern() method on the string to mark it ready for Garbage Collection

**Conclusions**

The testing showed on Java 8, that the behaviour observed a difference between using String and String builder / string buffer. The approach repeatedly allocated strings from

* A relatively small set of strings (approx. 450)
* or a large number of strings (suing large random strings)

Thus, at this stage, there is a desire to update DX CERES to replace allocated String with String Builder or String Buffer.

String buffer is synchronised, and is slower than String Builder - it took 10% longer to run across the randomised data.

Using the intern method was run on the limited strings (result 7) and the randomised data (result 6) and there were no specific observations made.

**Testing done**

The testing compared and contrasted using

* case A a small set of strings (likely to be in the heap)
* case B versus a large set of strings (random strings that were unlikely to be in the heap). The heap rapidly was rapidly filled with ‘unusual and unrepeated strings’ – 10 million times with 405 random strings.

This was tested

* **String and case A** Tested an allocated set of data using String allocations from a small set of strings - 450 strings - repeated many millions of times to allocate the strings - Result 1
* **String and case B** Tested an allocated set of data using String allocations from a large randomised set of strings - Result 2
* **Mutable String Builder and Case A** Tested an allocated set of data using StringBuilder from a small set of strings - 450 strings - repeated many millions of times to allocate the strings - Result 3
* **Mutable String Builder Tested and case B** an allocated set of data using String Builder from a large randomised set of strings - Result 4
* **Mutable String Buffer Tested and case B** an allocated set of data using String Builder from a large randomised set of strings - Result 5
* **String with intern() method added** – this allowed garbage collection to act upon the data. This case was done with a large set of randomised data - thus low chance of finding match in the heap. Result 6
* **String with intern() method added** – this allowed garbage collection to act upon the data. This case was done with a small set of known strings - thus high chance of finding match in the heap. Result 7

Test were carried out twice - one using JConsole and once suing J Visual VM

* JConsole test were for 4 million cycles
* JVisualVM were for 10 million iterations

Each cycle had 450 string allocations.

**Table of Results**

in Java \*, and then compare it to String Buffer and String Builder

Input data may be a small set of large set.

* Small set of repeated strings
* Large set of randomised of data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Input data | String Handling | CPU | Time to complete | Rough number of garbage collections |
| Result 1 | Small set | String - load a basic string | 30% | After 20 minutes only had 10% completed | Approx. 20 |
| Result 2 | Large randomised | String - load a basic string | 35% | After 20 minutes only 50% run | Doubled |
| Result 3 | Small set | String Builder | 35% | 7 | Approx. 50 |
| Result 4 | Large randomised | String Builder | 35% | 17 | Hundreds |
| Result 5 | Large randomised | String Buffer | 35% | 20 | Hundreds |
| Result 6 | Small set | String.intern() | 35% | 17 and 16 | Hundreds |
| Result 7 | Large randomised | String.intern() | 15% - but only 500k out of 20 million loops in 20 minutes | Greater than 2 hours | Hundreds (extrapolating) |

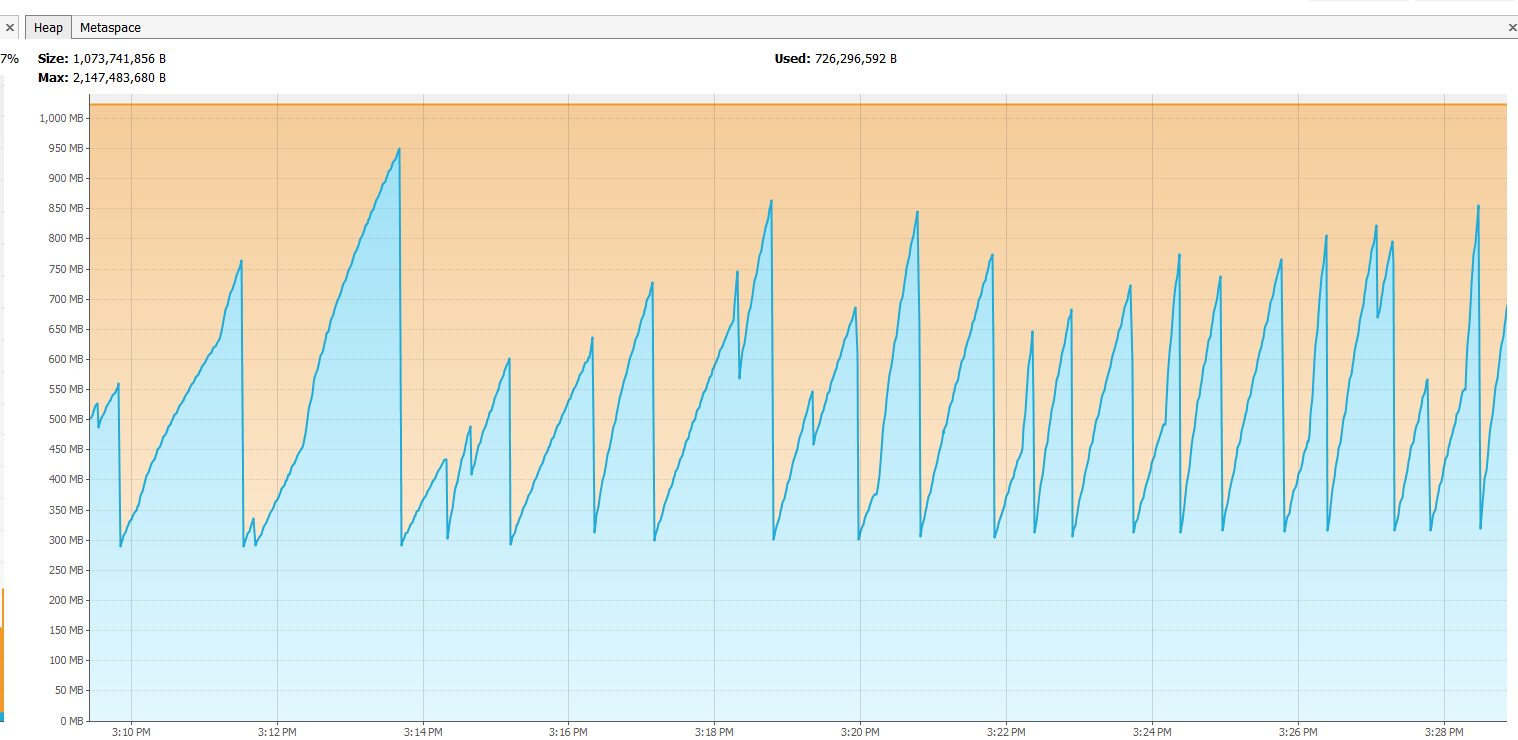
**String Buffer versus String Builder versus String Measurements**

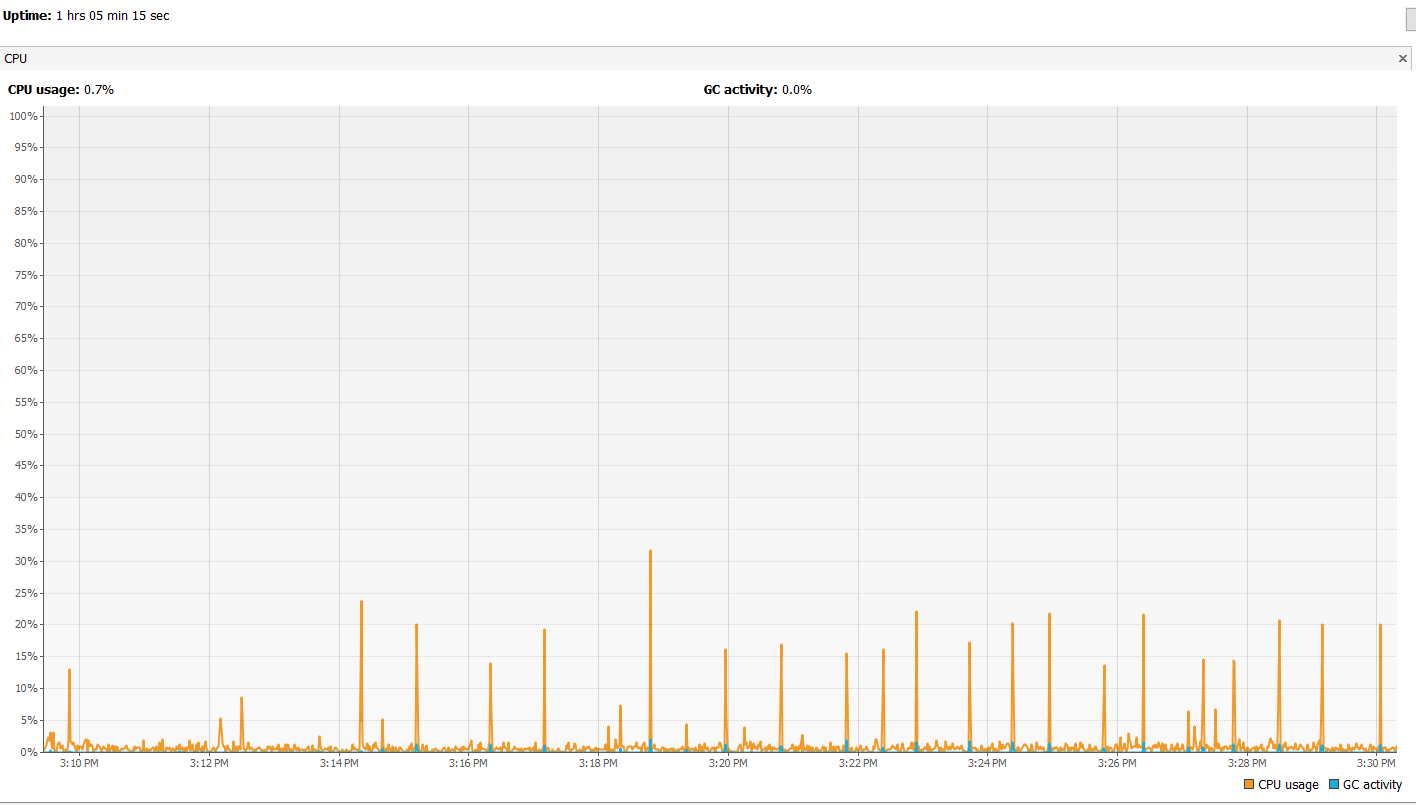
Code is stored at the following location for this .. held here … https://github.com/RobAllan27/JavaStringvsStringBuildervsStringBuffer.git

**Real Data**

**Result 1**

Using limited set of input strings i.e. can be found on the String pool - 10 million iterations on 3 minutes.



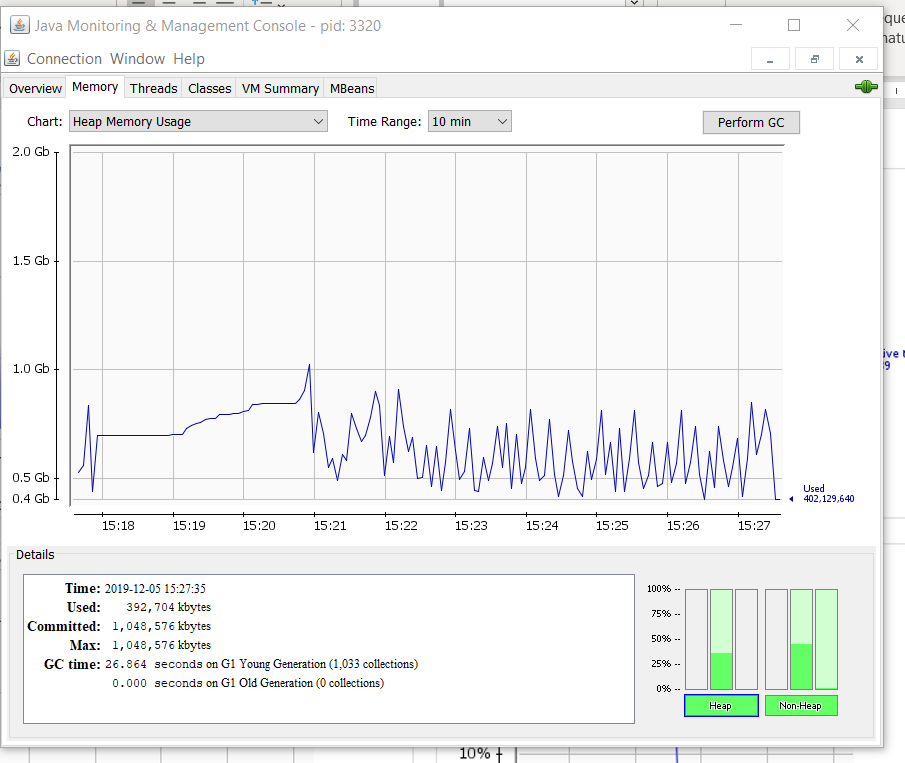


**With a fixed set of string and plain old string handling**

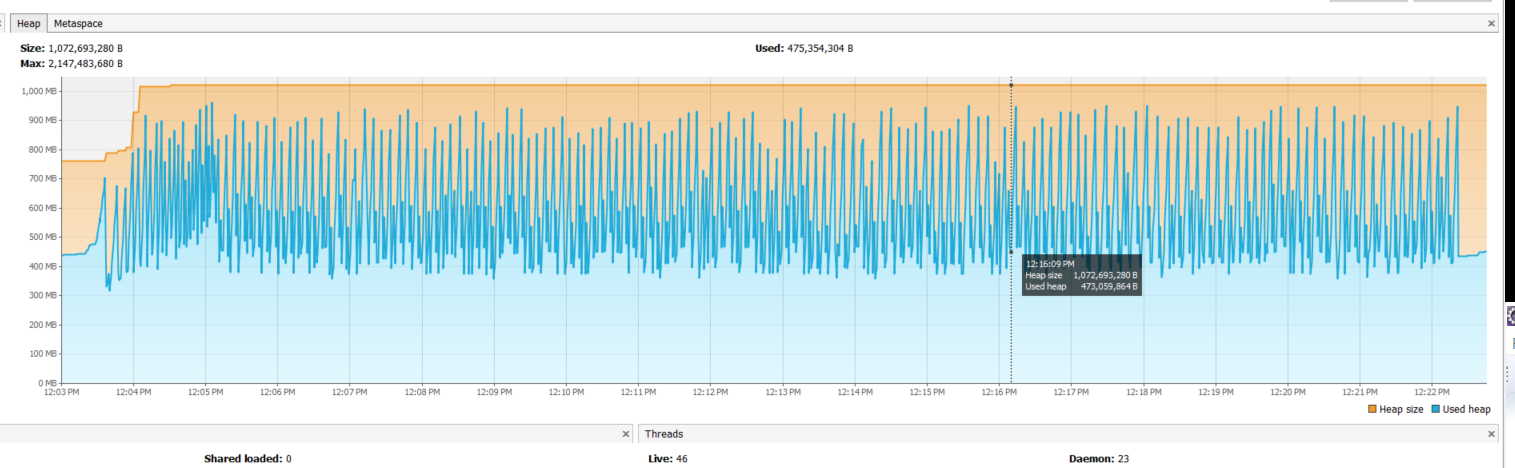
**Result 2**

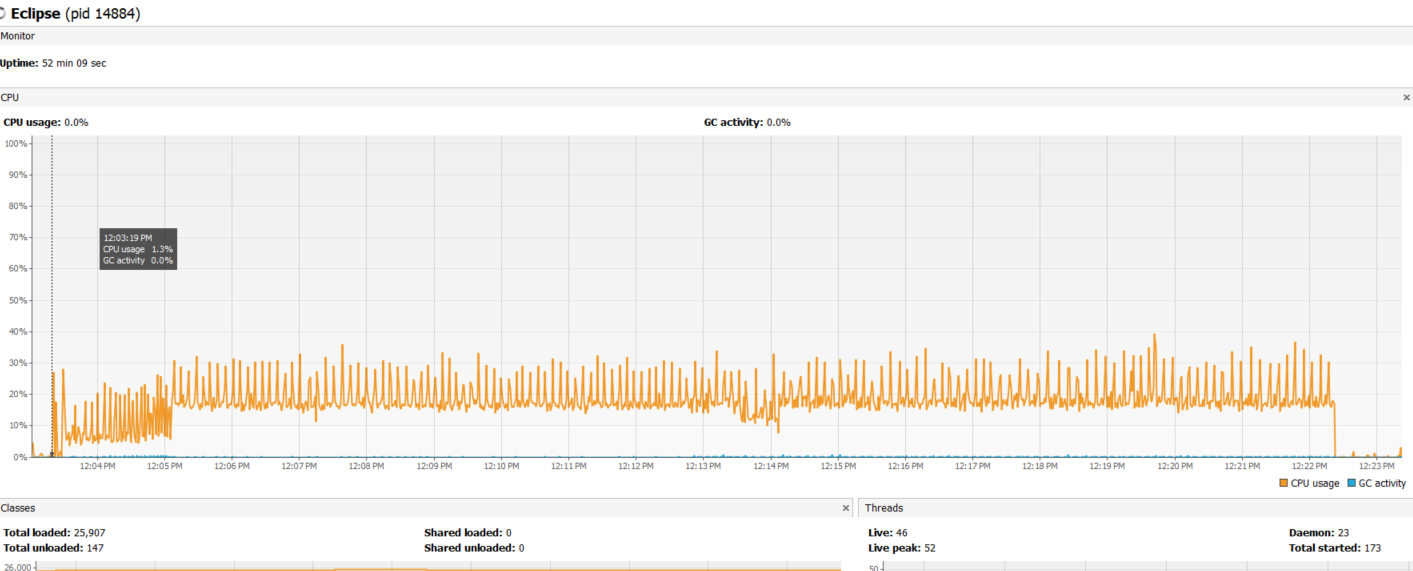
**Using the Random Strings**

4 millions iterations in approx 7 minutes

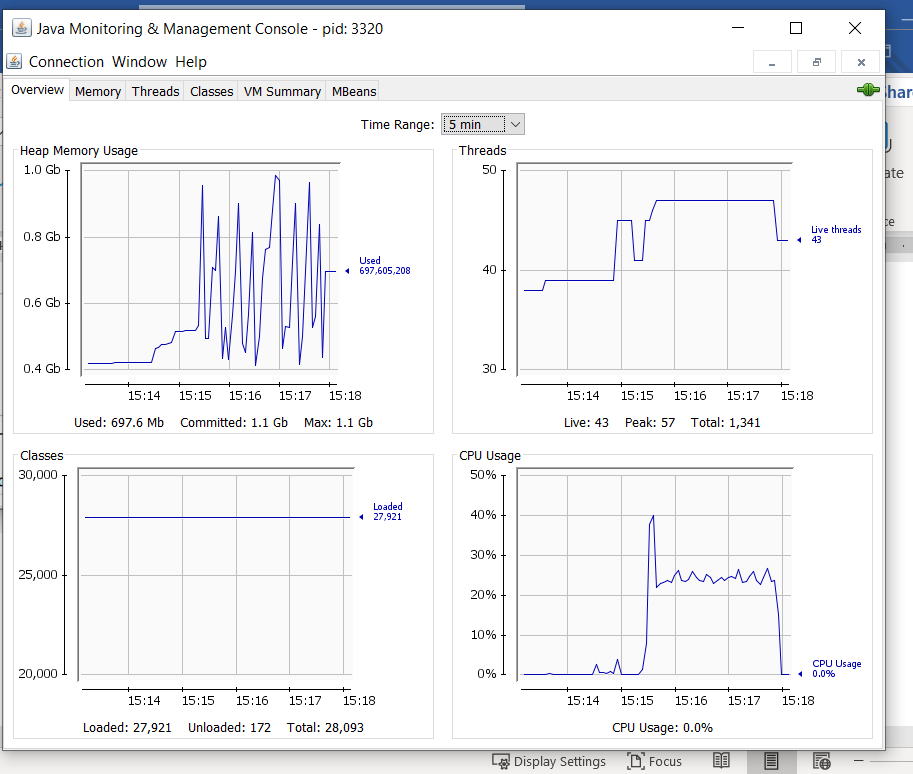


**For the case with the random strings - no string builder**

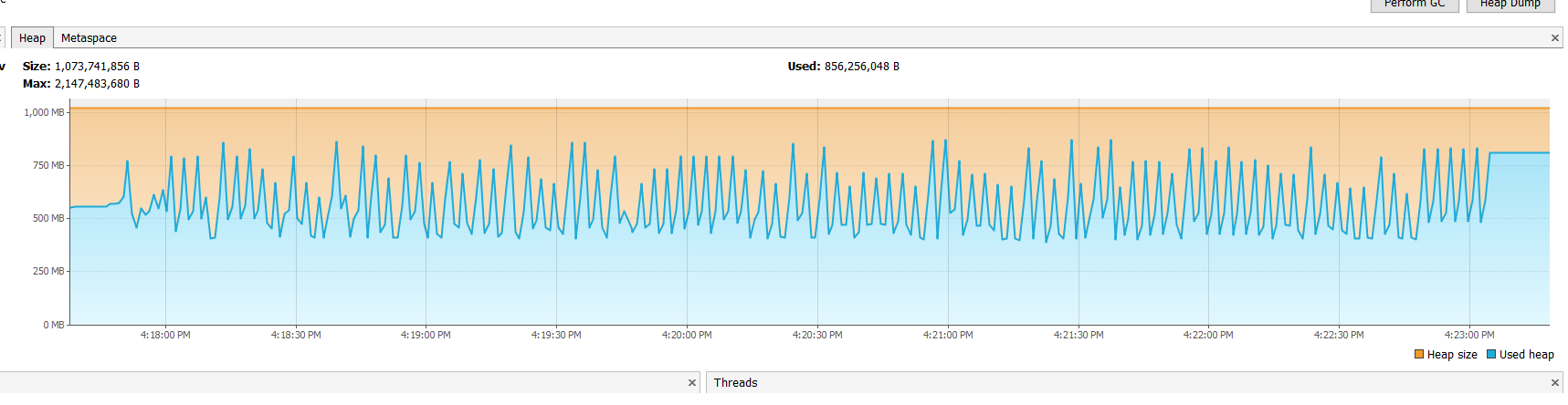




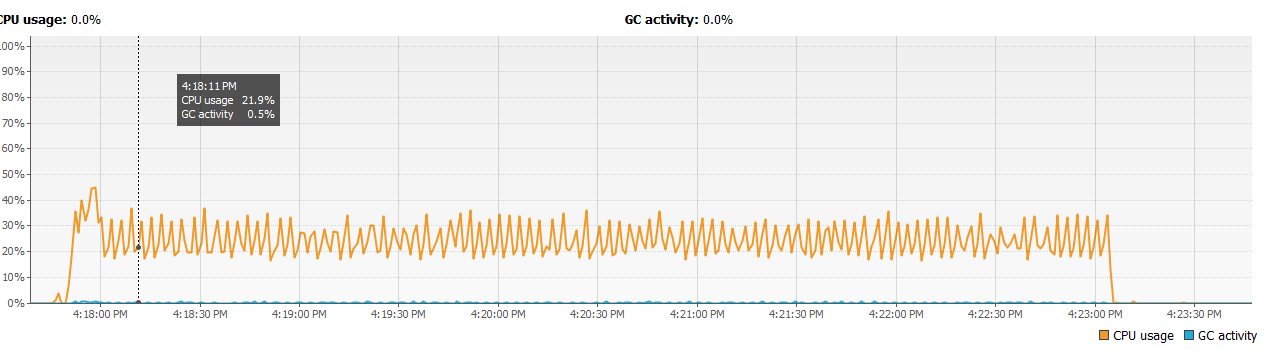
**Result 3 Using String Buffer**



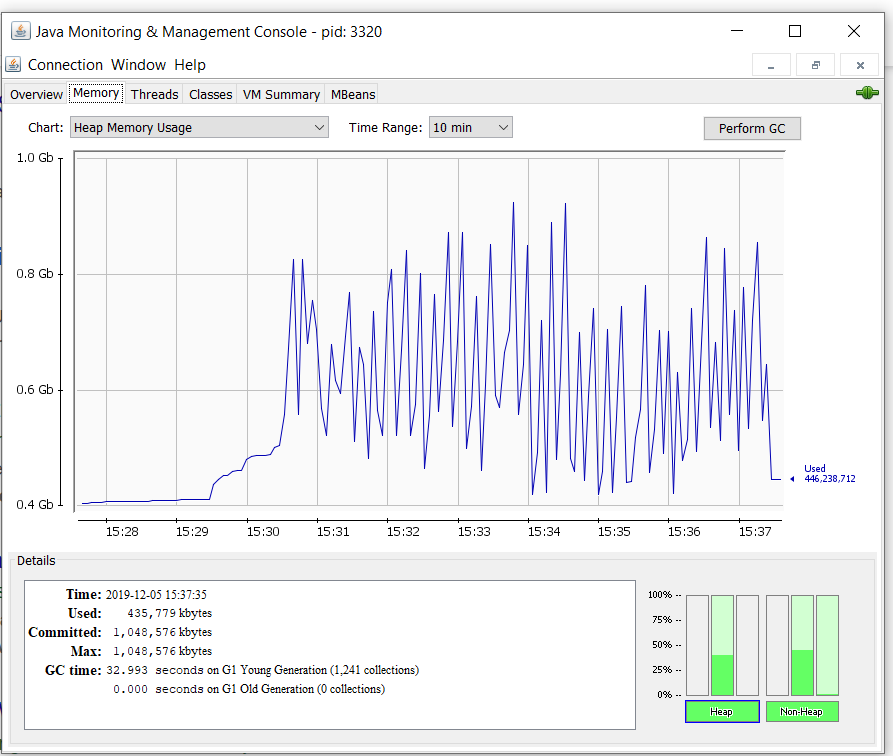
***For the basic case – loading strings with a StringBuilder***



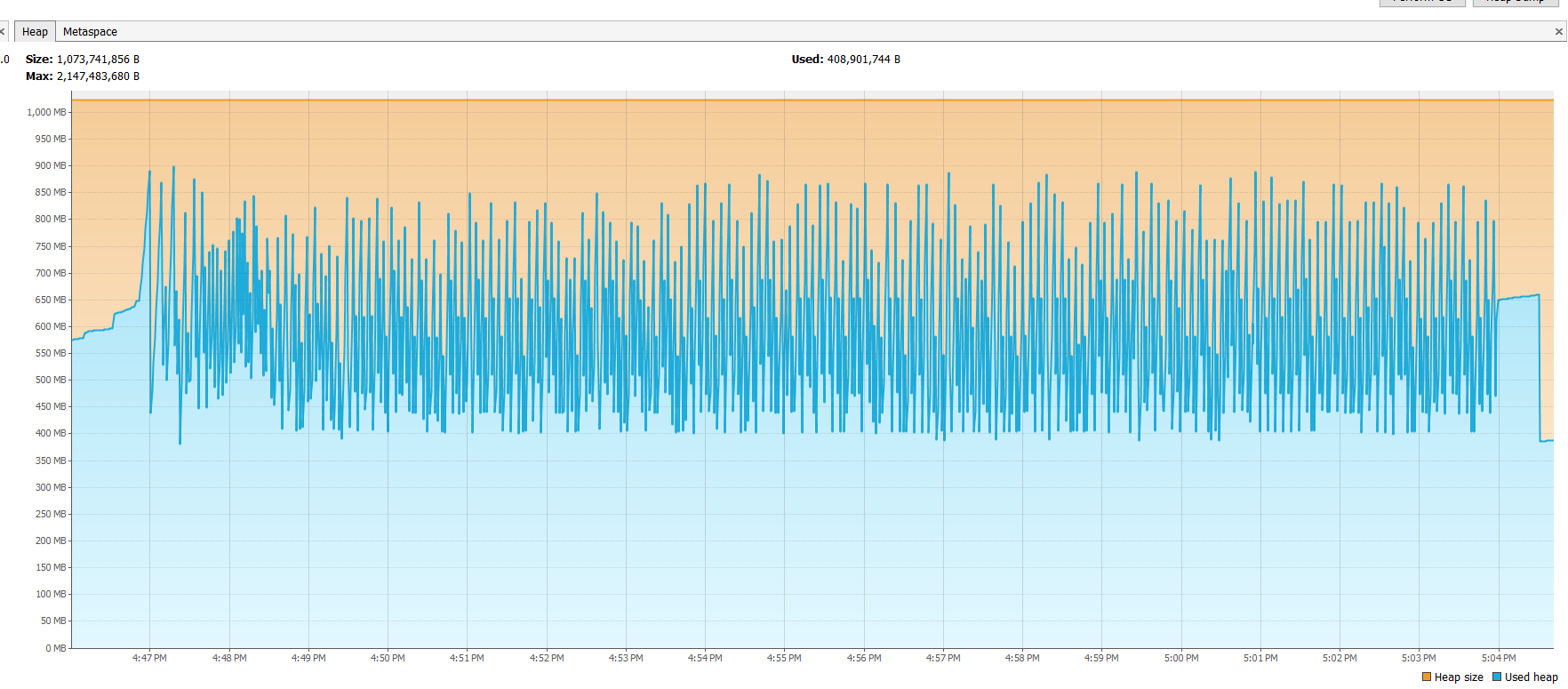
CPU and Garbage collection activities



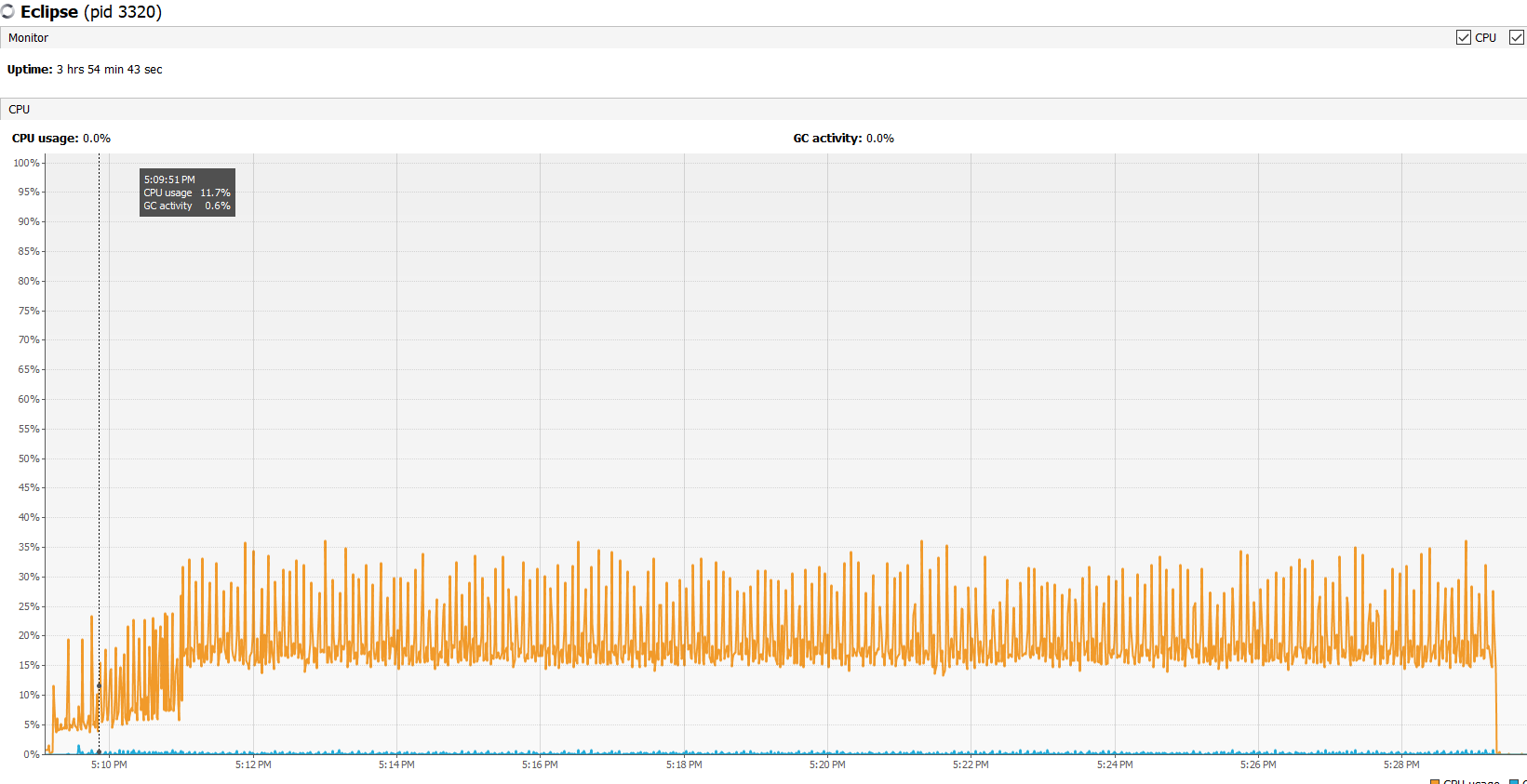
**Result 4**



With a string builder

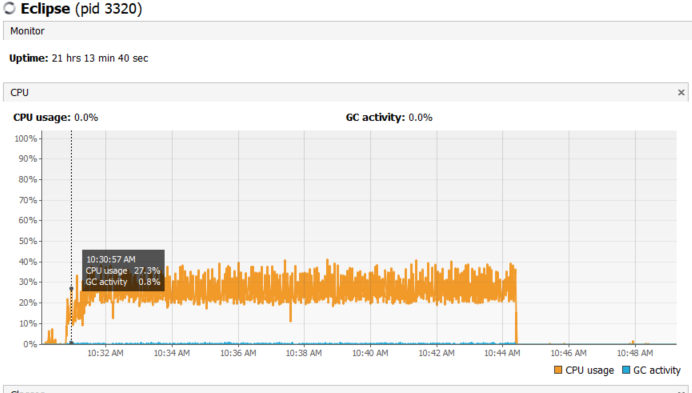


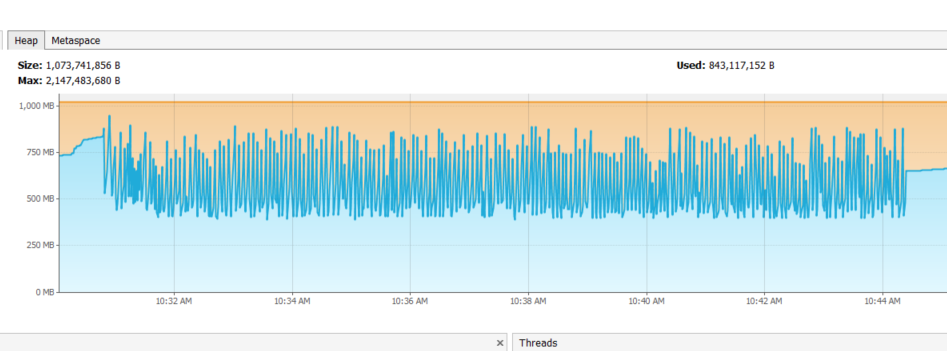
**Result 5 With a String Buffer**





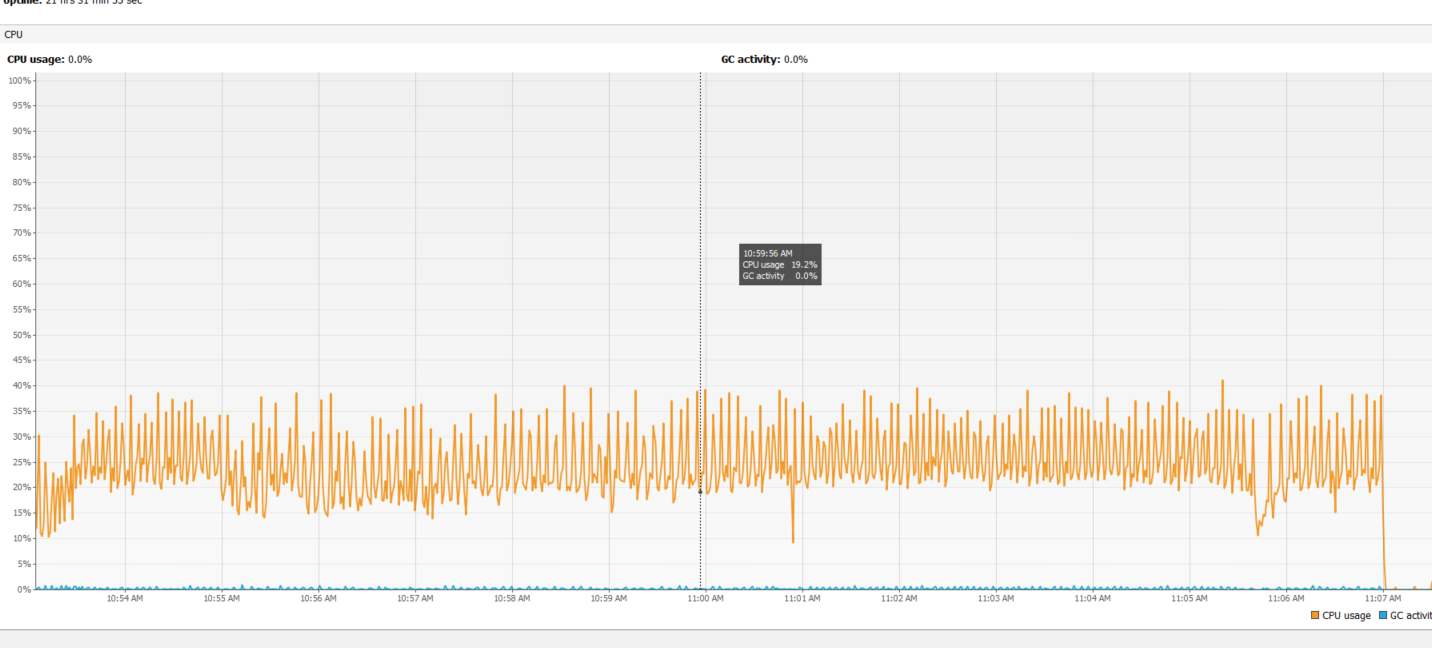
**Result 6**

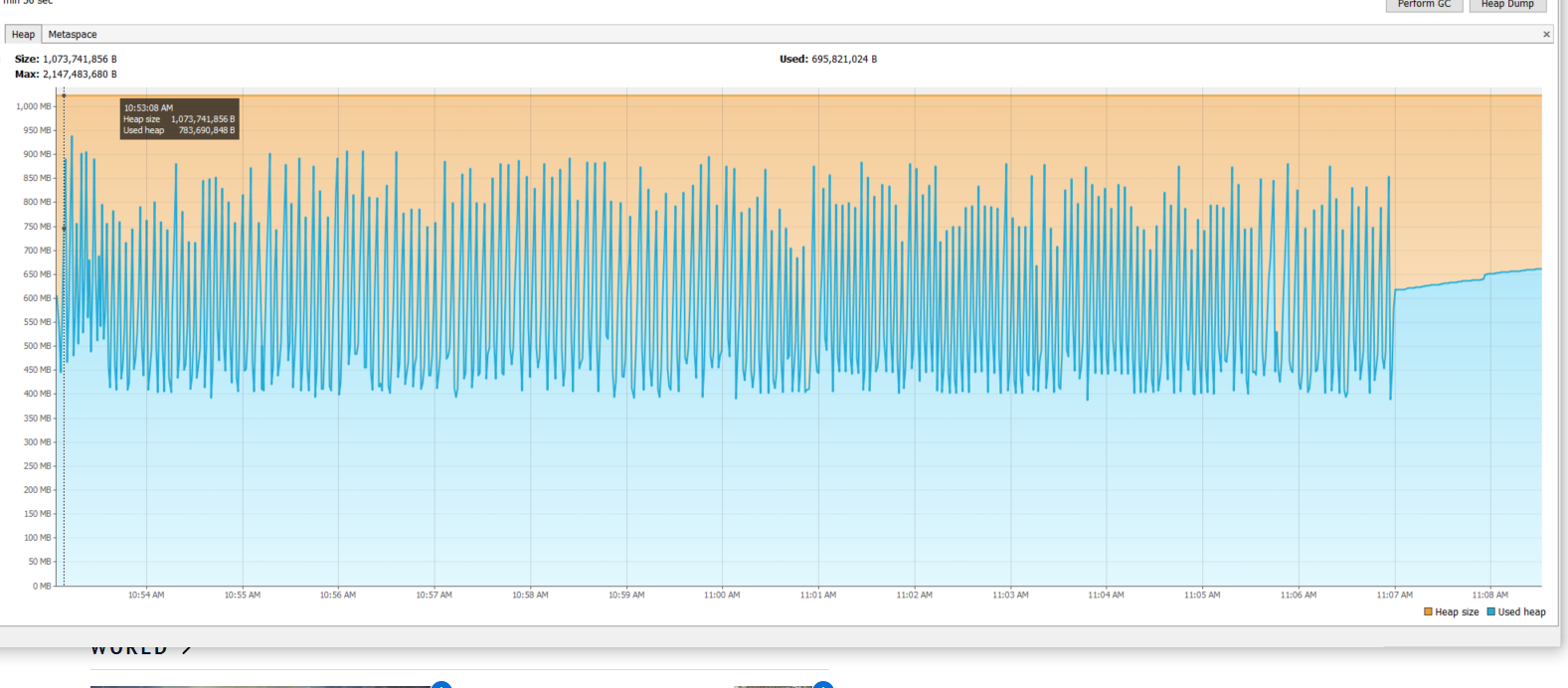




**Result 6A**

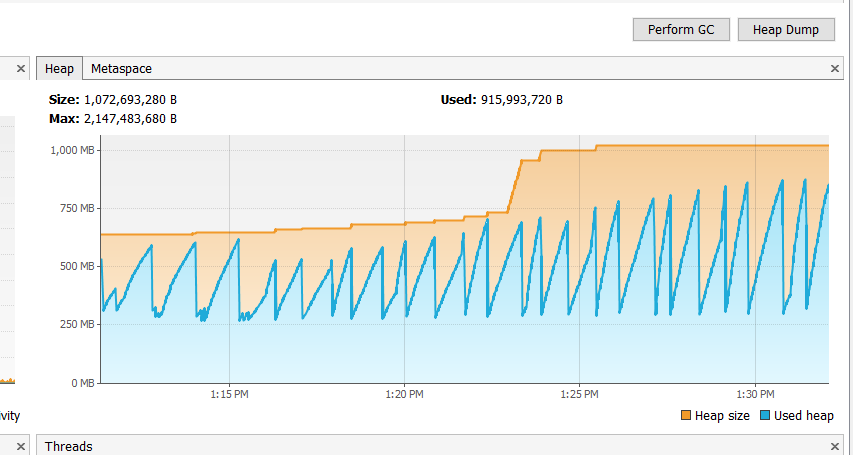
This was run again

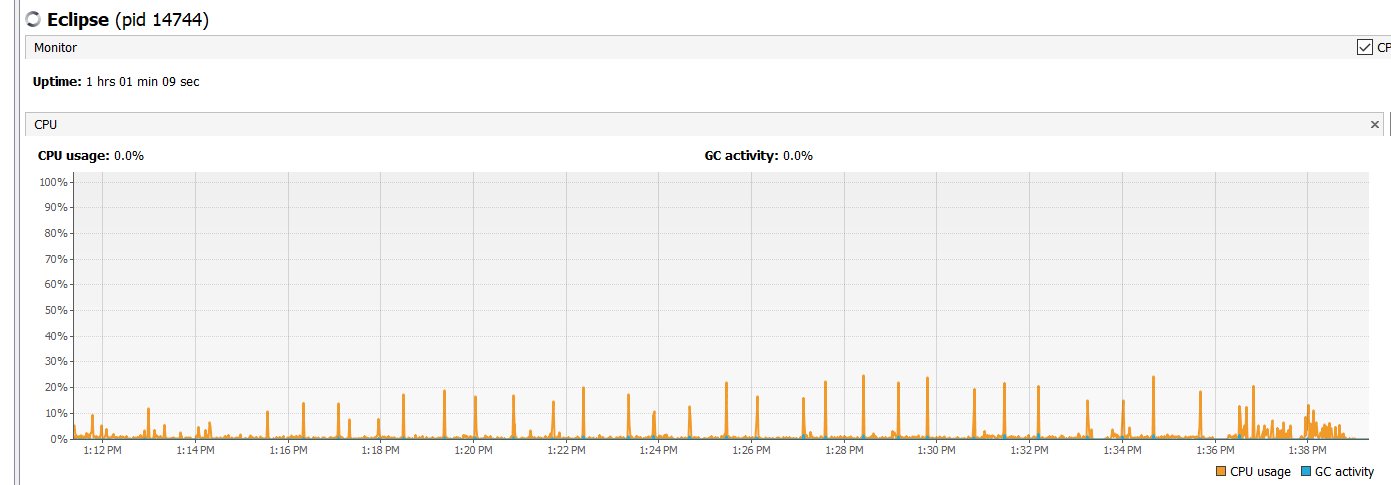




**Result 7**

This seemed to significantly longer as the intern process was looking in the large array of strings- in 20 minutes only 500k loops were executed - CPU was significantly lower..





**Appendix**

Some random note on the creation of the git pages etc and pushing to the host - just for cut and pasting into he command line.

**or create a new repository on the command line**

echo "# JavaStringvsStringBuildervsStringBuffer" >> README.md

git init

git add README.md

git commit -m "first commit"

git remote add origin https://github.com/RobAllan27/JavaStringvsStringBuildervsStringBuffer.git

git push -u origin master

**…or push an existing repository from the command line**

git remote add origin https://github.com/RobAllan27/JavaStringvsStringBuildervsStringBuffer.git

git push -u origin master