EXTENDED ESSAY

Computer Science

What is the Effect of Different Garbage Collectors on the Performance of Java Applications requiring Throughput and Response Time?

Candidate Name: Purvi Misal

Candidate No.: 000902 -0011

Supervisor Name: Ms. Poonam Kanda

School: Gandhi Memorial International School Jakarta, Indonesia

Word Count: 3896

Abstract

The whole idea of my project is to explore the working of various garbage collectors and how they affect the performance of the application developed in Java language through studying the effect of GCs on the throughput and response time of the application.

A large part of performance management is based on how the memory is managed. My project revolves around the idea of an application requiring resources to fulfil a request, but these resources are available in a very limited quantity. These resources consist of the Random Access Memory and also the CPU. The performance and scalability of an application directly depends upon the resources available. When there are fewer resources available, the more heavily what is available will be used and the performance will be degraded. Every system needs to make sure that its resources are used up efficiently, otherwise memory leakages are caused due to the lack of efficiency and the running out of resources. This can quickly lead to poor application scaling and ever longer execution times. In Java, garbage collection takes care of the memory management to advance the performance.

In this essay, I investigated the effect of different garbage collectors on Application's Response time and Throughput. During the process of writing this essay, I analysed the working of different garbage collectors used in the JVM to manage the memory optimally. There are various JVM's created by different software companies for example IBM, Sun Microsystems, Hotspot by oracle Inc. I am focusing on the Hotspot JVM[1] by Oracle. At the

1 Java Virtual Machine also referred to as JVM

end of my project I will be able to choose the garbage collector that is appropriate for applications that need a quicker response time and applications that require a larger throughput. (Word count- 294)

Acknowledgements

I would like to express my special gratitude towards the various people in making this project possible and the ones who encouraged me to take up this topic for my essay. I thank my supervisor, Ms. Poonam Kanda for generously spending her free time to guide me through this essay and helping me assess and evaluate it. I also thank my parents for supporting me throughout the process and providing me with the appropriate resources to complete my essay.

Table of Contents

1. Introduction

The topic of my Extended Essay clicked to me when I was studying Java language and how it is used, in Computer Science subject. I had learnt a bit of C++ in the past and had a little knowledge about the memory-allocation and de-allocation functions of C++. While learning Java in grade 12, I started comparing the two languages; C++ and Java. While learning and researching a bit, I realized that Java language does not have any malloc[2] and dealloc[3] functions that exclusively allocated the memory and de-allocated the memory when memory was about to get exhausted. Java has the inbuilt capabilities of managing the memory, and that happens to be the main difference between C++ and Java. Garbage Collection[4] is responsible for handling the Memory Management automatically and this process is totally transparent to the developers. Though it claims to do this management automatically and keep away the intricacies of memory issues, it doesn't work that way. There are impacts of GCs that cannot be totally ignored when issues like memory leaks and sluggishness in the performance rise. If ignored, the GC can bring the programs to stand still and affect the performance expectations of an application. Therefore, I realized that tuning the GCs is an important part of the performance management and this needs to be done by the developer, as the different GCs with different algorithms can affect the performance (throughput and response time) highly. Thus, I decided to investigate the effect of garbage collectors on the throughput and response time of an application and started off with my research.

2 Memory allocation
3 Deallocation

4 "14) Garbage Collection." *Java Tutorial Hub*. N.p., n.d. Web. 26 June 2013.

2. Performance/Resource Management

The Garbage Collection in JVM is made to do its memory management automatically. It takes CPU cycles to do this job. When the garbage collector[5] is running, the real application stops running because the CPU cycles are now used in doing the memory management. This is called as stop-the-world process, because when this process is happening, other processes stop running. If the Computer is having a Single CPU, the CPU cycles act as a limited resource too.

The time for which real application is stopped for application processing and is working for the GC is called the Pause Time. In some Mission Critical Online Real Time applications, the response time is utmost important and in such applications, developer cannot afford to have pause time for memory management and waste CPU cycles for this process. Thus, conscious decisions have to be taken to use an appropriate garbage collector to improve the response time, therefore, the performance by the developer.

There are other applications where only throughput is important, and in such an application the pause time taken by garbage collector for memory management (stop-the-world process) can be compromised. In this, the response time is not the prime concern because the Performance of the application depends on throughput.

Thus, I realized that the definition of Performance is different for different application. The type of application and what it requires plays an important part here. The developer needs to decide which garbage collector to use for the Application, keeping in mind the performance requirements of application. Different garbage collectors are, thus, implemented for different applications.

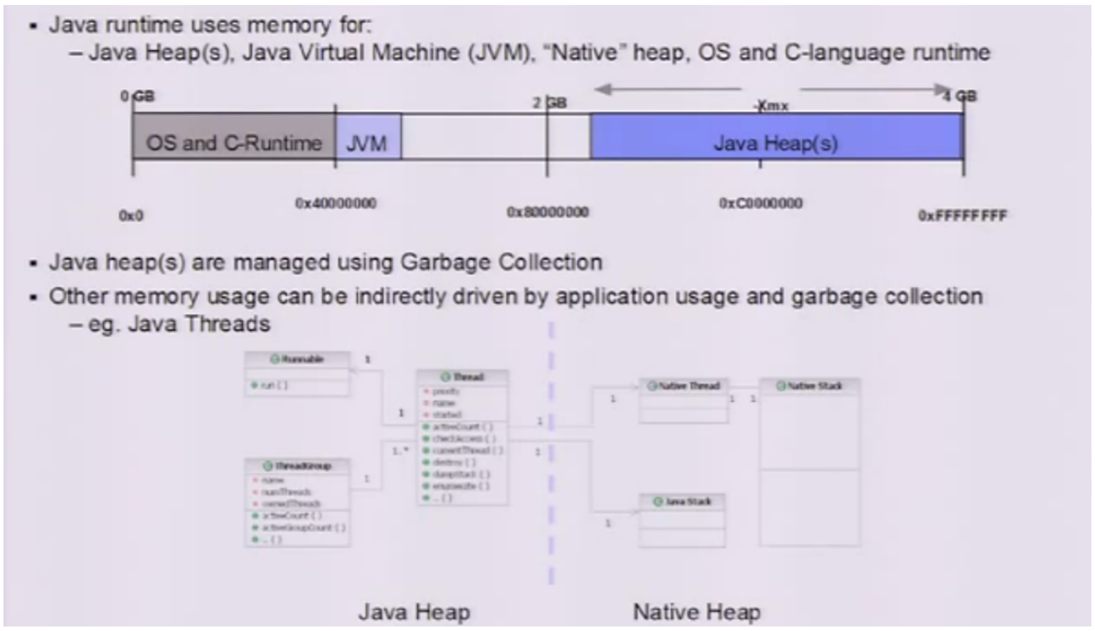
The CPU cycles are acting as a resource and also the Java Heap, which is the part of RAM[6] have to bear memory leakages and soon reach to the exhaustion of memory if not used wisely. While a resource is in use, it’s not available to attend other requests. These limited resources have the potential to become performance bottlenecks, thus, become an important

5 Also referred to as GC
6 Random Access Memory

focus for monitoring and application scaling. When application response becomes slower, it is almost always due to a lack of sufficient resources. So it's important to keep an eye on the utilization of individual resources and monitor the transactions using them.

- What is *Throughput*?
  The speed of Server processing to respond to request in one second of time is called Throughput. Throughput is the measure of the number of messages that a system can process in a given amount of time. In software, throughput is usually measured in "Requests /Seconds[7]" or "Transactions /Seconds".[8]
- What is *Response time*?
  The time to get first response from the server is called Response time. Response time is usually measured in units of "Seconds/Request[9]" or "Seconds / Transaction". [10]

3. Java Memory Management


- Java runtime uses memory for:
  – Java Heap(s), Java Virtual Machine (JVM), "Native" heap, OS and C-language runtime
- Java heap(s) are managed using Garbage Collection
- Other memory usage can be indirectly driven by application usage and garbage collection
  – eg. Java Threads

Different Programming Languages use different Storage Data Structures for storing the Objects. Java uses only the Heap for allocation of all the objects. All different objects are

7 Requests fulfilled in one second
8 "Calculating Throughput and Response Time » Javidjamae.com." *Calculating Throughput and Response Time » Javidjamae.com*. N.p., n.d. Web. 21 May 2013.
9 Seconds taken for one request or transaction
10 "Calculating Throughput and Response Time » Javidjamae.com." *Calculating Throughput and Response Time » Javidjamae.com*. N.p., n.d. Web. 21 May 2013.

allocated in a heap in no particular order leading to the problem of Fragmentation- having sufficient total available memory for the allocation of the large object in terms of size but not in a contiguous chunk as a usable space. Heap Fragmentation issue can be resolved by reorganizing the heap at runtime by relocating the live objects to other area within the heap. This is transparent to application other than the time it takes to do so. Most of the Algorithms for garbage collector stop the application while reorganizing the Heap which extend the response time of the application. When there is a need of Memory requirement, Java tells it to the System by the "new"[11] operator.

Java Memory Management has its own built-in garbage collection feature, which is one of the finest features in Java. It gives the developers the freedom to create new objects without worrying explicitly about memory allocation and de-allocation, because the GC automatically reclaims memory for reuse. This enables faster development and also helps eliminating memory leaks and other memory-related problems.[12]

Even though developers are away from doing the memory management, they are burdened with the extra step of selecting the proper GC that best meets their performance needs. For strict performance goals, the developers may need to clearly choose the garbage collector and adjust certain parameters to achieve the desired level of performance. Insufficient Java heap leads to: OutofMemoryError due to java heap exhaustion. Also, when the Garbage Collection is running excessively, the CPU usage is amplified, thus, affecting performance of the application.

4. Garbage Collection Algorithms

11 The keyword "new" is a Java operator used to initialize or make an instance of a class and allocates a memory in the RAM.
12 "Java Memory Management." *JavaWorld*. N.p., n.d. Web. 21 May 2013.

The following are the three basic algorithms[13] that are used in garbage collections by the GCs:

Reference Counting:

This Algorithm keeps the counter for each chunk of memory allocated and the number of pointers within the chunk. As assignments are made, the counter is updated. If the reference count drops to zero it is immediately returns to the free pool storage. This Algorithm is Resource intensive in multi-threaded Environments as the reference counts must be locked for mutual exclusion reference count updates.

Mark & Sweep:

In this Algorithm, the Marker starts at Root pointer which references to all threads, stacks and static (global) variables. It recursively marks all the objects that are directly or indirectly referenced from the objects reachable from the roots. Once the Marking is completed, the "Sweep" stage starts. The unmarked objects are swept back to the free list for reuse. Memory Compaction also takes place which brings down all the objects into one place. Compaction helps to get big chunk of free memory available for allocation.

Stop & Copy:

This algorithm halts all threads to go into an exclusive GC phase. The heap is divided into Active and New part known as "Semi-space". Active Pointer is not the garbage and copies it to the Active New semi-space and makes it current active. The advantage is it avoids the Heap fragmentation. It is the fastest Garbage Collection algorithm but requires double the Memory. It cannot be used in real time Systems, as it makes computer freeze.

13 "Understanding Java Garbage Collection." *CUBRID*. N.p., n.d. Web. 26 May 2013.

5. The Choice of the garbage collector Algorithms

To maximise the performance of an application, the developer needs to choose the most appropriate and apt GC with a suitable algorithm. The choice of the right garbage collector algorithms are based on the following comparisons:

- ❖ Parallel vs. Serial – Parallel methodology divides even the job of garbage collector into several slices and can be run simultaneously with different CPUs. In Serial, one task happens at one time, either the application is processed or garbage is collected.
- ❖ Stop-the-World vs. Concurrent – With Concurrent, one or more GC tasks are performed concurrently with the application, but can also face a few short stop-the-world pauses. In Stop-the-World algorithm, execution of application is completely put off during the collection.[14]
- ❖ Compacting vs. Non-Compacting vs. – Copying GC algorithm duplicates the live objects to another memory area by taking additional time for managing. Allocations of new objects are done in a faster and easier way at the first free space through Compacting algorithms. Whereas Non-Compacting algorithms does faster completion of garbage collection, with fragmentation; it does not move all live objects to create a large reclaimed region.

An *interactive* application might require low pause time, whereas overall execution is more important to a *non-interactive* application.[15]

14 "Garbage Collection: Serial vs. Parallel vs. Concurrent-Mark-Sweep." *Who Are We?* N.p., n.d. Web. 25 June 2013.
15 "Java SE at a Glance." *Java SE*. N.p., n.d. Web. 19 June 2013.

6. Java Heap

(What role does the heap play during garbage collection?[16])

In the Generational collection of the heap, the memory is divided into Generations – separate portions holding objects from different ages. Weak Generational Hypothesis is being used in the Generational Garbage Collection –

- For the Long Time most of the objects are not being referenced, they usually die in short time.
- From Older to younger objects Few References can be exists.

The frequently happening of the Young Generation is Fast and efficient due to the small space of the Young Generation. This space is mostly small and lot of objects are contained in it which are not referenced.[17]

The Tenured or promoted object which comes to old generation has already survived few numbers of collections from young generation. Young generation is smaller than Old Generation and thus its occupancy grows very slowly. Thus, more time is taken in Old Generation collection which is not so frequent.[18]

Entire heap, when it fills up or its part is reached to a certain threshold percentage of occupancy. Time, space and frequency have to be compromised quite often. If heap is large, more time will be taken to fill it up, taking longer and thus, collections will be less frequent and thus, longer response time and more pause times will occur. With a smaller heap, space will be lesser and it will fill up quickly, thus, more frequent collections will be required.

16 "Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine." *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*. N.p., n.d. Web. 21 May 2013.
17 "What the Frequency of the Garbage Collection in Java?" *- Stack Overflow*. N.p., n.d. Web. 22 May 2013.
18 "Java SE at a Glance." *Java SE*. N.p., n.d. Web. 21 May 2013.

7. Garbage Collectors

The GCs are supposed to behave in the ways give below, as I have learnt through my research on this topic[19]:

- Serial GC: Both Young and Old Collections are done consecutively (using one CPU) by halting the application execution while the collection taking place. The Young Generation Collection with a Serial collector is mostly used in applications in which there is no requirement of low pause times. It is used as the default collector for the machines which are not the Server class machines.
- Parallel GC (Throughput Collector): This is used in machines having a lot of memory and multiple CPUs and applications that do not face low pause time restraints, like Batch Processing, billing, payroll, scientific computing. It is usually the default collector in the Server class machines. On the other machines, it can be requested using the -XX: +UseParallelGC command line option.
- Parallel Compacting GC: It is most suitable for machines with multiple CPUs, which reduces the pause time and thus, giving better performance. (-XX:+UseParallelOldGC)
- Concurrent Mark Sweep GC[20]: Being used for Fast Response Time, collection of the Old Generation happens concurrently along with the running of the application.

9. The Experiment

To investigate the effect of GCs on application's response time and throughput:

I will first run the application as it is (default) without specifying any particular type of GC, letting the System make the programmed choices based on the platform and operating system on which the application is running. Thus, checking the performance of the application and see if it matches the Performance aimed for. If its performance is acceptable

19 "Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning." *Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning*. N.p., n.d. Web. 23 June 2013.
20 Also known as CMS GC or Low Latency Collector

from the response time or the through put perspective, will do nothing. If the application seems to have issues with the performance I can review the default Garbage Collection as appropriate or not. I can measure and analyse the performance using the tools. Based on the analysis, I will consider modifying the options for controlling the heap size or the garbage collection behaviour. Thus, my approach is to first measure and then control. All JVM's deploy the best strategy but they cannot detect whether the application cares more about the response time or the throughput. Default Garbage Collection settings are a performance compromise to make sure that most applications will work, but such settings are not optimized for either type of application. Performance of both response time-bound and throughput-oriented applications suffers from frequent garbage collection. However, the strategies for fixing the problem are different for the two kinds of application. Determining the correct strategy and sizing will allow us to remedy the majority of GC-related performance problems. To optimize for response time, we will use the GC suspensions monitoring, this will let us know how much time is being spent for the suspended application and will work towards the Garbage collection which will avoid these longer suspensions. To optimize for the throughput, it is important to make the GC as fast as possible. Using the CPU's in parallel is one of the ways to reduce the suspensions and getting the work done in parallel. Incorrect or non-optimized garbage-collection configurations are the most common cause for GC-related performance problems. There are various tools to analyse the applications logs to identify the GC-related issues. Some tools are the command line based to generate the logs for the heap and manually analysing them and other are the Graphical User Interface which are easier to understand and increases the readability. I will use the Tool JVisualVM[21] which is used by most of the Performance Expert of IT Industry to monitor the java applications analysis for observing the various performance parameters like CPU activity, GC activity , how the java heap and it various internal data structures like the

21 JVisualVM- "Java VisualVM." *VisualVM*. N.p., n.d. Web. 24 July 2013.

Survival spaces, old generation and the permanent generation are behaving. Advantage of using the JVisualVM is that it is having most of the graphical tabs and easy to understand statistics rather than going into the details of the complicated log files mostly generated by the command line tools. The ultimate goal is to analyse the heap and how the GC is behaving.

I will do the repeatable and representative tests to measure how the results look. First I will measure the baseline performance. Wherever possible will measure multiple times as the variations will occur between the tests. I have written the java program which will be used to demonstrate the cases. I am using the NetBeans[22] Integrated Development Environment which is the popular tool among java developers world to execute the java program, the listing of the java program is GCObjectsDemo.java, given in the next section.

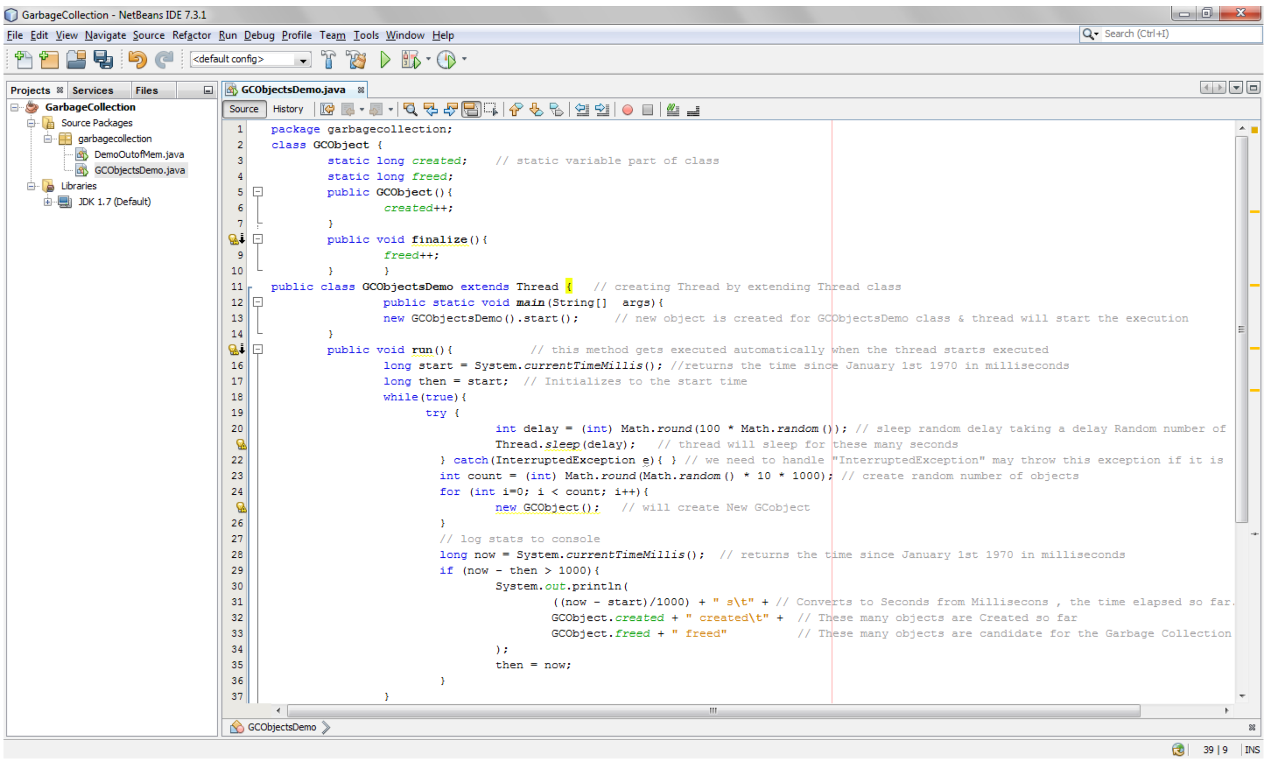
10. The Program (Java Application)

```
//Start of the Program GCObjectsDemo.java

public class GCObjectsDemo extends Thread    // creating Thread by extending Thread class
{
public static void main(String[]  args)
{
new GCObjectsDemo().start();          // new object is created for GCObjectsDemo class & thread will start the execution
}
public void run()              // this method gets executed automatically when the thread starts executed
{
```

22 Netbeans- "NetBeans IDE." *Welcome to NetBeans*. N.p., n.d. Web. 24 July 2013.

```
long start = System.currentTimeMillis();        //returns the time since January 1st 1970 in
milliseconds

long then = start;                               // Initializes to the start time

while(true)

{                              // sleep random delay taking a delay Random number of seconds

try {

int delay = (int) Math.round(100 * Math.random());

Thread.sleep(delay);                              // thread will sleep for these many seconds

} catch(InterruptedException e){ }          // we need to handle "InterruptedException" may
throw this exception if it is  interrupted


// create random number of objects

int count = (int) Math.round (Math.random() * 10 * 1000);

for (int i=0; i < count; i++)

{

new GCObject();                                            // will create New GCobject

}

// log stats to console

long now = System.currentTimeMillis();  // returns the time since January 1st 1970 in
milliseconds

if (now - then > 1000){

System.out.println(

((now - start)/1000) + " s\t" + // Converts to Seconds from Milliseconds , the time elapsed so
far.

GCObject.created + " created\t" +  // These many objects are Created so far in ONE second
GCObject.freed + " freed"          // These many objects are candidates for the next Garbage
Collection (objects that are no longer referenced to)
);
then = now;
}
}
}
}

class GCObject
{
static long created;     // static variable part of class
static long freed;

public GCObject(){
created++;    //Objects getting created in one second
}
public void finalize(){
freed++;          //Objects getting freed in one second
}
}
//End of the Program GCObjectsDemo.java
```

(Screen shot of the program when it's not running. Screenshots of program when it's running with each garbage collector in given in the Appendix 1.)



```java
package garbagecollection;
class GCObject {
        static long created;    // static variable part of class
        static long freed;
        public GCObject(){
                created++;
        }
        public void finalize(){
                freed++;
        }       }
public class GCObjectsDemo extends Thread {   // creating Thread by extending Thread class
                public static void main(String[]  args){
                new GCObjectsDemo().start();     // new object is created for GCObjectsDemo class & thread will start the execution
        }
        public void run(){           // this method gets executed automatically when the thread starts executed
                long start = System.currentTimeMillis(); //returns the time since January 1st 1970 in milliseconds
                long then = start;  // Initializes to the start time
                while(true){
                        try {
                                int delay = (int) Math.round(100 * Math.random()); // sleep random delay taking a delay Random number of
                                Thread.sleep(delay);   // thread will sleep for these many seconds
                        } catch(InterruptedException e){ } // we need to handle "InterruptedException" may throw this exception if it is
                        int count = (int) Math.round(Math.random() * 10 * 1000); // create random number of objects
                        for (int i=0; i < count; i++){
                                new GCObject();   // will create New GCobject
                        }
                        // log stats to console
                        long now = System.currentTimeMillis();  // returns the time since January 1st 1970 in milliseconds
                        if (now - then > 1000){
                                System.out.println(
                                        ((now - start)/1000) + " s\t" + // Converts to Seconds from Millisecons , the time elapsed so far.
                                        GCObject.created + " created\t" +  // These many objects are Created so far
                                        GCObject.freed + " freed"          // These many objects are candidate for the Garbage Collection
                                );
                                then = now;
                        }
                }
```

11. The Procedure

The observations will be noted by executing the GCObjectsDemo.java program. The java program is executed for keeping the entire parameters constant for the execution and changing the garbage collection only in each case. The parameters which are kept constant are, the time for execution of each run is 1800 Seconds, Java Heap Initial and Maximum Size to 256 MB. None of the garbage collection logs are mentioned as any parameter since JVisualVM directly reads the execution of each process automatically. The First Execution is for the Default garbage collection, I kept the program running for 1800 seconds continuously & break the execution after 1800 seconds and noted

- The observations of how many objects got created after 1800 seconds,
- How many times the CPU consumption was reached to 100% which indicates the pause time for the application, significantly
- The number of times the classes were loading and unloading and the average time spend in this activity,
- How many times the Garbage Collections happened during the run of 1800 seconds and,

- The time spent for all those garbage collection, while calculating the throughput and the response time.

The throughput is calculated against the default setting of the garbage collection. I have executed the run of the program for 6 different Garbage collection algorithms and seen how the throughput and response times are varying. The throughput is based on in 1800 seconds how many objects my Java program was able to create. When the objects gets created it slowly gets the java heap filled up, and thus, JVM decides based on the certain threshold when to start collecting the non-reference objects and automatically triggers the garbage collection to happen, the timings for GC varies depend on each GC algorithm. Since the program is written to be executed in a while…true loop which is supposed to be called as an infinite loop, the manually stopping of the program exactly at 1800 seconds by cancelling is bit difficult and shown that it causes the program to run some more seconds. The program was run for 30 minutes for each of the 6 different cases, that is, using six different GCs in the same way, one of which is the Default GC that JVM uses. To change the GC, I just had to specify to Netbeans which GC I wanted to use and the heap size I wanted my application to use during the run time.

The 6 cases/GCs[23]:

1. Default – Automatically taken by JVM
2. -XX:+UseConcMarkSweepGC-X:ParallelCMSThreads=2
3. -XX:+UseSerialGC
4. -XX:+UseParallelGC
5. -XX:+UseParNewGC
6. -XX:+UseParallelOldGC

23 The screenshots of every step of the procedure of the experiment is given in the Appendix1

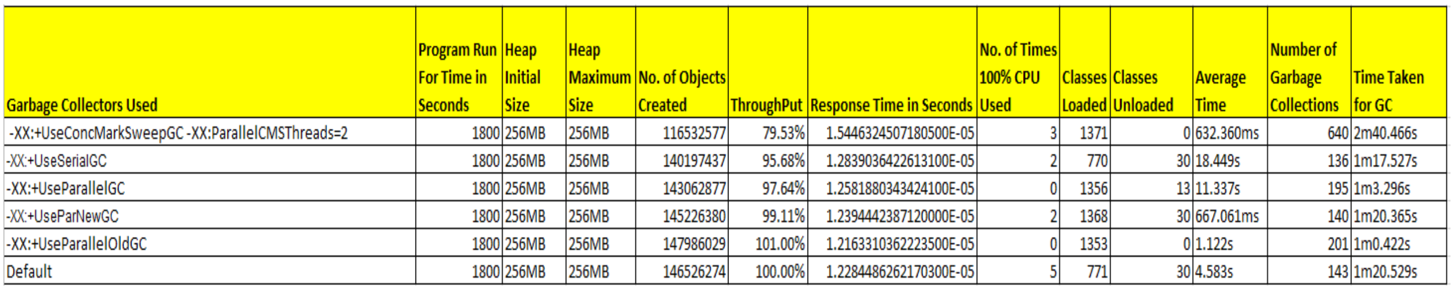
12. Results of the Experiment

The observations/results of my experiment[24]:

Through the execution of my program in Netbeans, I noted down the total number of objects that were created in 30 minutes. I noted the other essential, analytical observations from the JVisualVM, portraying the behaviour of each GC that I tested.

That left me with the following results.

Raw Data



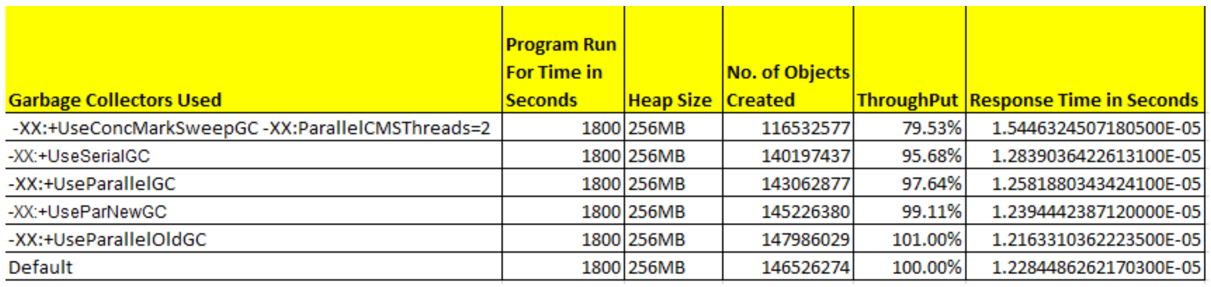
Garbage Collectors Used	Program Run For Time in Seconds	Heap Initial Size	Heap Maximum Size	No. of Objects Created	ThroughPut	Response Time in Seconds	No. of Times 100% CPU Used	Classes Loaded	Classes Unloaded	Average Time	Number of Garbage Collections	Time Taken for GC
-XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2	1800	256MB	256MB	116532577	79.53%	1.5446324507180500E-05	3	1371	0	632.360ms	640	2m40.466s
-XX:+UseSerialGC	1800	256MB	256MB	140197437	95.68%	1.2839036422613100E-05	2	770	30	18.449s	136	1m17.527s
-XX:+UseParallelGC	1800	256MB	256MB	143062877	97.64%	1.2581880343424100E-05	0	1356	13	11.337s	195	1m3.296s
-XX:+UseParNewGC	1800	256MB	256MB	145226380	99.11%	1.2394442387120000E-05	2	1368	30	667.061ms	140	1m20.365s
-XX:+UseParallelOldGC	1800	256MB	256MB	147986029	101.00%	1.2163310362223500E-05	0	1353	0	1.122s	201	1m0.422s
Default	1800	256MB	256MB	146526274	100.00%	1.2284486262170300E-05	5	771	30	4.583s	143	1m20.529s

I noted down all the data that JVisualVM gave me and wrote it in an organized, tabular form so I can be able to compare the results produced by each of the five GCs and the default GC.

The Processed Data

The raw data contained of a lot of unnecessary observations that were not needed to compare the 6 GCs and observe their effects on the throughput and response time. So I processed the data and eliminated some data to make the effects of the six GCs clear and distinct.

24 The evidence of my experiment are given in the Appendix 2.



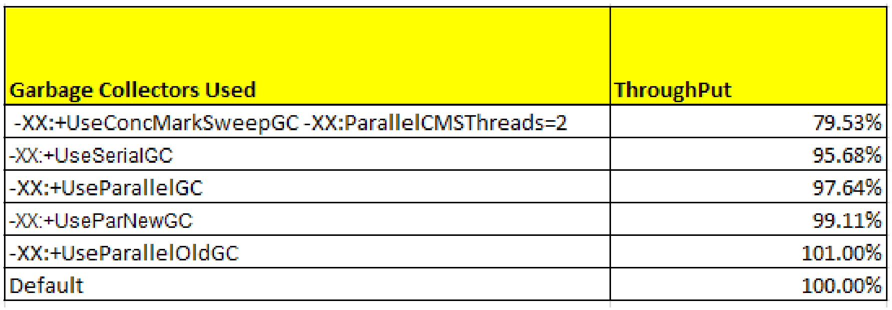
Garbage Collectors Used	Program Run For Time in Seconds	Heap Size	No. of Objects Created	ThroughPut	Response Time in Seconds
-XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2	1800	256MB	116532577	79.53%	1.5446324507180500E-05
-XX:+UseSerialGC	1800	256MB	140197437	95.68%	1.2839036422613100E-05
-XX:+UseParallelGC	1800	256MB	143062877	97.64%	1.2581880343424100E-05
-XX:+UseParNewGC	1800	256MB	145226380	99.11%	1.2394442387120000E-05
-XX:+UseParallelOldGC	1800	256MB	147986029	101.00%	1.2163310362223500E-05
Default	1800	256MB	146526274	100.00%	1.2284486262170300E-05

I made this table in Microsoft Excel 2010. I calculated the time taken for 1 object to be formed by the program with each GC by dividing 1800(seconds) by the number of objects created with every GC. I calculated the throughput[25] with comparison to the number of objects created in the default GC. These were very basic calculations that I did in MSExcel.

Graph

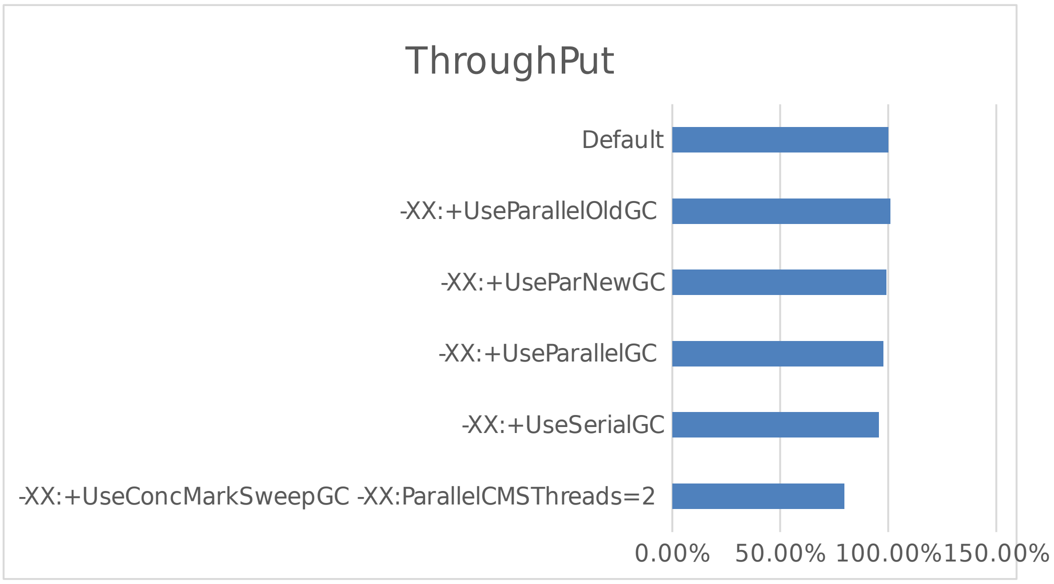
To make the difference in the effects of the six GCs, I plotted two graphs:

Throughput of the GCs



Garbage Collectors Used	ThroughPut
-XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2	79.53%
-XX:+UseSerialGC	95.68%
-XX:+UseParallelGC	97.64%
-XX:+UseParNewGC	99.11%
-XX:+UseParallelOldGC	101.00%
Default	100.00%

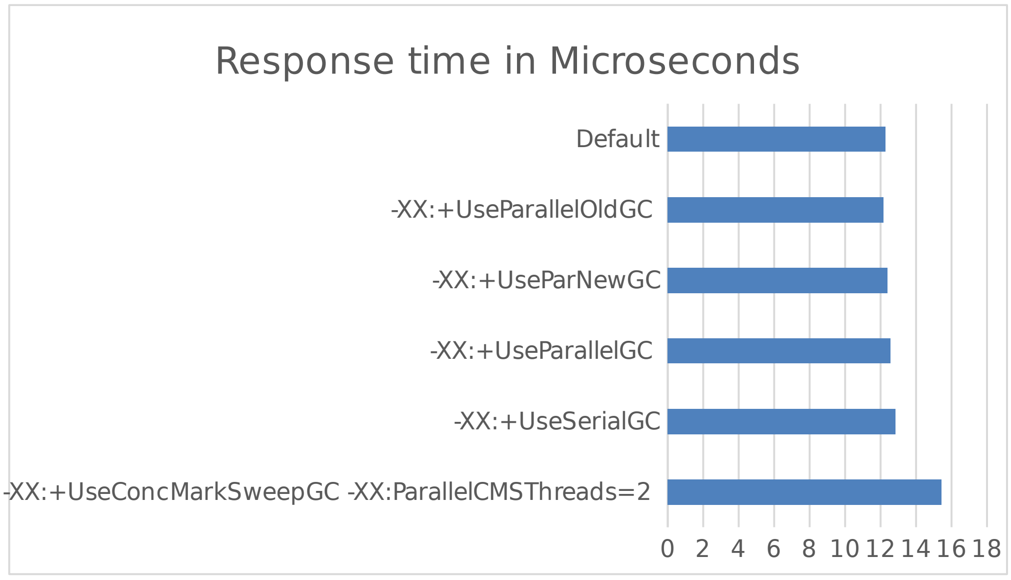
25 Throughput is the percentage of objects created in 1 second of the garbage collector compared to the objects created by the default garbage collector



This graph shows that the throughput of different GCs varies by a visible amount. The Old Parallel GC gives the most throughput results, and the Concurrent Mark & Sweep GC gives the least. Thus, Parallel new GC is considered to be a better GC for applications requiring higher throughput, according to my experiment.

Response time of the GCs

Garbage Collectors Used	**Response time in Microseconds**
-XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2	15.44632451
-XX:+UseSerialGC	12.83903642
-XX:+UseParallelGC	12.58188034
-XX:+UseParNewGC	12.39444239
-XX:+UseParallelOldGC	12.16331036
Default	12.28448626



The graph shows the effect of different GCs on the response time of the application. The Parallel Old GC gives the response time, whereas Concurrent Mark & Sweep gives the most, evidently. Thus, showing that Parallel Old GC will make the application run faster than any other GC, according to my experiment.

13. Conclusion

According to results of my experiment, Parallel Old GC is the GCthat will give the most throughput and the least response time. This is highlighted through the two graphs shown above. I realized that in these results were quite in tuned with the research I had done and proved my apparent hypothesis right. The Old Parallel GC does the garbage collection of the old generation in the java heap in parallel, so it speeds up the process, thus, giving relatively less response time compared to the other GCs. My results also presented the Concurrent-Mark-Sweep GC as the one with longest response time and least throughput, which goes against my research and my apparent hypothesis as, from sources, this GC was supposed to have a shorted response time than most of the other GCs, this is because most of the garbage collection is done concurrently along with the application running. This GC is said to be used for applications that expect short response time, not considering much about the throughput required in running the application. This defect in the result of my experiment can be the effect of the human errors and a few drawbacks of my experiment.

Reflection

1. The program was supposed to run for exact 1800 seconds, but I was unable to stop the program instantly at correct time. So that must have disturbed the results.
2. The program I used for this experiment did only one job, it created and freed objects in a limited memory heap size. This was just one kind of application. For other, more complex programs the results might be different. So that might have affected the reliability of my results.
3. I used only one application to calculate both throughput and the response time. If I used two different applications; one requiring larger throughput and one requiring a shorter response time, then the results would be more accurate and consistent.

The *aim* of my experiment was to demonstrate the effect of different GCs on the running any application, which is very clear and evident through the results of my experiment, and not to find out the best GC that should be used for any kind of application.

While writing and researching for my extended essay, I learnt a lot about how the performance is affected and also a lot of features of Java language, which helped me understand the language in depth and connect to programming, which is a part of my Computer Science syllabus. I wrote the program (for my experiment) using my Computer Science knowledge of Java. Through the essay, I learnt several aspects of resource management and how it affects the performance of an application.

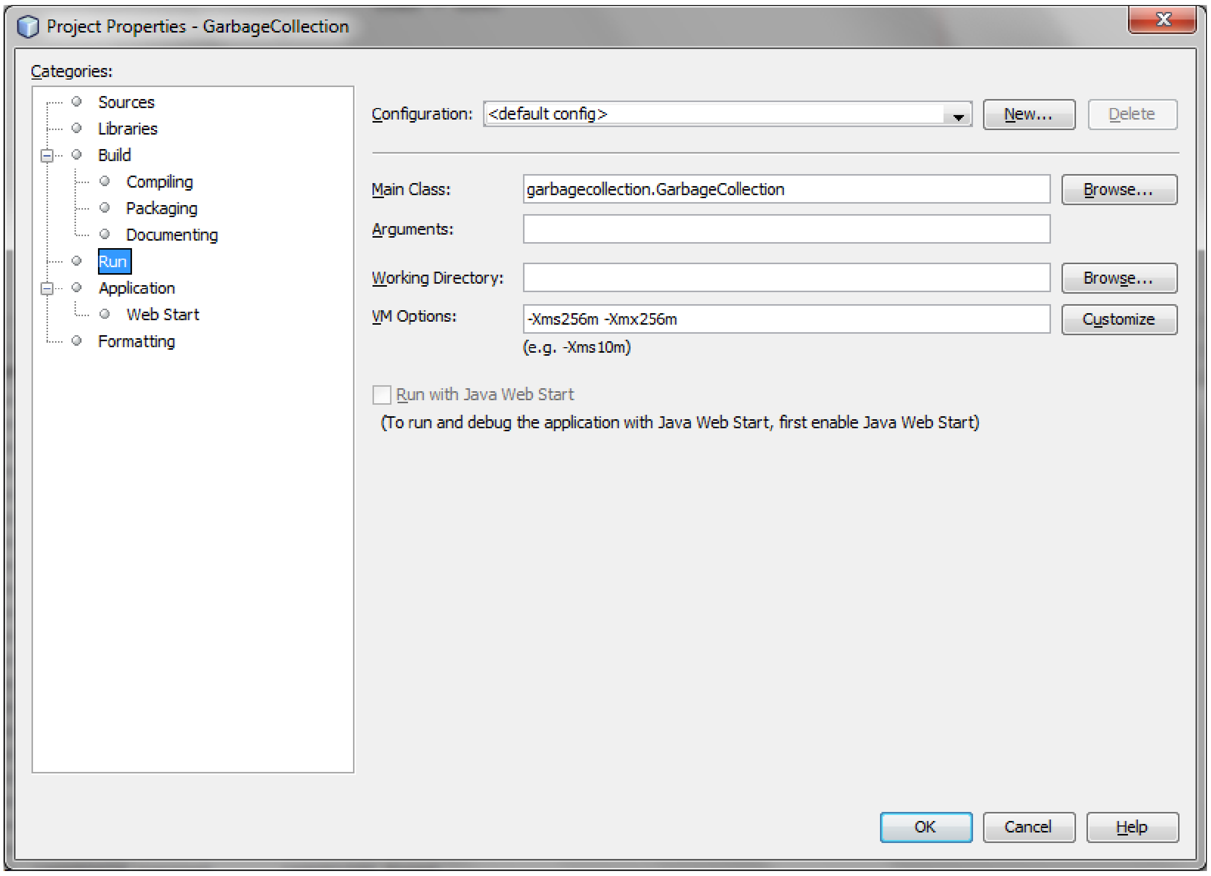
14. Appendices

Appendix 1

Screenshots/Evidence of the procedure of the Experiment:

CASE - 1

DEFAULT SETTINGS FOR GARBAGE COLLECTION



Configuration: <default config>

Main Class: garbagecollection.GarbageCollection

Arguments:

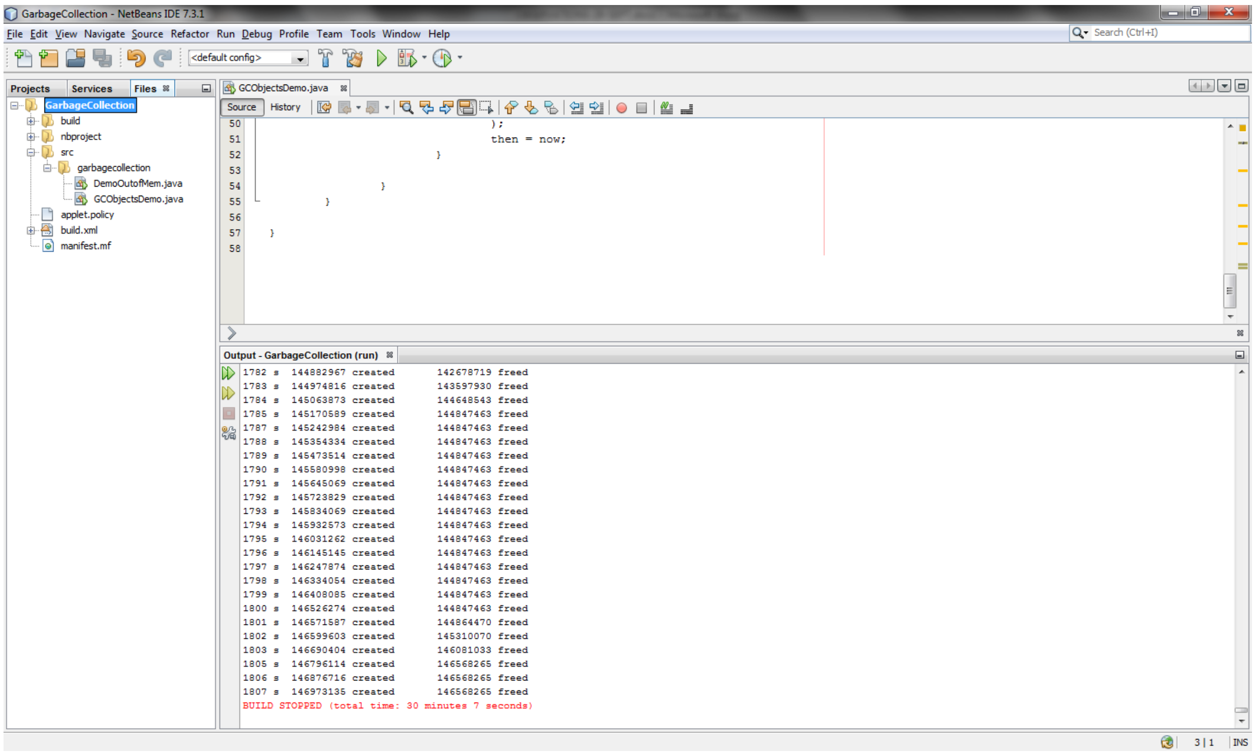
Working Directory:

VM Options: -Xms256m -Xmx256m

(e.g. -Xms10m)

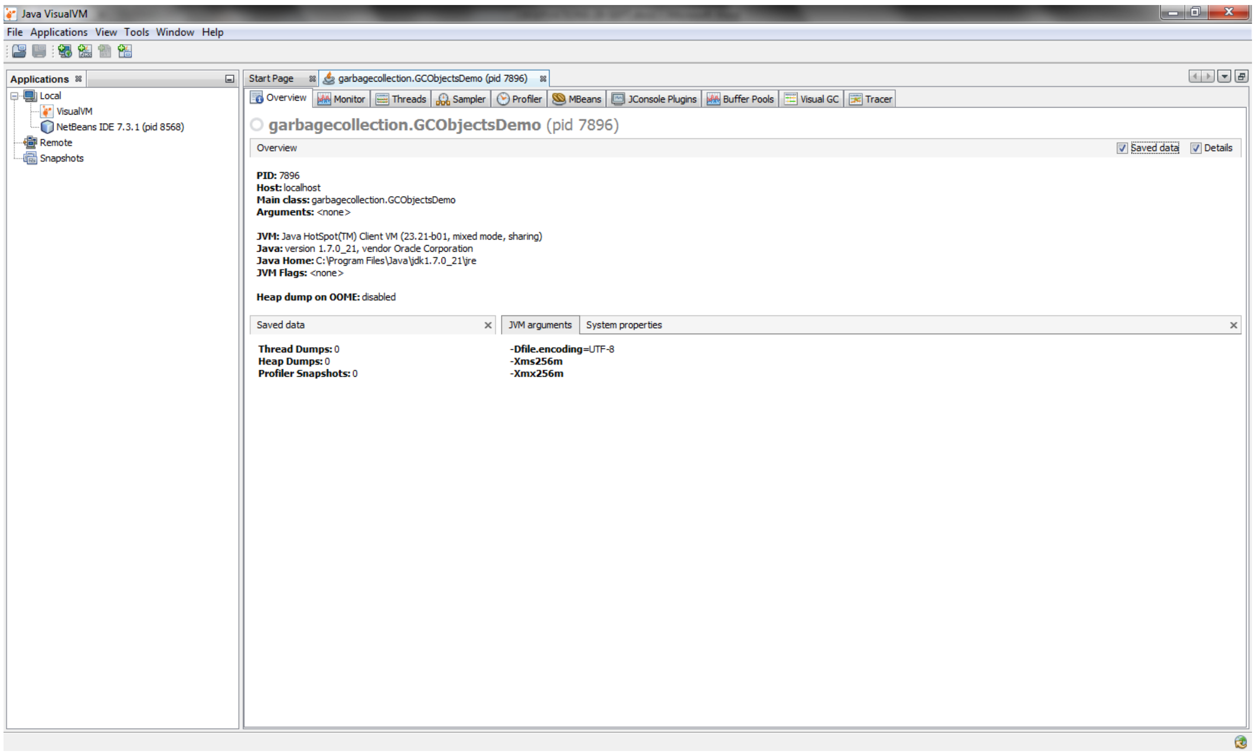
☐ Run with Java Web Start

(To run and debug the application with Java Web Start, first enable Java Web Start)



```
                                        );
                                        then = now;
                                }

                        }
                }

        }

```

```
1782 s  144882967 created       142678719 freed
1783 s  144974816 created       143597930 freed
1784 s  145063873 created       144648543 freed
1785 s  145170589 created       144847463 freed
1787 s  145242984 created       144847463 freed
1788 s  145354334 created       144847463 freed
1789 s  145473514 created       144847463 freed
1790 s  145580998 created       144847463 freed
1791 s  145645069 created       144847463 freed
1792 s  145723829 created       144847463 freed
1793 s  145834069 created       144847463 freed
1794 s  145932573 created       144847463 freed
1795 s  146031262 created       144847463 freed
1796 s  146145145 created       144847463 freed
1797 s  146247874 created       144847463 freed
1798 s  146334054 created       144847463 freed
1799 s  146408085 created       144847463 freed
1800 s  146526274 created       144847463 freed
1801 s  146571587 created       144864470 freed
1802 s  146599603 created       145310070 freed
1803 s  146690404 created       146081033 freed
1805 s  146796114 created       146568265 freed
1806 s  146876716 created       146568265 freed
1807 s  146973135 created       146568265 freed
BUILD STOPPED (total time: 30 minutes 7 seconds)
```



garbagecollection.GCObjectsDemo (pid 7896)

Overview

PID: 7896
Host: localhost
Main class: garbagecollection.GCObjectsDemo
Arguments: <none>

JVM: Java HotSpot(TM) Client VM (23.21-b01, mixed mode, sharing)
Java: version 1.7.0_21, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jdk1.7.0_21\jre
JVM Flags: <none>

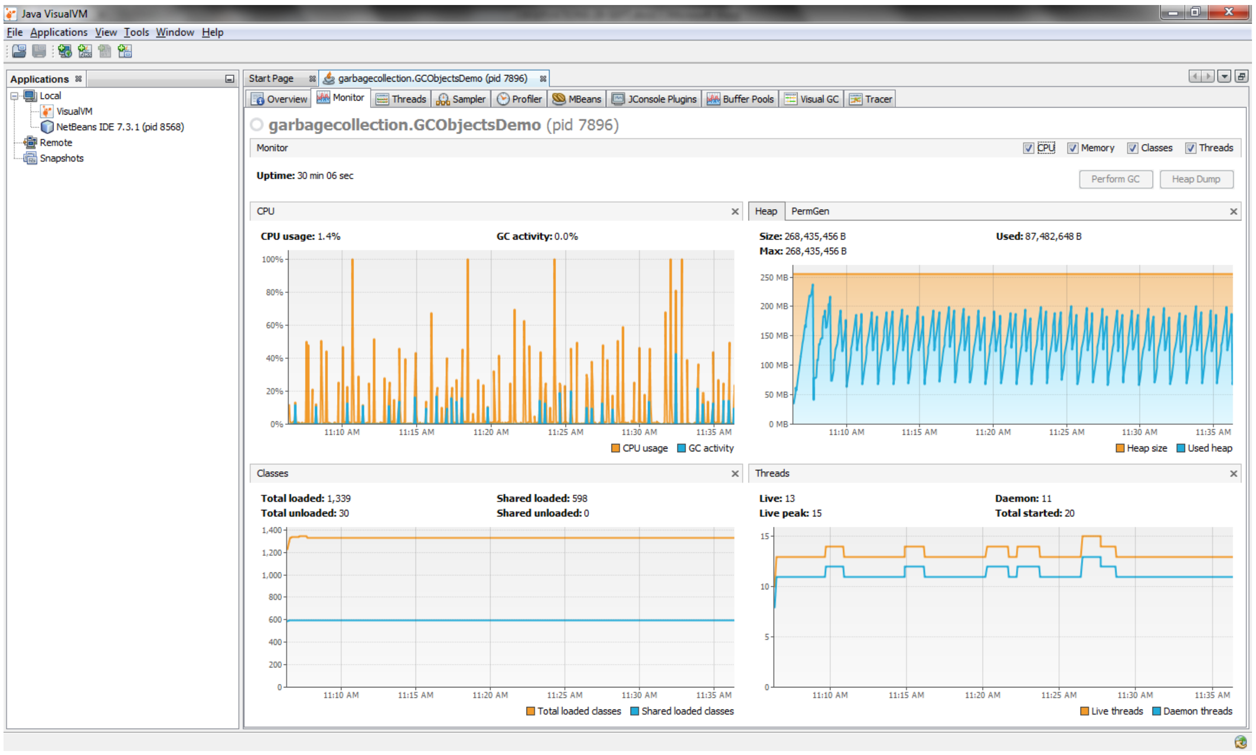
Heap dump on OOME: disabled

Saved data

Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0

JVM arguments

-Dfile.encoding=UTF-8
-Xms256m
-Xmx256m



garbagecollection.GCObjectsDemo (pid 7896)

Monitor

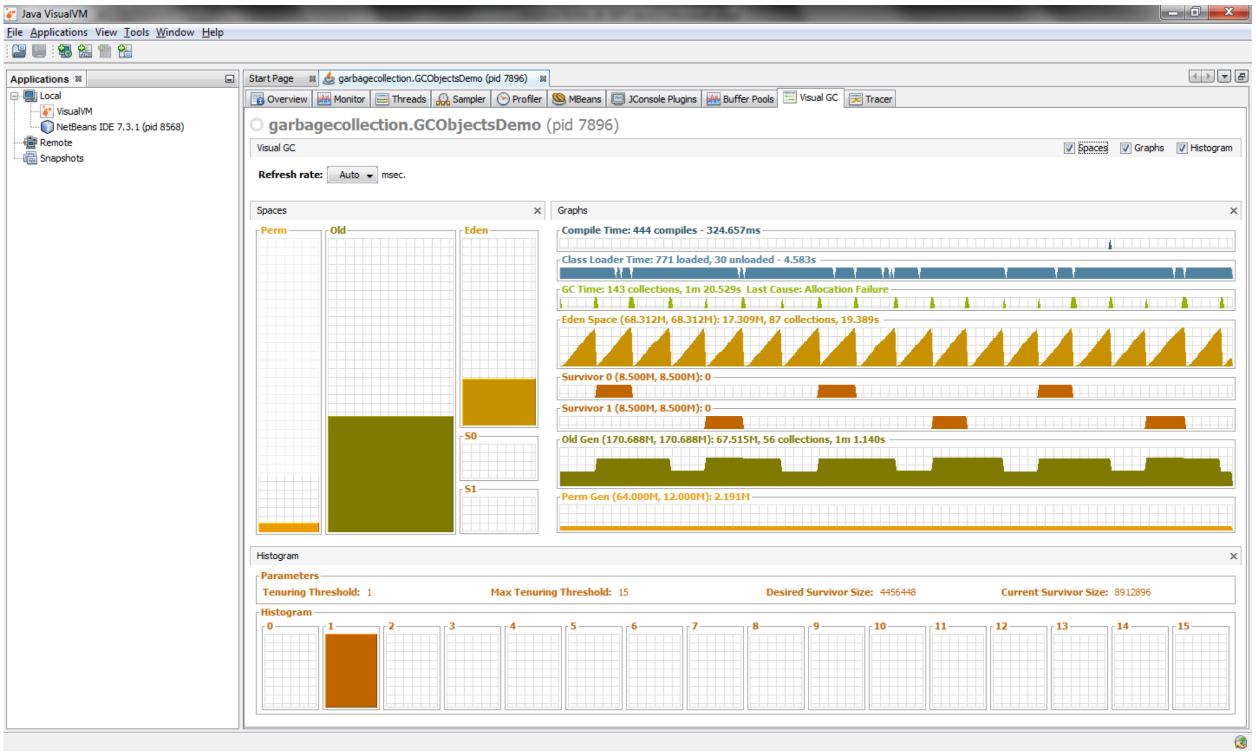
Uptime: 30 min 06 sec

CPU usage: 1.4% **GC activity:** 0.0%

Size: 268,435,456 B **Used:** 87,482,648 B
Max: 268,435,456 B

Total loaded: 1,339 **Shared loaded:** 598
Total unloaded: 30 **Shared unloaded:** 0

Live: 13 **Daemon:** 11
Live peak: 15 **Total started:** 20



garbagecollection.GCObjectsDemo (pid 7896)

Visual GC

Refresh rate: Auto msec.

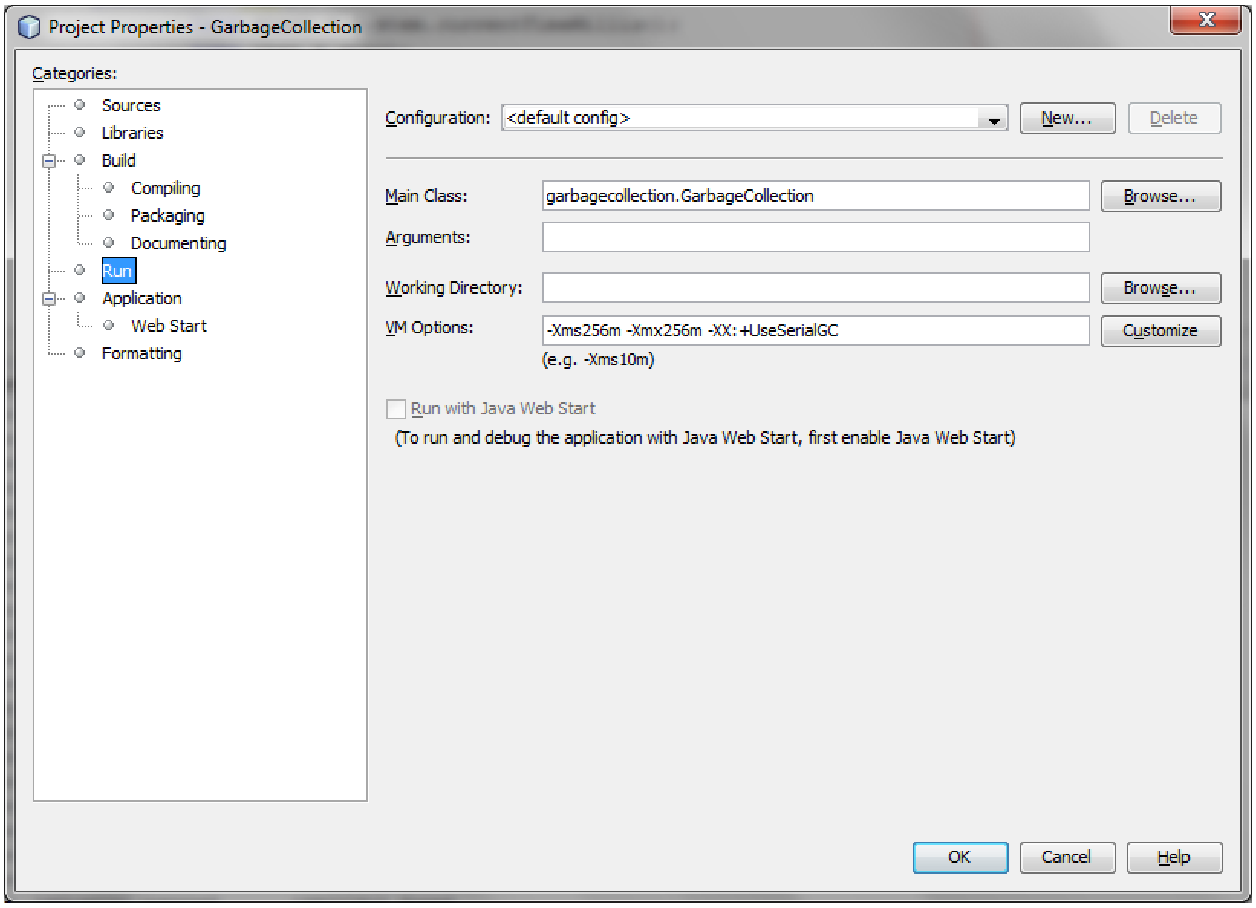
Compile Time: 444 compiles - 324.657ms
Class Loader Time: 771 loaded, 30 unloaded - 4.583s
GC Time: 143 collections, 1m 20.529s Last Cause: Allocation Failure
Eden Space (68.312M, 68.312M): 17.309M, 87 collections, 19.389s
Survivor 0 (8.500M, 8.500M): 0
Survivor 1 (8.500M, 8.500M): 0
Old Gen (170.688M, 170.688M): 67.515M, 56 collections, 1m 1.140s
Perm Gen (64.000M, 12.000M): 2.191M

Histogram

Parameters
Tenuring Threshold: 1 Max Tenuring Threshold: 15 Desired Survivor Size: 4456448 Current Survivor Size: 8912896

CASE-2

GARBAGE COLLECTION USED IS -XX:+UseSerialGC


Project Properties - GarbageCollection

Categories:
- Sources
- Libraries
- Build
  - Compiling
  - Packaging
  - Documenting
- Run
- Application
  - Web Start
- Formatting

Configuration: <default config>

Main Class: garbagecollection.GarbageCollection

Arguments:

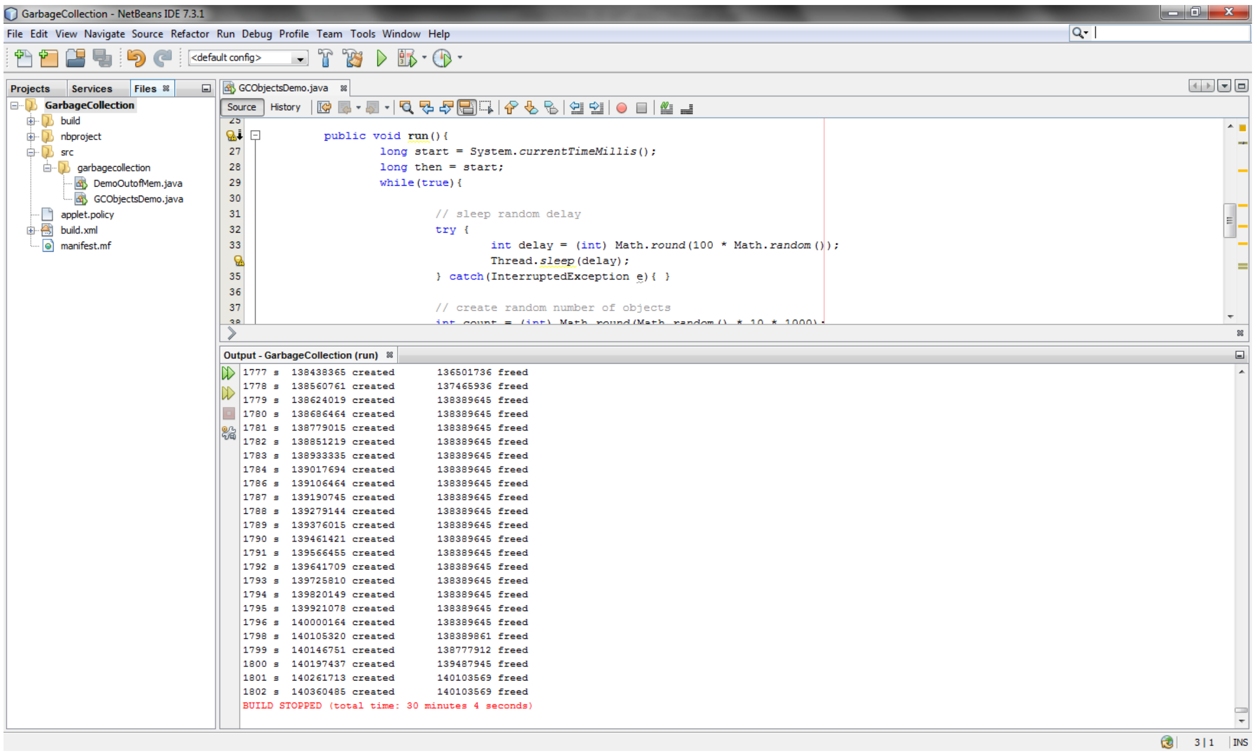
Working Directory:

VM Options: -Xms256m -Xmx256m -XX:+UseSerialGC

(e.g. -Xms10m)

☐ Run with Java Web Start

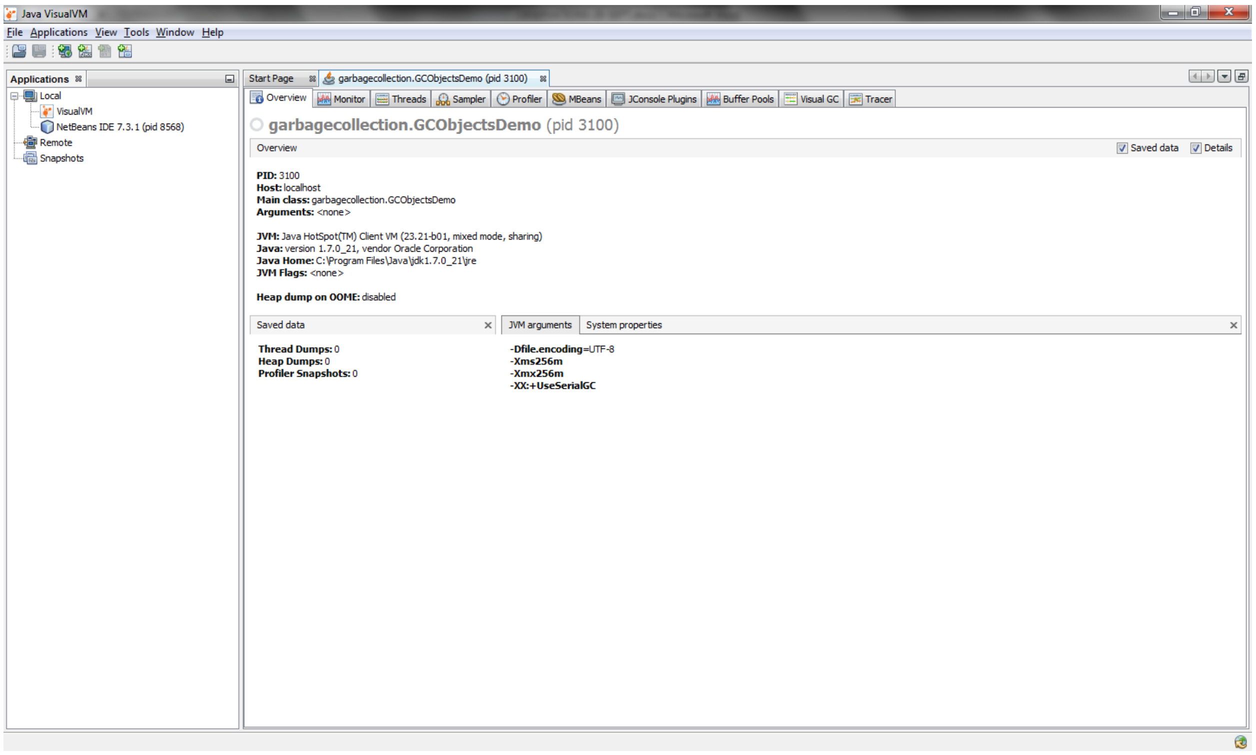
(To run and debug the application with Java Web Start, first enable Java Web Start)



```
public void run(){
        long start = System.currentTimeMillis();
        long then = start;
        while(true){

                // sleep random delay
                try {
                        int delay = (int) Math.round(100 * Math.random());
                        Thread.sleep(delay);
                } catch(InterruptedException e){ }

                // create random number of objects
```

Output - GarbageCollection (run)

```
1777 s  138438365 created       136501736 freed
1778 s  138560761 created       137465936 freed
1779 s  138624019 created       138389645 freed
1780 s  138686464 created       138389645 freed
1781 s  138779015 created       138389645 freed
1782 s  138851219 created       138389645 freed
1783 s  138933335 created       138389645 freed
1784 s  139017694 created       138389645 freed
1786 s  139106464 created       138389645 freed
1787 s  139190745 created       138389645 freed
1788 s  139279144 created       138389645 freed
1789 s  139376015 created       138389645 freed
1790 s  139461421 created       138389645 freed
1791 s  139566455 created       138389645 freed
1792 s  139641709 created       138389645 freed
1793 s  139725810 created       138389645 freed
1794 s  139820149 created       138389645 freed
1795 s  139921078 created       138389645 freed
1796 s  140000164 created       138389645 freed
1798 s  140105320 created       138389861 freed
1799 s  140146751 created       138777912 freed
1800 s  140197437 created       139487945 freed
1801 s  140261713 created       140103569 freed
1802 s  140360485 created       140103569 freed
BUILD STOPPED (total time: 30 minutes 4 seconds)
```



garbagecollection.GCObjectsDemo (pid 3100)

Overview

PID: 3100
Host: localhost
Main class: garbagecollection.GCObjectsDemo
Arguments: <none>

JVM: Java HotSpot(TM) Client VM (23.21-b01, mixed mode, sharing)
Java: version 1.7.0_21, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jdk1.7.0_21\jre
JVM Flags: <none>

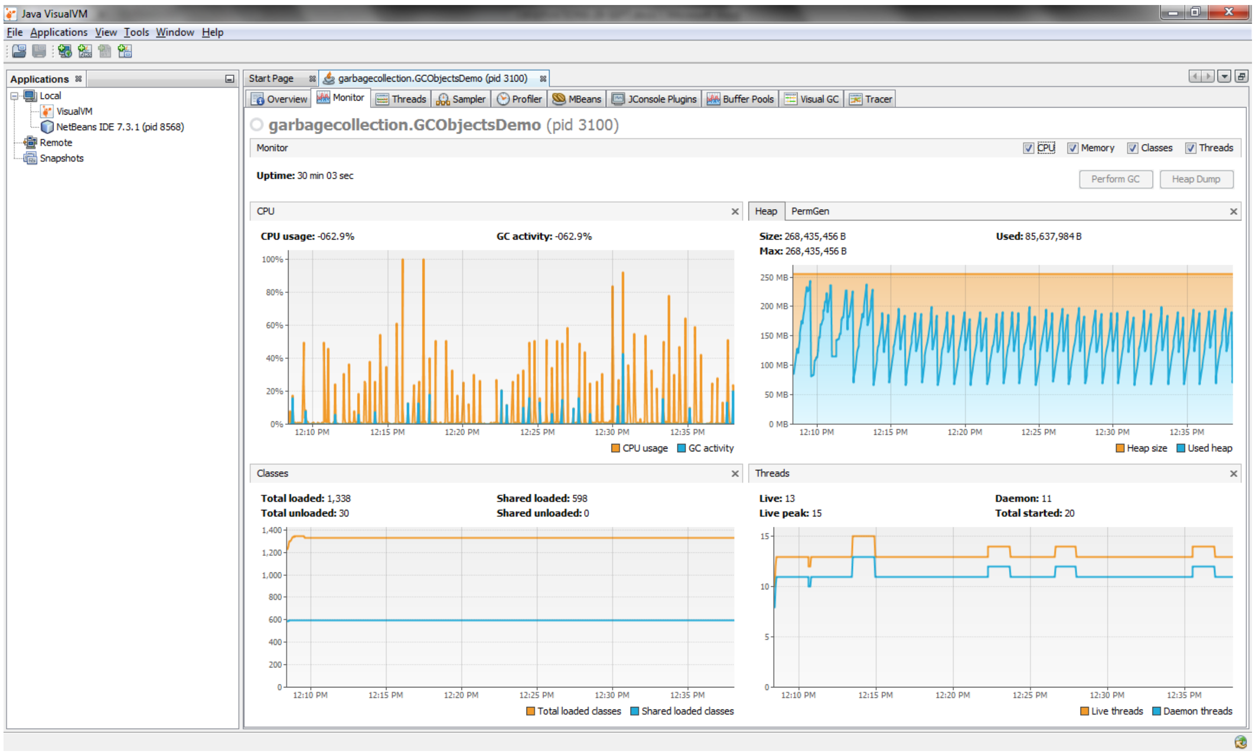
Heap dump on OOME: disabled

Saved data

Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0

JVM arguments

```
-Dfile.encoding=UTF-8
-Xms256m
-Xmx256m
-XX:+UseSerialGC
```


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 8568)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 3100)
Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer
Monitor
CPU Memory Classes Threads
Perform GC Heap Dump
CPU
Heap PermGen
CPU usage GC activity
Heap size Used heap
Classes
Threads
Total loaded classes Shared loaded classes
Live threads Daemon threads

garbagecollection.GCObjectsDemo (pid 3100)

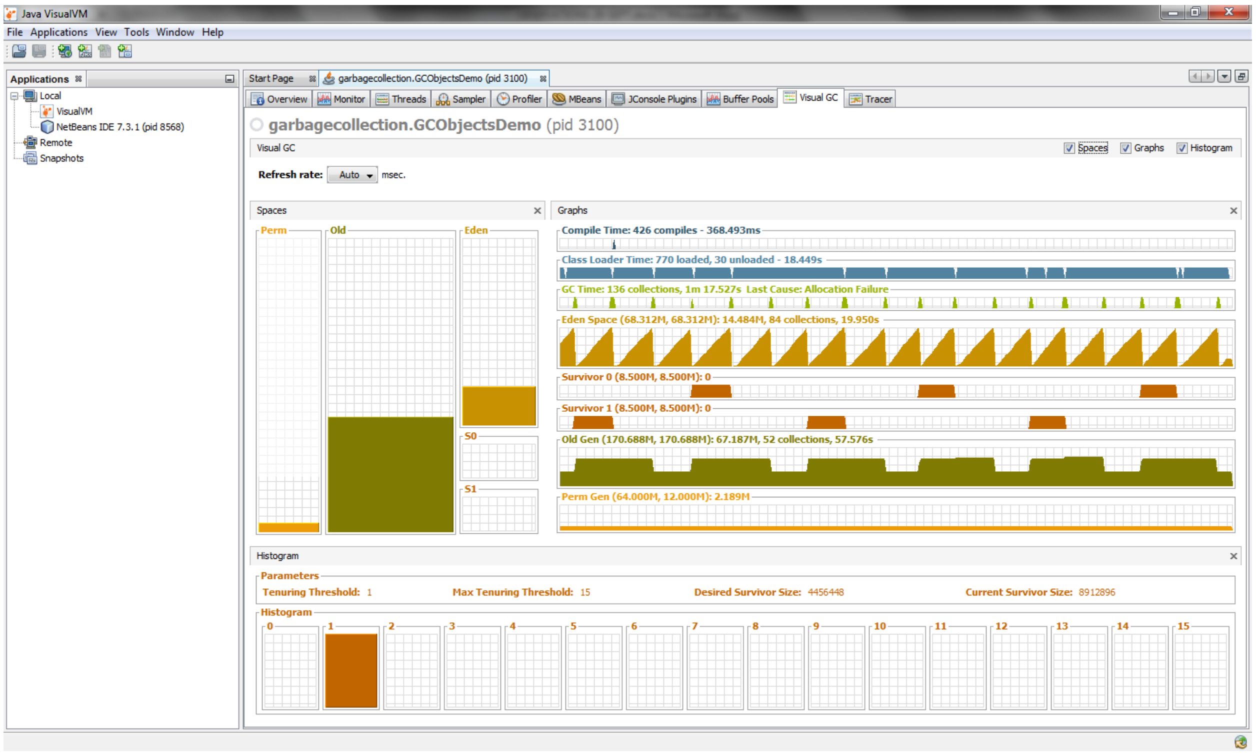
Uptime: 30 min 03 sec

CPU usage: -062.9% GC activity: -062.9%

Size: 268,435,456 B Used: 85,637,984 B
Max: 268,435,456 B

Total loaded: 1,338 Shared loaded: 598
Total unloaded: 30 Shared unloaded: 0

Live: 13 Daemon: 11
Live peak: 15 Total started: 20


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 8568)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 3100)
Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer
Visual GC
Spaces Graphs Histogram
Spaces
Perm
Old
Eden
S0
S1
Graphs
Histogram

garbagecollection.GCObjectsDemo (pid 3100)

Refresh rate: Auto msec.

Compile Time: 426 compiles - 368.493ms
Class Loader Time: 770 loaded, 30 unloaded - 18.449s
GC Time: 136 collections, 1m 17.527s Last Cause: Allocation Failure
Eden Space (68.312M, 68.312M): 14.484M, 84 collections, 19.950s
Survivor 0 (8.500M, 8.500M): 0
Survivor 1 (8.500M, 8.500M): 0
Old Gen (170.688M, 170.688M): 67.187M, 52 collections, 57.576s
Perm Gen (64.000M, 12.000M): 2.189M

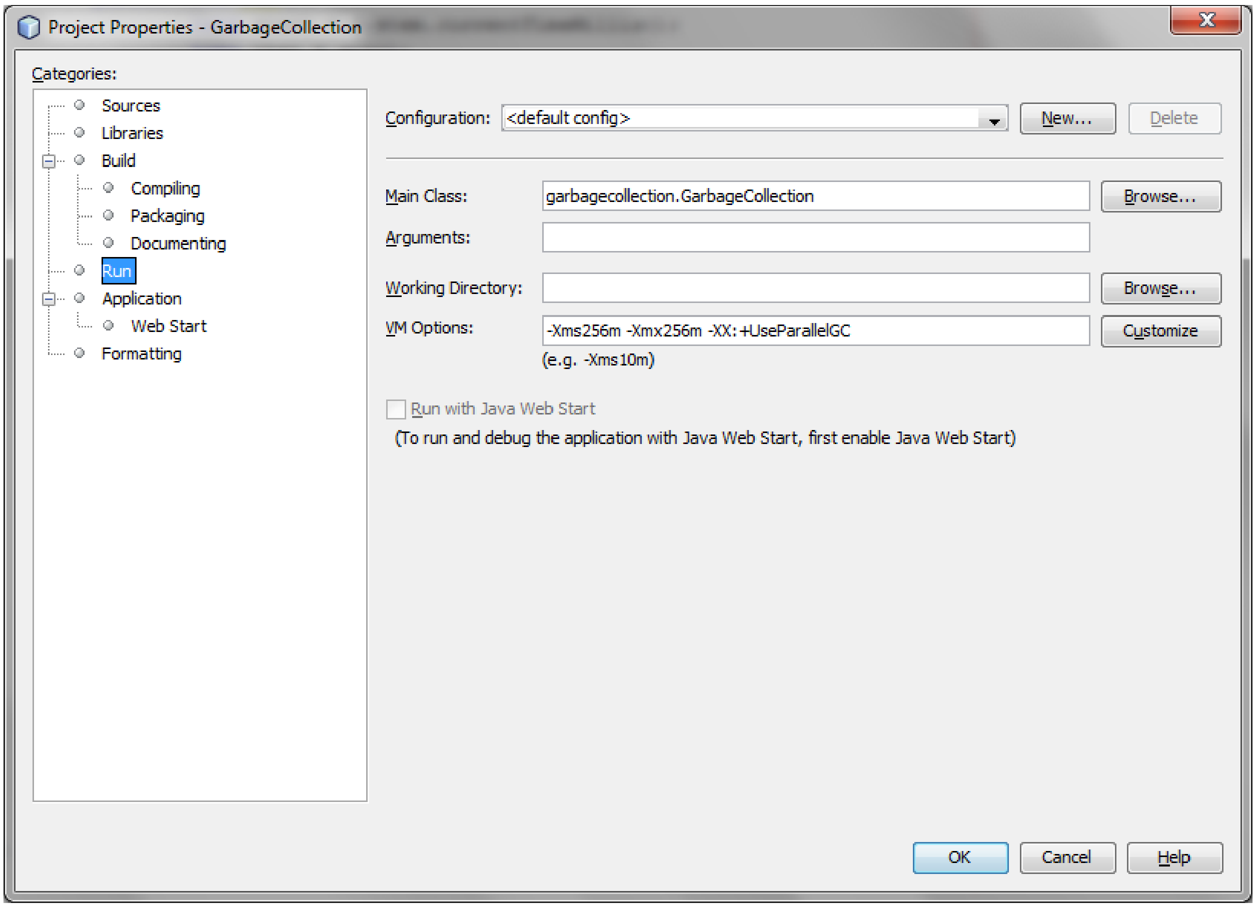
Parameters
Tenuring Threshold: 1 Max Tenuring Threshold: 15 Desired Survivor Size: 4456448 Current Survivor Size: 8912896

Histogram
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

What is the Effect of Different Garbage Collectors on the Performance of Java Applications requiring Throughput and Response Time?

CASE-3

GARBAGE COLLECTION IS -XX:+UseParallelGC



Project Properties - GarbageCollection

Categories: Sources, Libraries, Build (Compiling, Packaging, Documenting), Run, Application (Web Start), Formatting

Configuration: <default config>

Main Class: garbagecollection.GarbageCollection

Arguments:

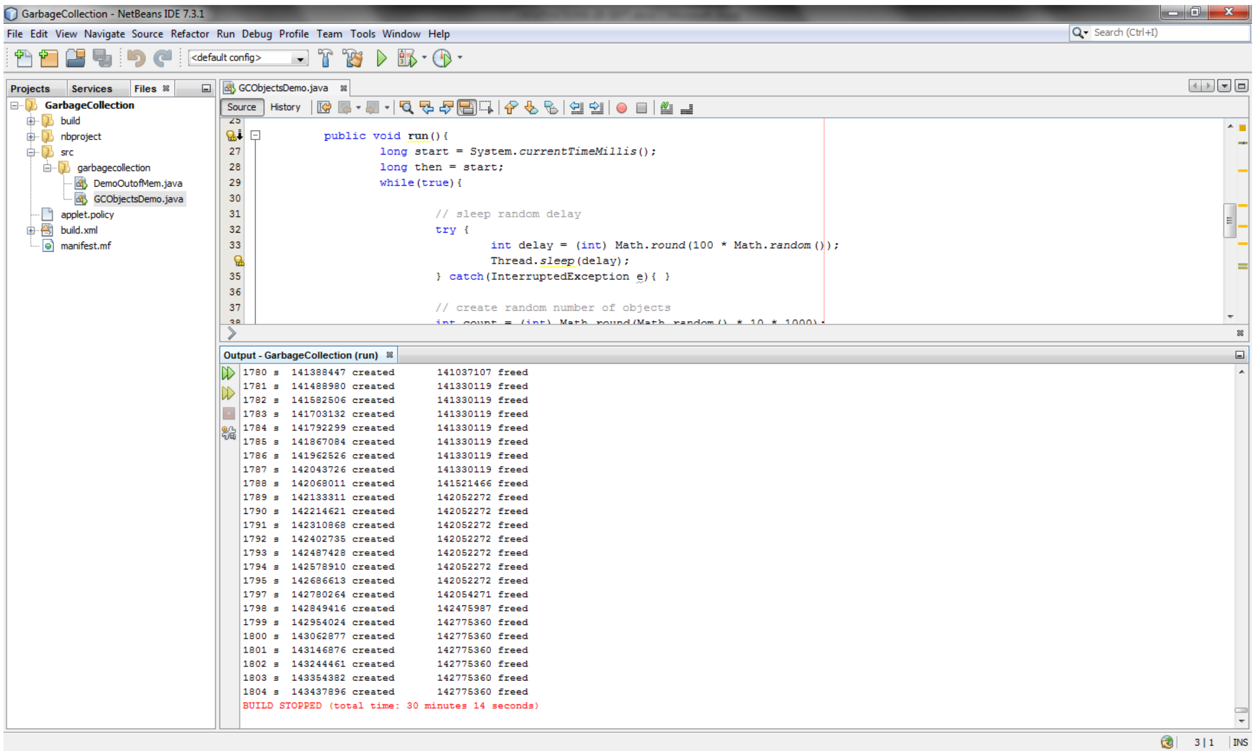
Working Directory:

VM Options: -Xms256m -Xmx256m -XX:+UseParallelGC

(e.g. -Xms10m)

Run with Java Web Start

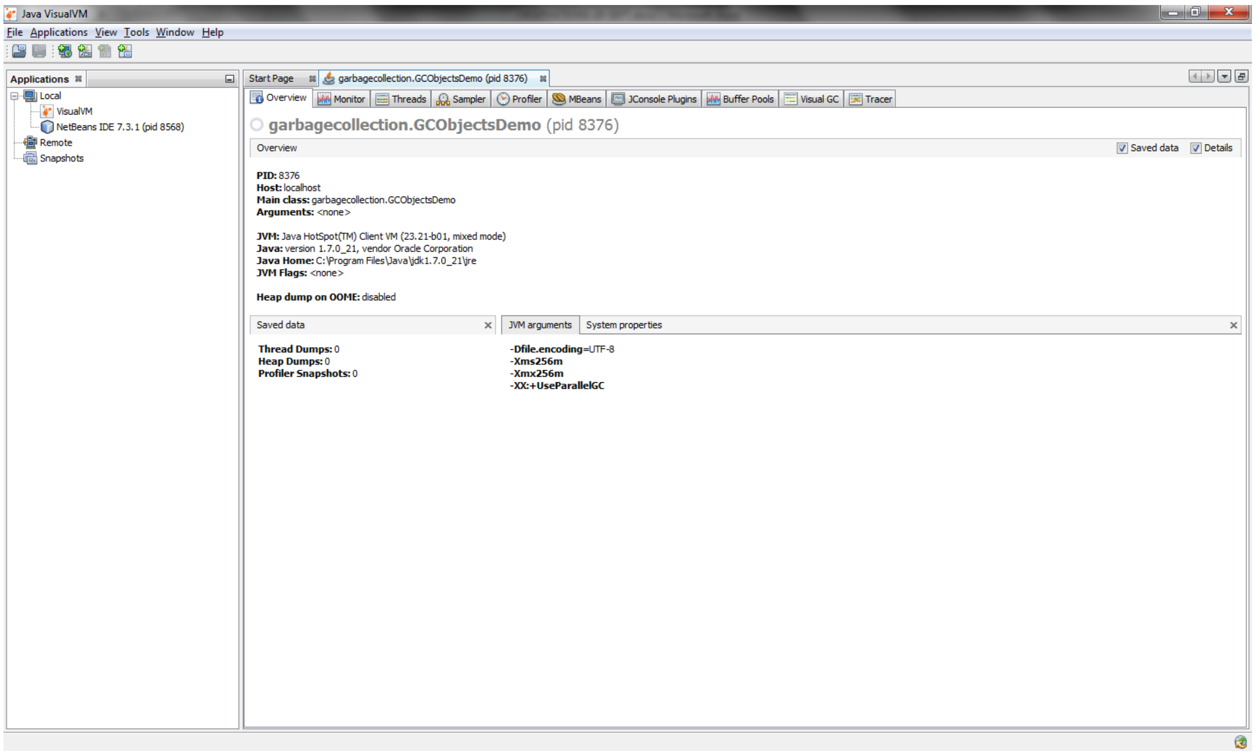
(To run and debug the application with Java Web Start, first enable Java Web Start)



```java
        public void run(){
                long start = System.currentTimeMillis();
                long then = start;
                while(true){

                        // sleep random delay
                        try {
                                int delay = (int) Math.round(100 * Math.random());
                                Thread.sleep(delay);
                        } catch(InterruptedException e){ }

                        // create random number of objects
```

Output - GarbageCollection (run)

```
1780 s  141388447 created     141037107 freed
1781 s  141488980 created     141330119 freed
1782 s  141582506 created     141330119 freed
1783 s  141703132 created     141330119 freed
1784 s  141792299 created     141330119 freed
1785 s  141867084 created     141330119 freed
1786 s  141962526 created     141330119 freed
1787 s  142043726 created     141330119 freed
1788 s  142068011 created     141521466 freed
1789 s  142133311 created     142052272 freed
1790 s  142214621 created     142052272 freed
1791 s  142310868 created     142052272 freed
1792 s  142402735 created     142052272 freed
1793 s  142487428 created     142052272 freed
1794 s  142578910 created     142052272 freed
1795 s  142686613 created     142052272 freed
1797 s  142780264 created     142054271 freed
1798 s  142849416 created     142475987 freed
1799 s  142954024 created     142775360 freed
1800 s  143062877 created     142775360 freed
1801 s  143146876 created     142775360 freed
1802 s  143244461 created     142775360 freed
1803 s  143354382 created     142775360 freed
1804 s  143437896 created     142775360 freed
BUILD STOPPED (total time: 30 minutes 14 seconds)
```


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 8568)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 8376)
Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer
Saved data Details

garbagecollection.GCObjectsDemo (pid 8376)

Overview

PID: 8376
Host: localhost
Main class: garbagecollection.GCObjectsDemo
Arguments: <none>

JVM: Java HotSpot(TM) Client VM (23.21-b01, mixed mode)
Java: version 1.7.0_21, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jdk1.7.0_21\jre
JVM Flags: <none>

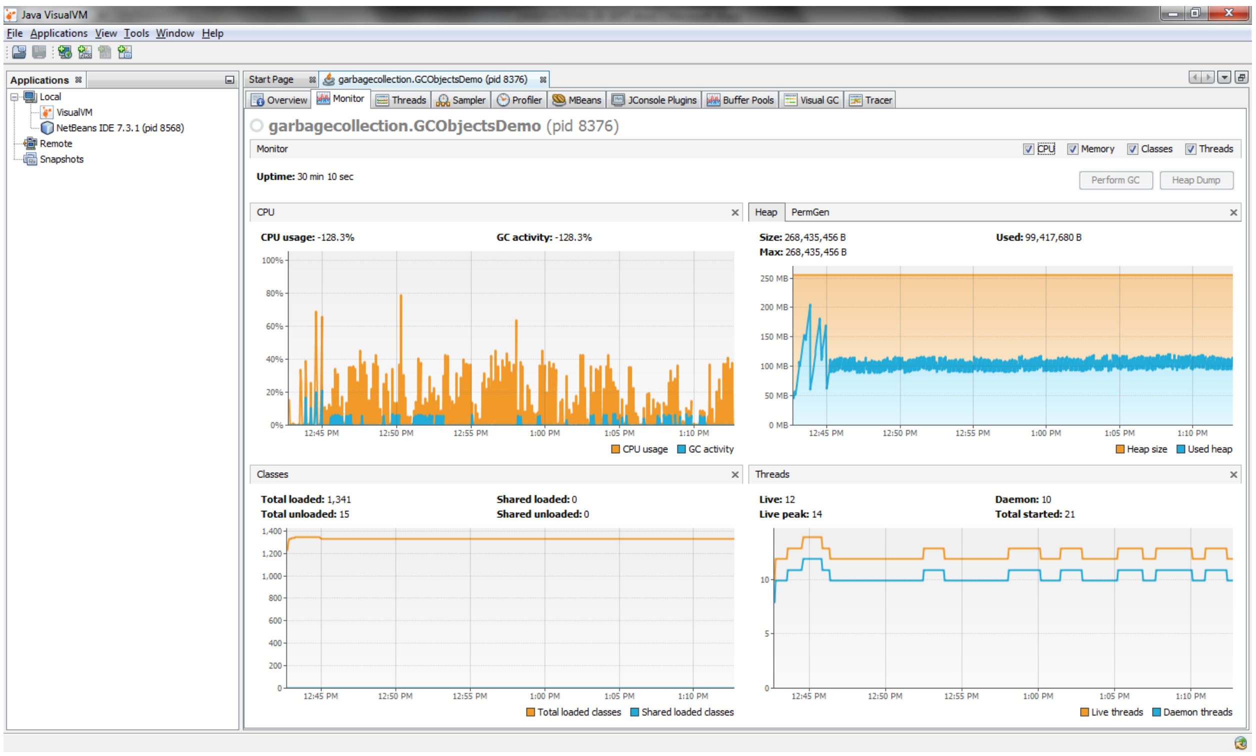
Heap dump on OOME: disabled

Saved data

Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0

JVM arguments | System properties

-Dfile.encoding=UTF-8
-Xms256m
-Xmx256m
-XX:+UseParallelGC


Java VisualVM

garbagecollection.GCObjectsDemo (pid 8376)

Monitor

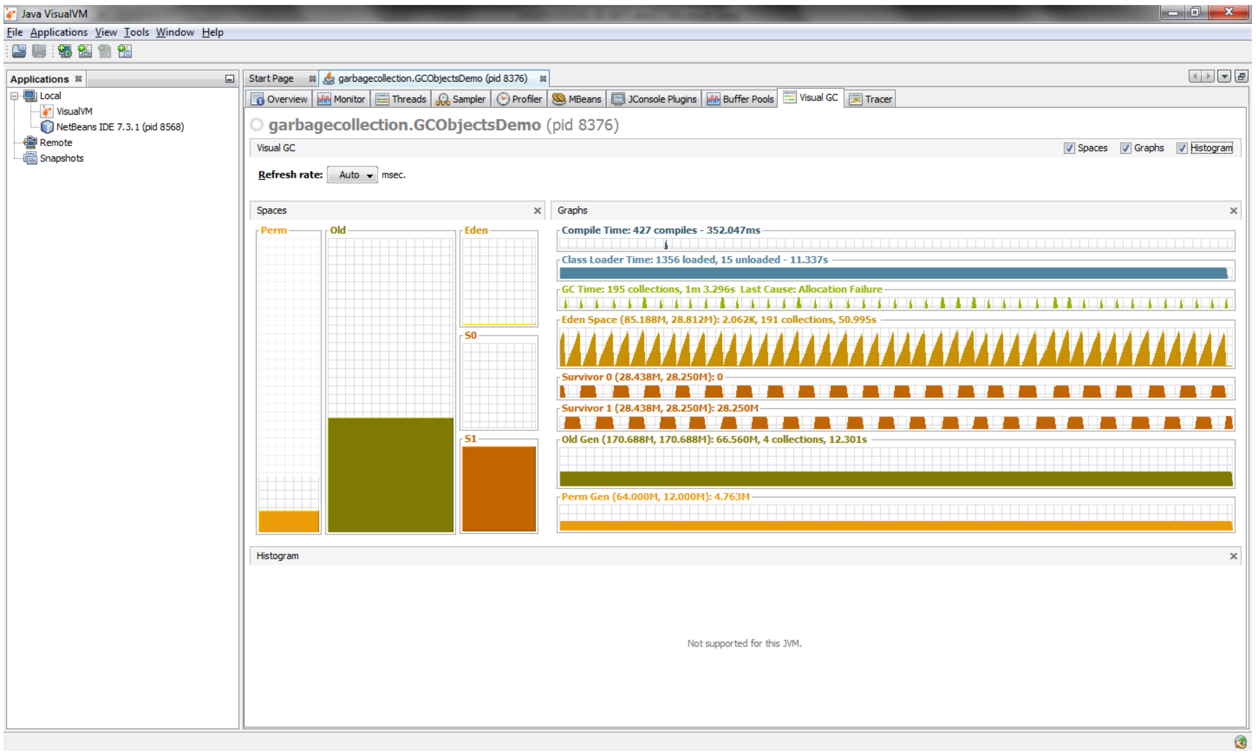
Uptime: 30 min 10 sec

CPU usage: -128.3% GC activity: -128.3%

Heap: Size: 268,435,456 B Used: 99,417,680 B Max: 268,435,456 B

Classes: Total loaded: 1,341 Shared loaded: 0 Total unloaded: 15 Shared unloaded: 0

Threads: Live: 12 Daemon: 10 Live peak: 14 Total started: 21


Java VisualVM

garbagecollection.GCObjectsDemo (pid 8376)

Visual GC

Refresh rate: Auto msec.

Compile Time: 427 compiles - 352.047ms

Class Loader Time: 1356 loaded, 15 unloaded - 11.337s

GC Time: 195 collections, 1m 3.296s Last Cause: Allocation Failure

Eden Space (85.188M, 28.812M): 2.062K, 191 collections, 50.995s

Survivor 0 (28.438M, 28.250M): 0

Survivor 1 (28.438M, 28.250M): 28.250M

Old Gen (170.688M, 170.688M): 66.560M, 4 collections, 12.301s

Perm Gen (64.000M, 12.000M): 4.763M

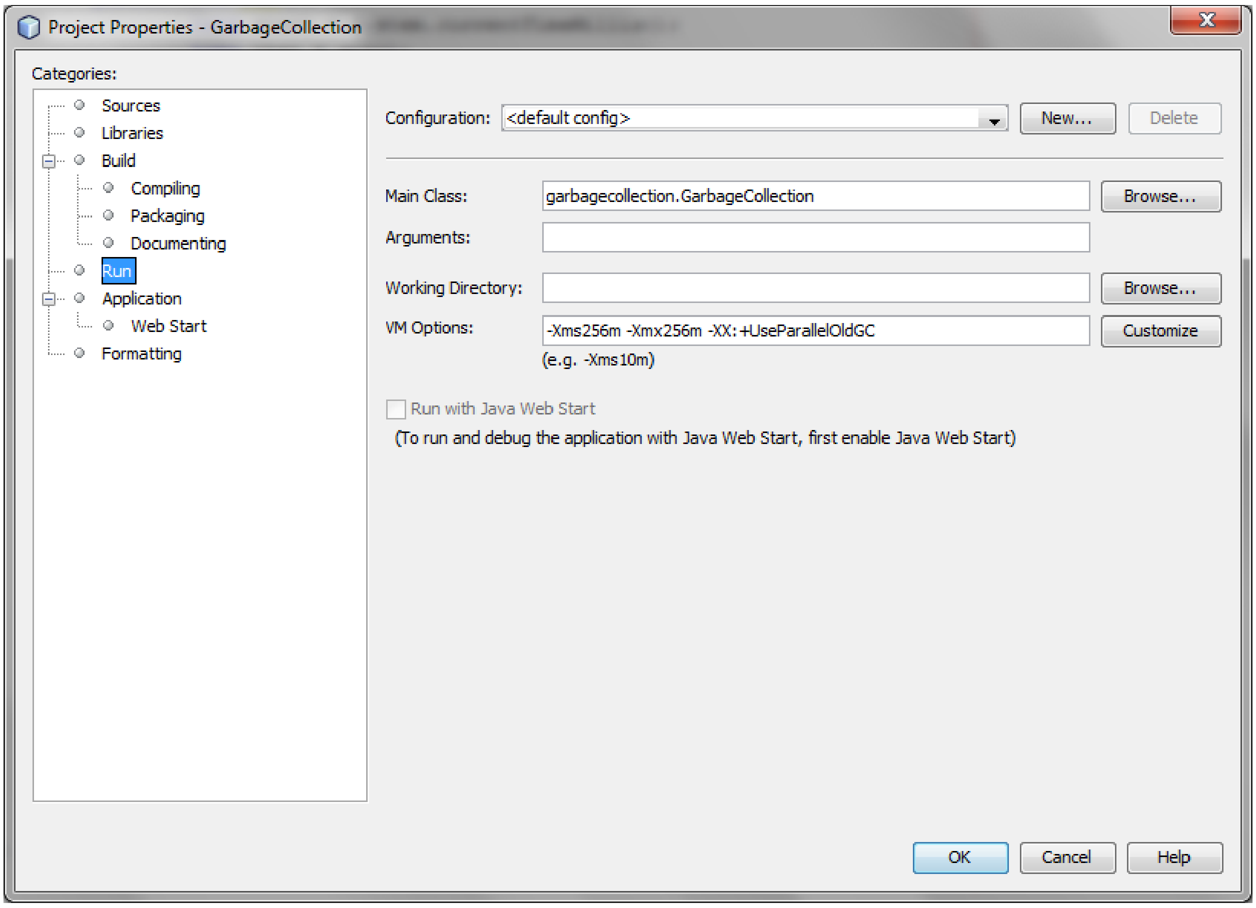
Histogram

Not supported for this JVM.

What is the Effect of Different Garbage Collectors on the Performance of Java Applications requiring Throughput and Response Time?

CASE-4

GARBAGE COLLECTION IS -XX:+UseParallelOldGC


Project Properties - GarbageCollection

Categories: Sources, Libraries, Build, Compiling, Packaging, Documenting, Run, Application, Web Start, Formatting

Configuration: <default config>

Main Class: garbagecollection.GarbageCollection

Arguments:

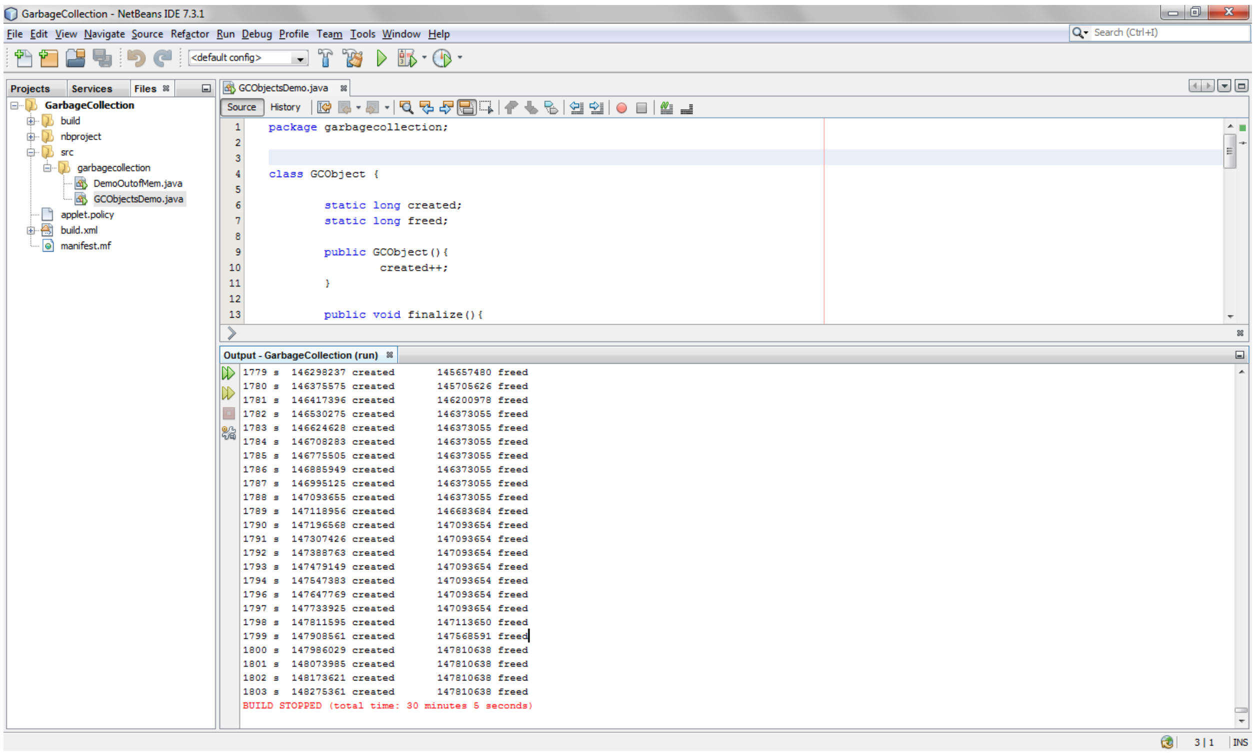
Working Directory:

VM Options: -Xms256m -Xmx256m -XX:+UseParallelOldGC

(e.g. -Xms10m)

Run with Java Web Start

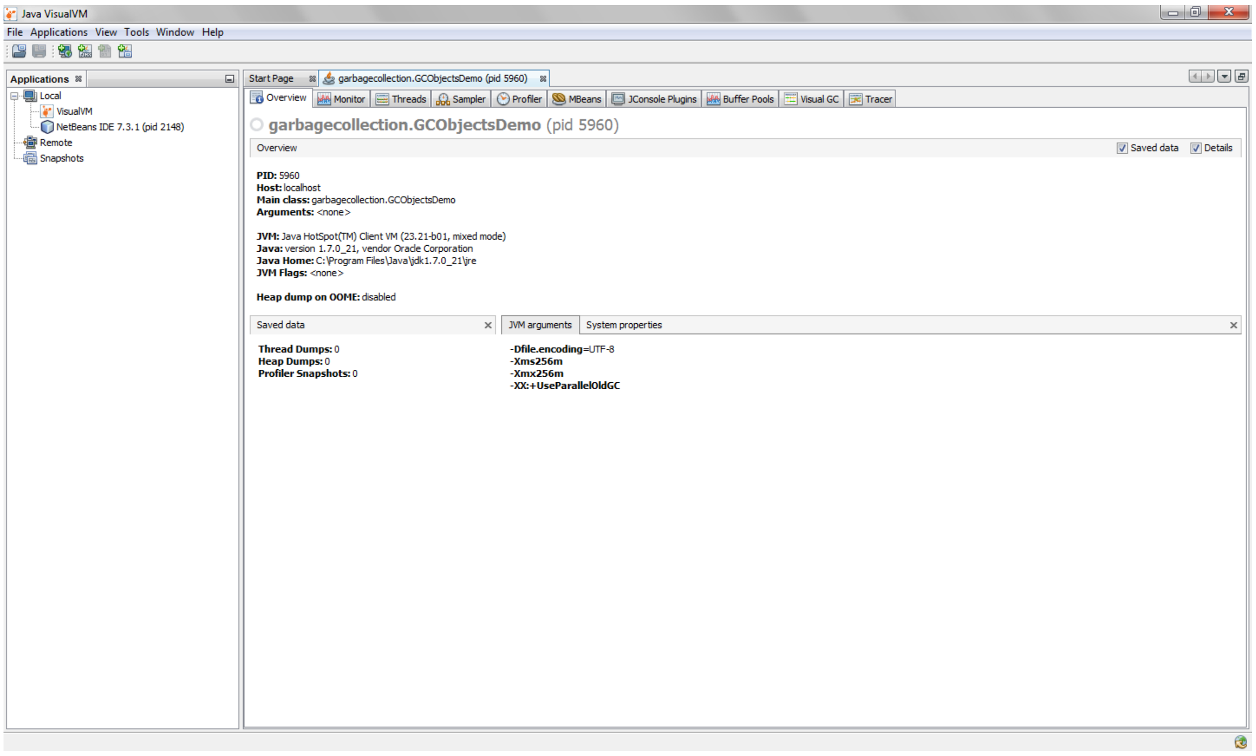
(To run and debug the application with Java Web Start, first enable Java Web Start)



```
package garbagecollection;


class GCObject {

        static long created;
        static long freed;

        public GCObject(){
                created++;
        }

        public void finalize(){
```

Output - GarbageCollection (run)

```
1779 s  146298237 created      145657480 freed
1780 s  146375575 created      145705626 freed
1781 s  146417396 created      146200978 freed
1782 s  146530275 created      146373055 freed
1783 s  146624628 created      146373055 freed
1784 s  146708283 created      146373055 freed
1785 s  146775505 created      146373055 freed
1786 s  146885949 created      146373055 freed
1787 s  146995125 created      146373055 freed
1788 s  147093655 created      146373055 freed
1789 s  147118956 created      146683684 freed
1790 s  147196568 created      147093654 freed
1791 s  147307426 created      147093654 freed
1792 s  147388763 created      147093654 freed
1793 s  147479149 created      147093654 freed
1794 s  147547383 created      147093654 freed
1796 s  147647769 created      147093654 freed
1797 s  147733925 created      147093654 freed
1798 s  147811595 created      147113650 freed
1799 s  147908561 created      147568591 freed
1800 s  147986029 created      147810638 freed
1801 s  148073985 created      147810638 freed
1802 s  148173621 created      147810638 freed
1803 s  148275361 created      147810638 freed
BUILD STOPPED (total time: 30 minutes 5 seconds)
```


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 2148)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 5960)
Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer
Saved data
Details

garbagecollection.GCObjectsDemo (pid 5960)

Overview

PID: 5960
Host: localhost
Main class: garbagecollection.GCObjectsDemo
Arguments: <none>

JVM: Java HotSpot(TM) Client VM (23.21-b01, mixed mode)
Java: version 1.7.0_21, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jdk1.7.0_21\jre
JVM Flags: <none>

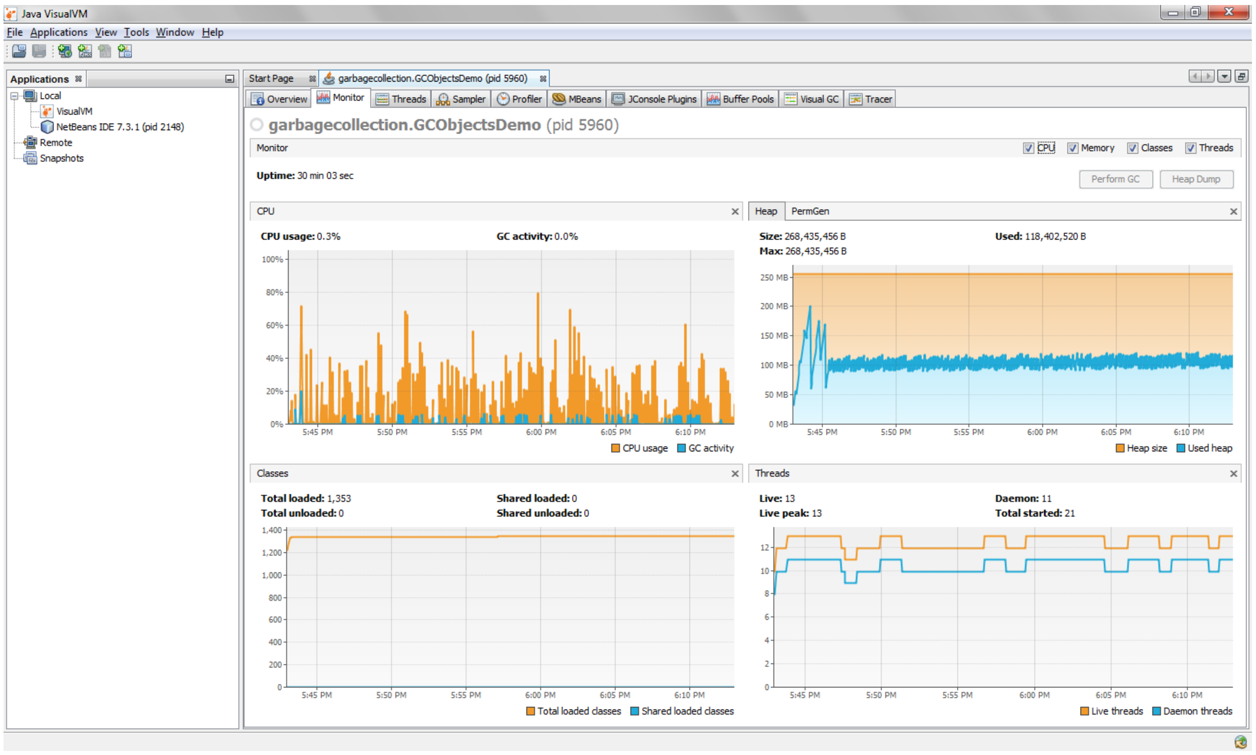
Heap dump on OOME: disabled

Saved data

Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0

JVM arguments System properties

-Dfile.encoding=UTF-8
-Xms256m
-Xmx256m
-XX:+UseParallelOldGC



garbagecollection.GCObjectsDemo (pid 5960)

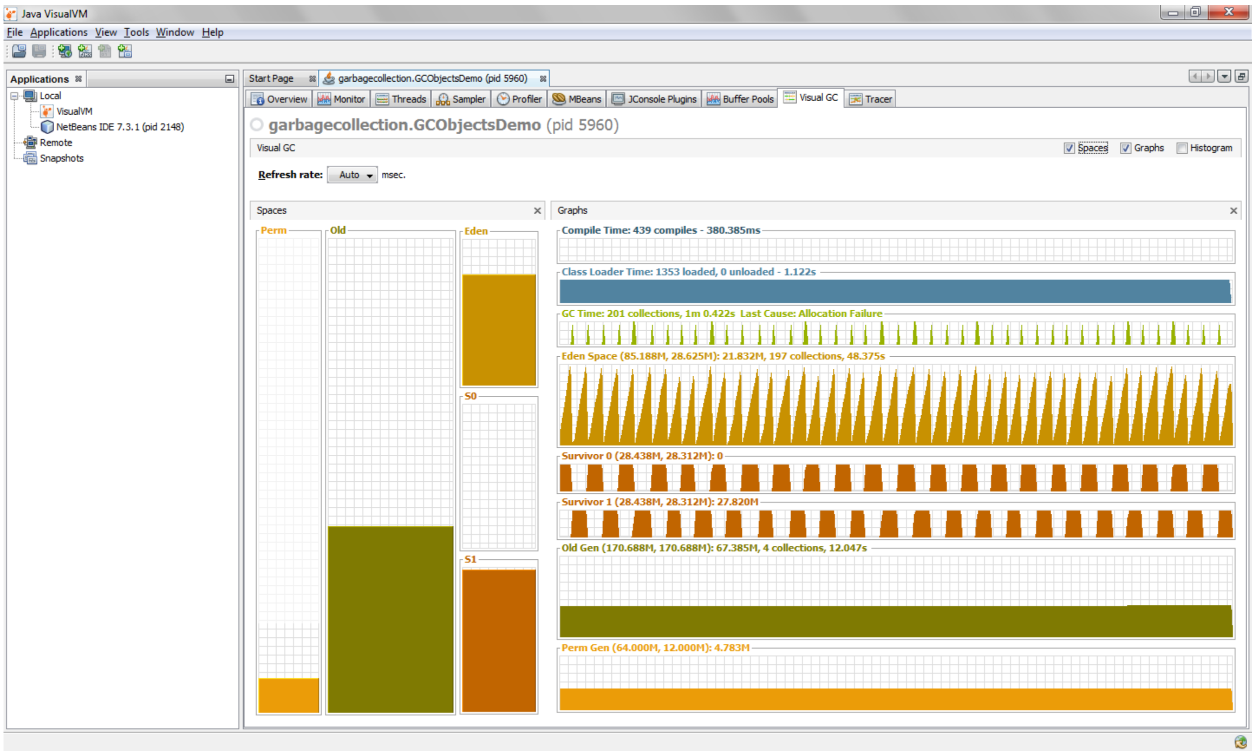
Uptime: 30 min 03 sec

CPU usage: 0.3% GC activity: 0.0%

Size: 268,435,456 B Used: 118,402,520 B
Max: 268,435,456 B

Total loaded: 1,353 Shared loaded: 0
Total unloaded: 0 Shared unloaded: 0

Live: 13 Daemon: 11
Live peak: 13 Total started: 21



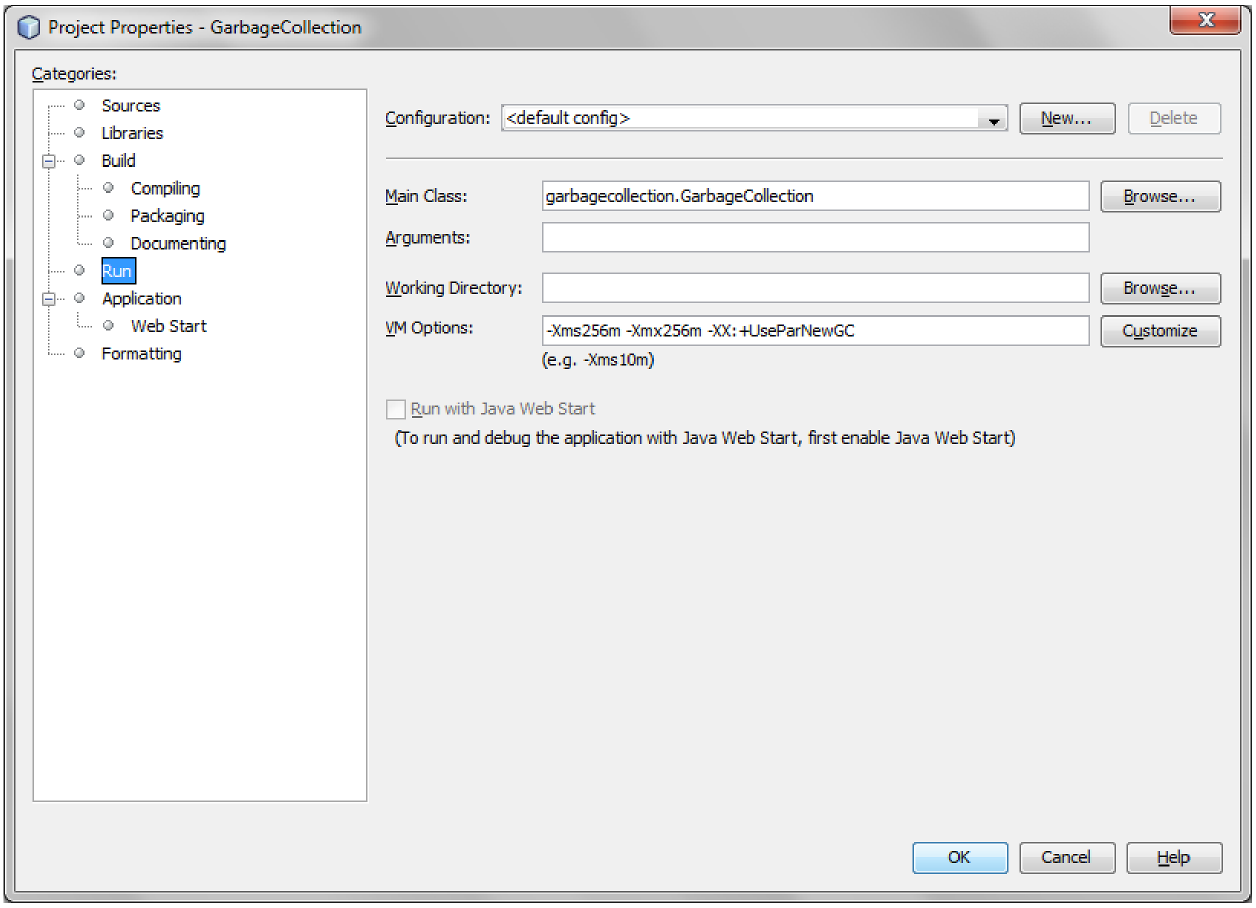
garbagecollection.GCObjectsDemo (pid 5960)

Refresh rate: Auto msec.

Compile Time: 439 compiles - 380.385ms
Class Loader Time: 1353 loaded, 0 unloaded - 1.122s
GC Time: 201 collections, 1m 0.422s Last Cause: Allocation Failure
Eden Space (85.188M, 28.625M): 21.832M, 197 collections, 48.375s
Survivor 0 (28.438M, 28.312M): 0
Survivor 1 (28.438M, 28.312M): 27.820M
Old Gen (170.688M, 170.688M): 67.385M, 4 collections, 12.047s
Perm Gen (64.000M, 12.000M): 4.783M

CASE-5

GARBAGE COLLECTION IS -XX:+UseParNewGC



Configuration: <default config>

Main Class: garbagecollection.GarbageCollection

Arguments:

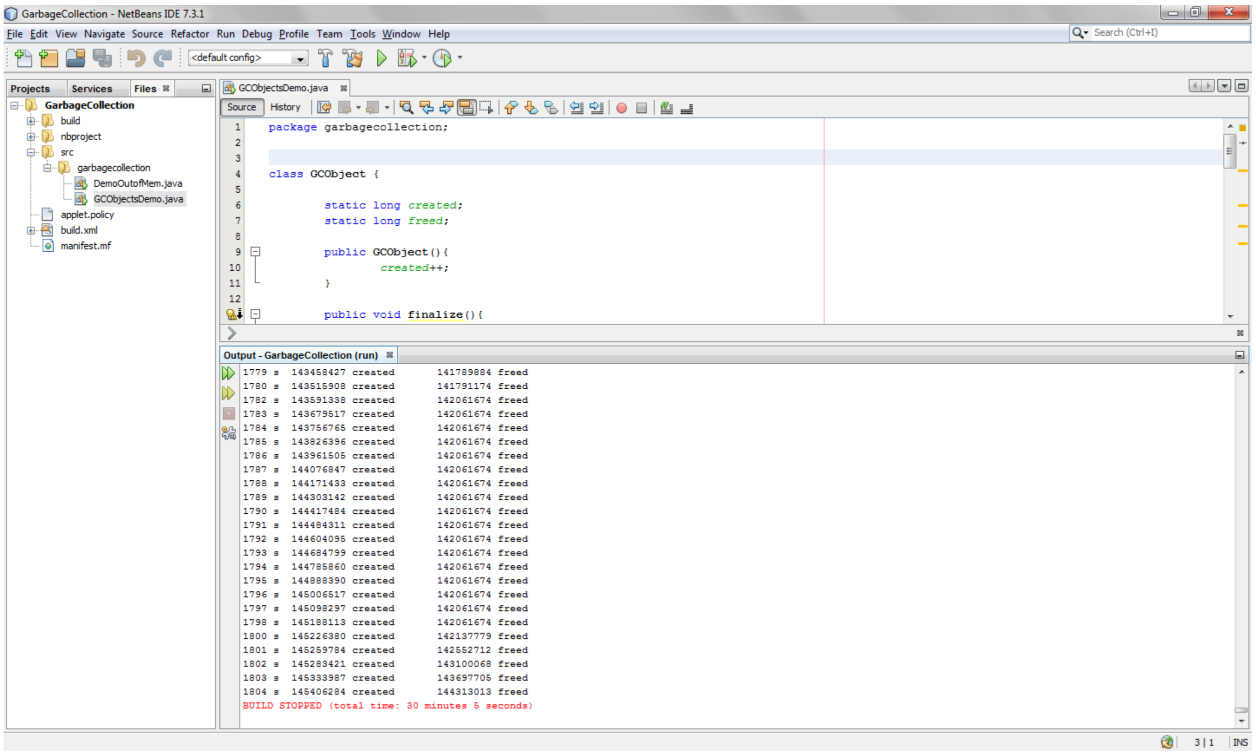
Working Directory:

VM Options: -Xms256m -Xmx256m -XX:+UseParNewGC

(e.g. -Xms10m)

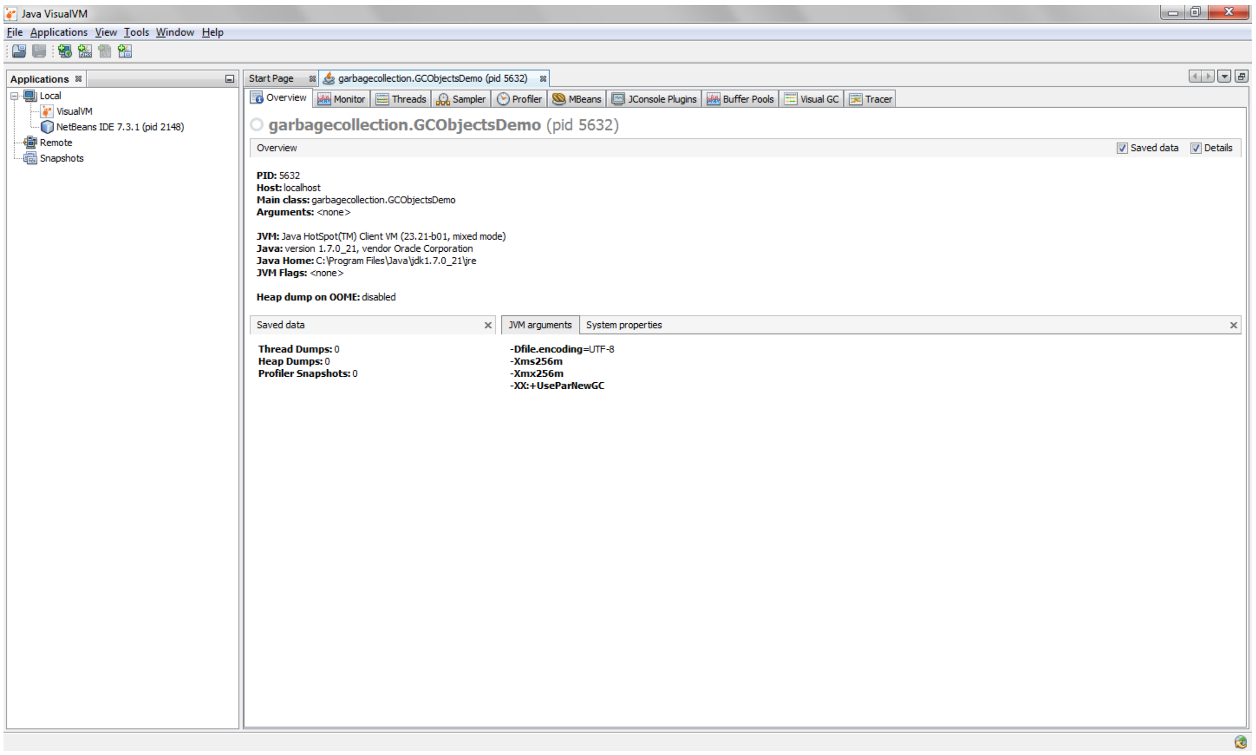
Run with Java Web Start

(To run and debug the application with Java Web Start, first enable Java Web Start)



```
package garbagecollection;


class GCObject {

        static long created;
        static long freed;

        public GCObject(){
                created++;
        }

        public void finalize(){
```

```
1779 s  143458427 created       141789884 freed
1780 s  143515908 created       141791174 freed
1782 s  143591338 created       142061674 freed
1783 s  143679517 created       142061674 freed
1784 s  143756765 created       142061674 freed
1785 s  143826396 created       142061674 freed
1786 s  143961505 created       142061674 freed
1787 s  144076847 created       142061674 freed
1788 s  144171433 created       142061674 freed
1789 s  144303142 created       142061674 freed
1790 s  144417484 created       142061674 freed
1791 s  144484311 created       142061674 freed
1792 s  144604095 created       142061674 freed
1793 s  144684799 created       142061674 freed
1794 s  144785860 created       142061674 freed
1795 s  144888390 created       142061674 freed
1796 s  145006517 created       142061674 freed
1797 s  145098297 created       142061674 freed
1798 s  145188113 created       142061674 freed
1800 s  145226380 created       142137779 freed
1801 s  145259784 created       142552712 freed
1802 s  145283421 created       143100068 freed
1803 s  145333987 created       143697705 freed
1804 s  145406284 created       144313013 freed
BUILD STOPPED (total time: 30 minutes 5 seconds)
```


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 2148)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 5632)
Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer
Saved data Details

garbagecollection.GCObjectsDemo (pid 5632)

Overview

PID: 5632
Host: localhost
Main class: garbagecollection.GCObjectsDemo
Arguments: <none>

JVM: Java HotSpot(TM) Client VM (23.21-b01, mixed mode)
Java: version 1.7.0_21, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jdk1.7.0_21\jre
JVM Flags: <none>

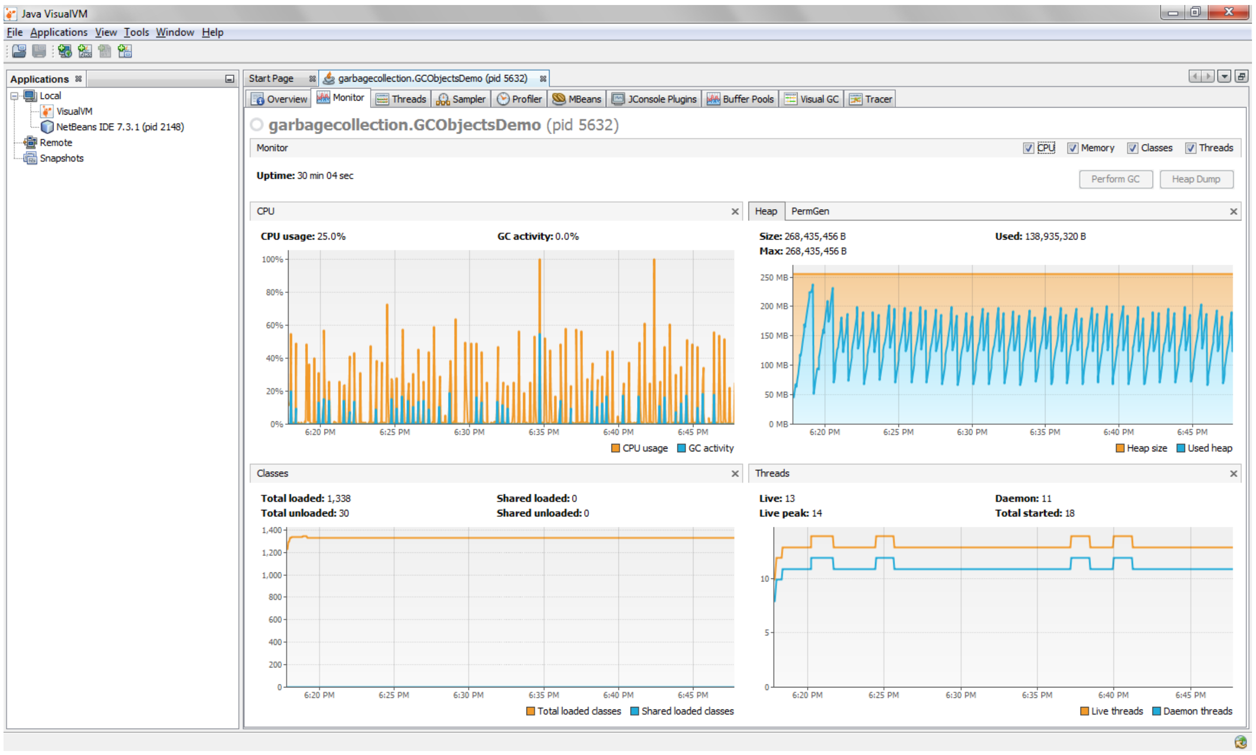
Heap dump on OOME: disabled

Saved data

Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0

JVM arguments | System properties

-**Dfile.encoding**=UTF-8
-**Xms256m**
-**Xmx256m**
-**XX:+UseParNewGC**


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 2148)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 5632)
Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer
Monitor
CPU Memory Classes Threads
Perform GC Heap Dump
CPU
Heap PermGen
CPU usage GC activity
Heap size Used heap
Classes
Threads
Total loaded classes Shared loaded classes
Live threads Daemon threads

garbagecollection.GCObjectsDemo (pid 5632)

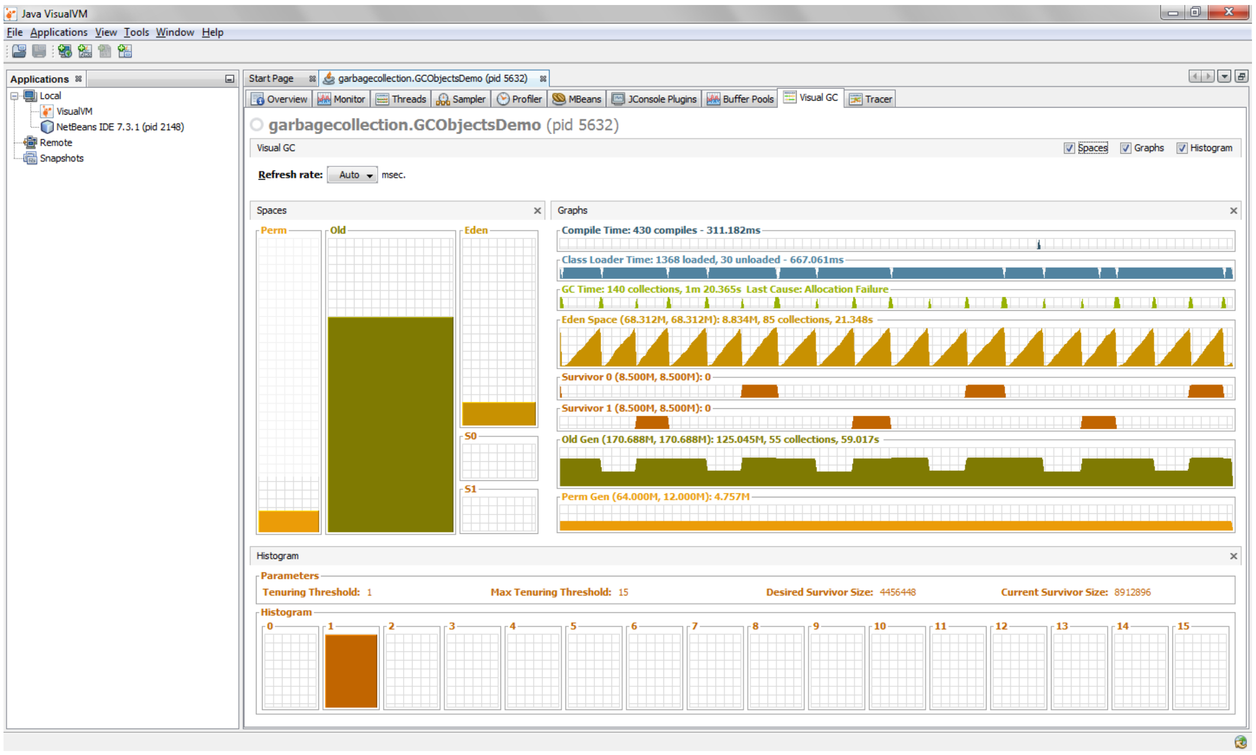
Uptime: 30 min 04 sec

CPU usage: 25.0% GC activity: 0.0%

Size: 268,435,456 B Used: 138,935,320 B
Max: 268,435,456 B

Total loaded: 1,338 Shared loaded: 0
Total unloaded: 30 Shared unloaded: 0

Live: 13 Daemon: 11
Live peak: 14 Total started: 18


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 2148)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 5632)
Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer
Visual GC
Spaces Graphs Histogram
Spaces
Perm
Old
Eden
S0
S1
Graphs
Histogram
Parameters
Histogram
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

garbagecollection.GCObjectsDemo (pid 5632)

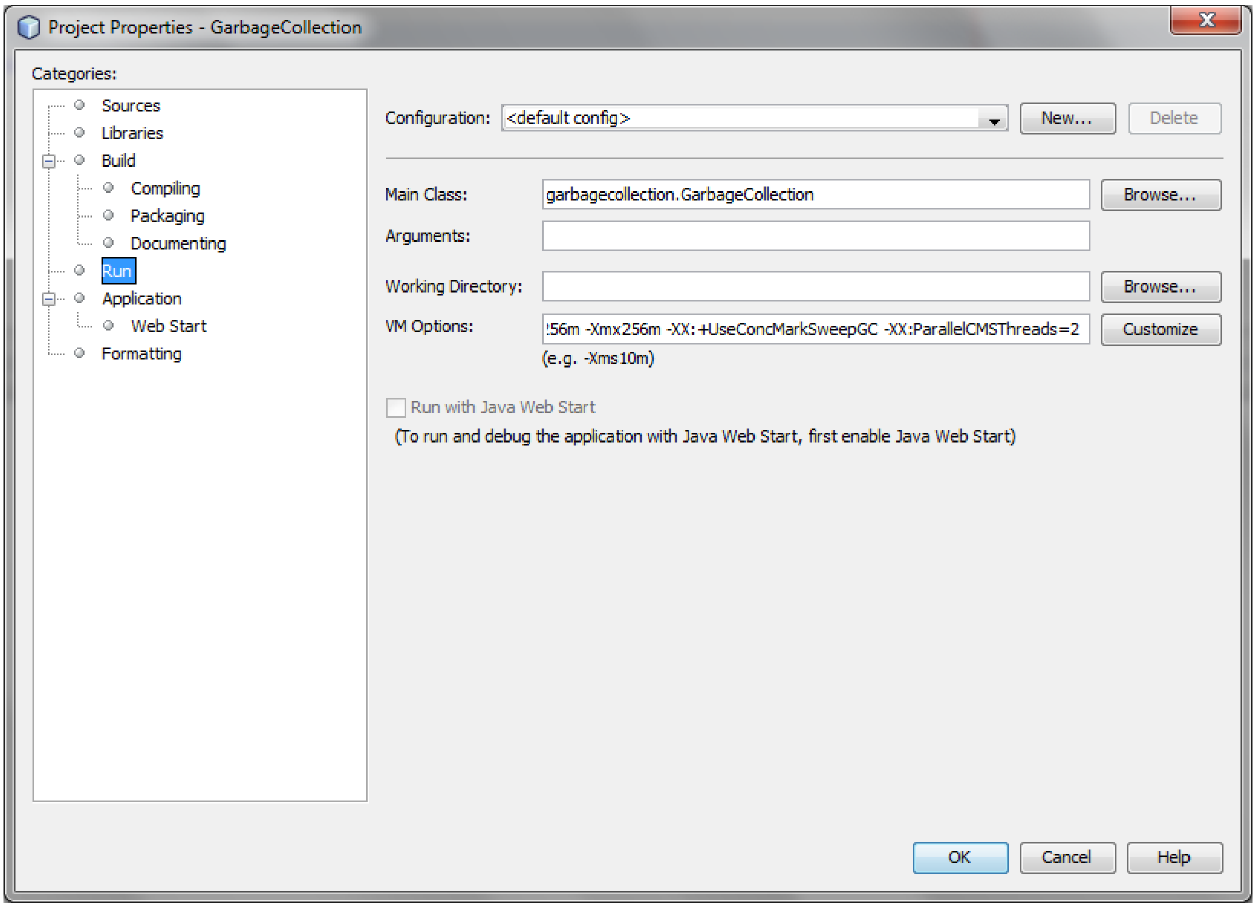
Refresh rate: Auto msec.

Compile Time: 430 compiles - 311.182ms
Class Loader Time: 1368 loaded, 30 unloaded - 667.061ms
GC Time: 140 collections, 1m 20.365s Last Cause: Allocation Failure
Eden Space (68.312M, 68.312M): 8.834M, 85 collections, 21.348s
Survivor 0 (8.500M, 8.500M): 0
Survivor 1 (8.500M, 8.500M): 0
Old Gen (170.688M, 170.688M): 125.045M, 55 collections, 59.017s
Perm Gen (64.000M, 12.000M): 4.757M

Tenuring Threshold: 1 Max Tenuring Threshold: 15 Desired Survivor Size: 4456448 Current Survivor Size: 8912896

CASE-6

GARBAGE COLLECTION IS -XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2



Configuration: <default config>

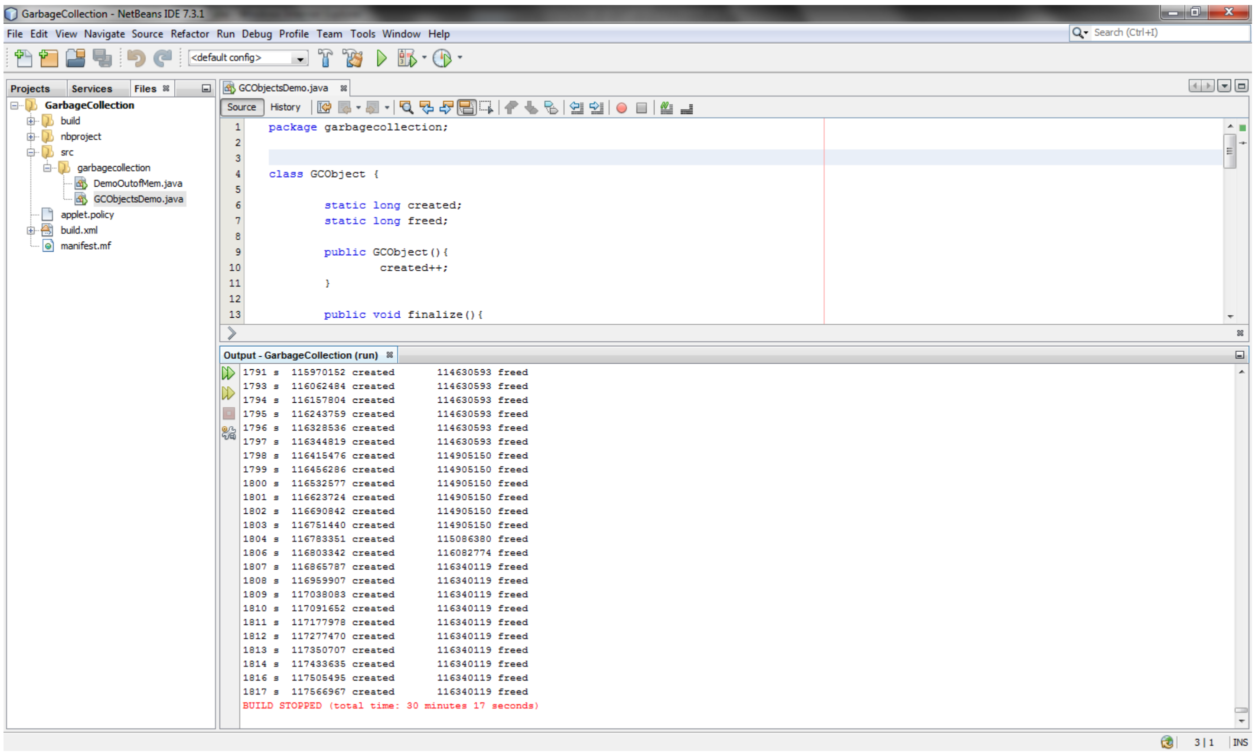
Main Class: garbagecollection.GarbageCollection

Arguments:

Working Directory:

VM Options: !56m -Xmx256m -XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2
(e.g. -Xms10m)

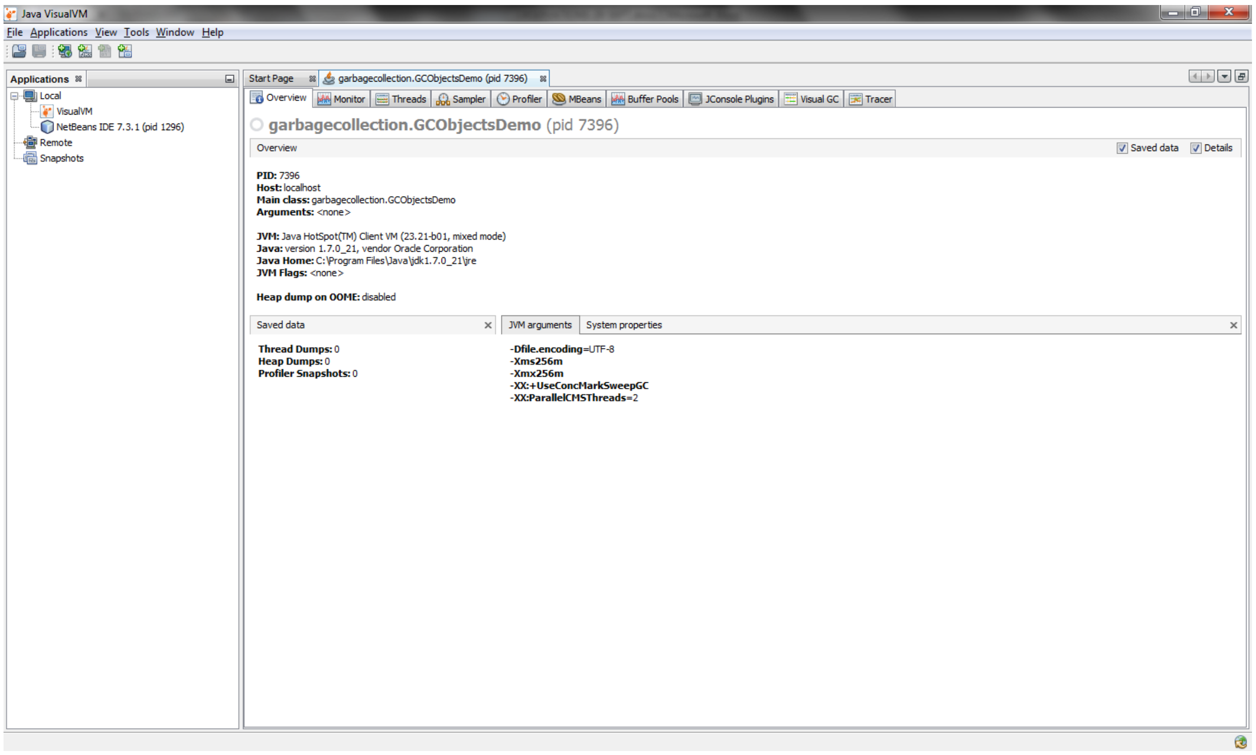
Run with Java Web Start
(To run and debug the application with Java Web Start, first enable Java Web Start)



```
package garbagecollection;


class GCObject {

        static long created;
        static long freed;

        public GCObject(){
                created++;
        }

        public void finalize(){
```

Output - GarbageCollection (run)

```
1791 s  115970152 created       114630593 freed
1793 s  116062484 created       114630593 freed
1794 s  116157804 created       114630593 freed
1795 s  116243759 created       114630593 freed
1796 s  116328536 created       114630593 freed
1797 s  116344819 created       114630593 freed
1798 s  116415476 created       114905150 freed
1799 s  116456286 created       114905150 freed
1800 s  116532577 created       114905150 freed
1801 s  116623724 created       114905150 freed
1802 s  116690842 created       114905150 freed
1803 s  116751440 created       114905150 freed
1804 s  116783351 created       115086380 freed
1806 s  116803342 created       116082774 freed
1807 s  116865787 created       116340119 freed
1808 s  116959907 created       116340119 freed
1809 s  117038083 created       116340119 freed
1810 s  117091652 created       116340119 freed
1811 s  117177978 created       116340119 freed
1812 s  117277470 created       116340119 freed
1813 s  117350707 created       116340119 freed
1814 s  117433635 created       116340119 freed
1816 s  117505495 created       116340119 freed
1817 s  117566967 created       116340119 freed
BUILD STOPPED (total time: 30 minutes 17 seconds)
```


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 1296)
Remote
Snapshots
Start Page
garbagecollection.GCObjectsDemo (pid 7396)
Overview Monitor Threads Sampler Profiler MBeans Buffer Pools JConsole Plugins Visual GC Tracer
Saved data Details

garbagecollection.GCObjectsDemo (pid 7396)

Overview

PID: 7396
Host: localhost
Main class: garbagecollection.GCObjectsDemo
Arguments: <none>

JVM: Java HotSpot(TM) Client VM (23.21-b01, mixed mode)
Java: version 1.7.0_21, vendor Oracle Corporation
Java Home: C:\Program Files\Java\jdk1.7.0_21\jre
JVM Flags: <none>

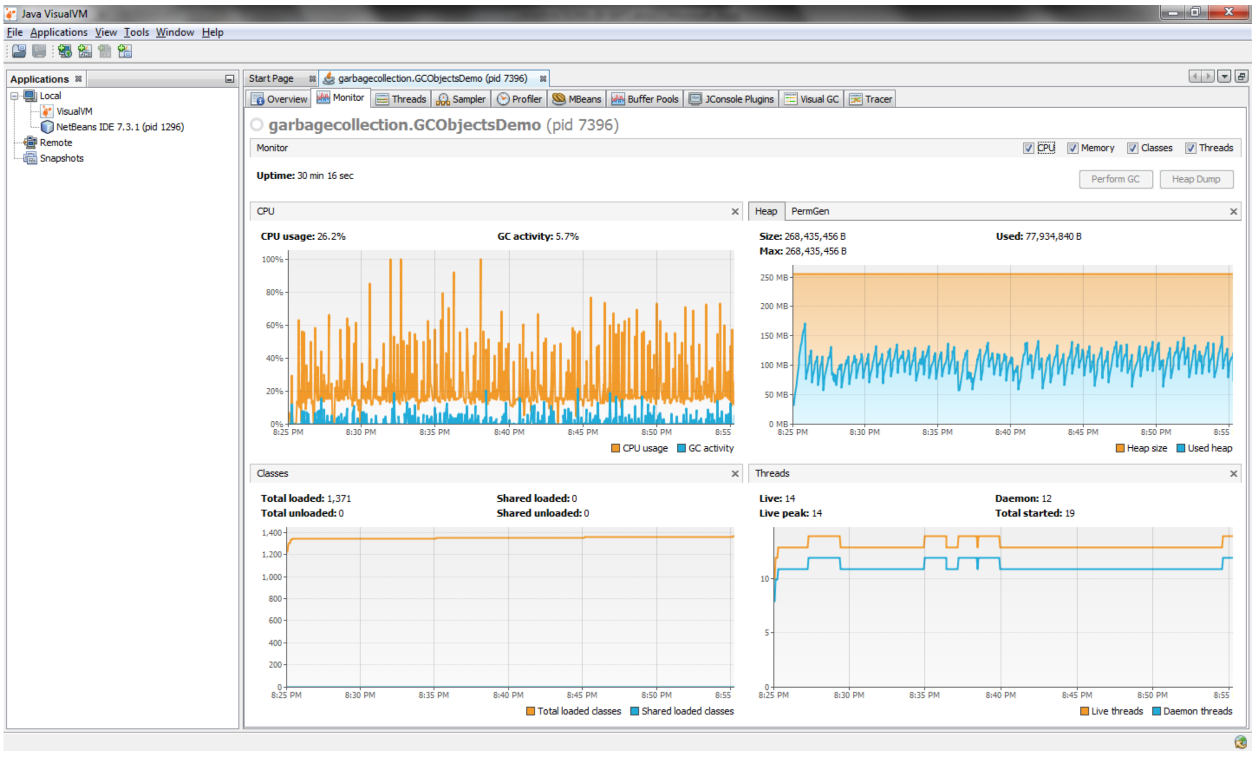
Heap dump on OOME: disabled

Saved data

Thread Dumps: 0
Heap Dumps: 0
Profiler Snapshots: 0

JVM arguments | System properties

-Dfile.encoding=UTF-8
-Xms256m
-Xmx256m
-XX:+UseConcMarkSweepGC
-XX:ParallelCMSThreads=2


garbagecollection.GCObjectsDemo (pid 7396)

Uptime: 30 min 16 sec

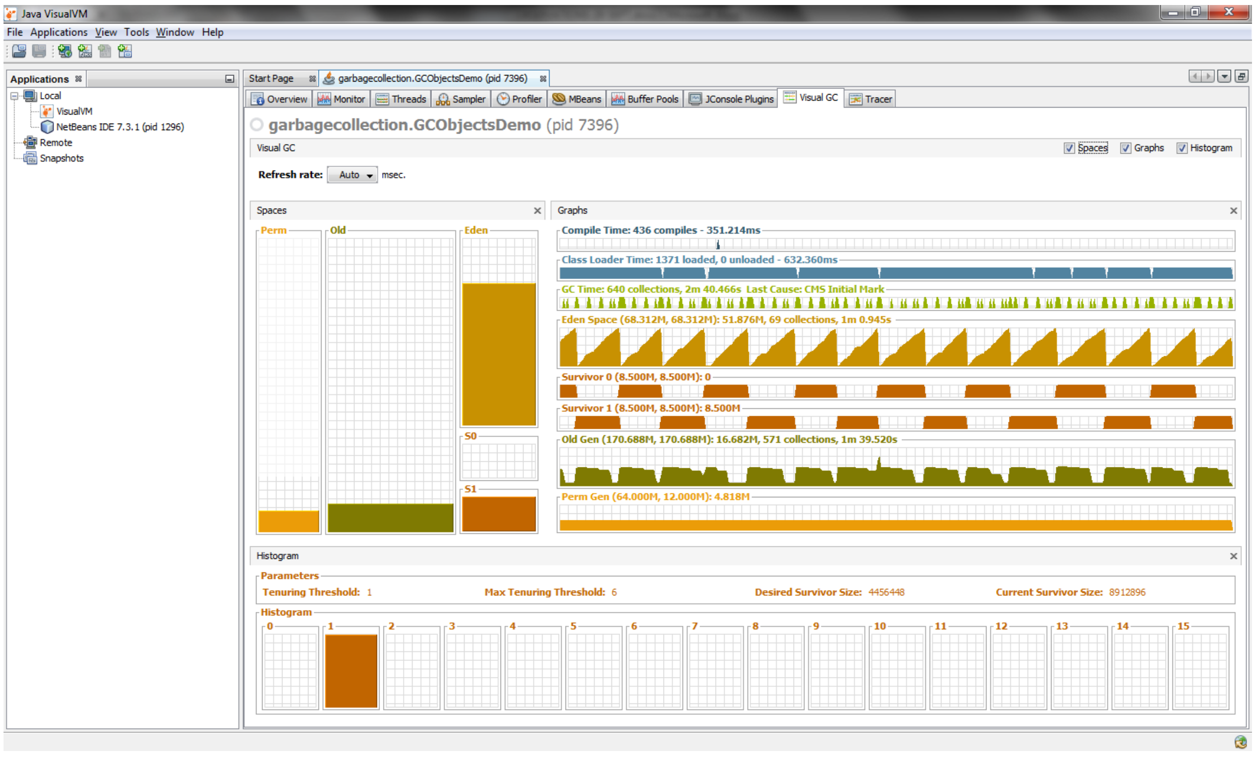
CPU usage: 26.2% GC activity: 5.7%

Size: 268,435,456 B Used: 77,934,840 B
Max: 268,435,456 B

Total loaded: 1,371 Shared loaded: 0
Total unloaded: 0 Shared unloaded: 0

Live: 14 Daemon: 12
Live peak: 14 Total started: 19

Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans IDE 7.3.1 (pid 1296)
Remote
Snapshots
Start Page
Overview Monitor Threads Sampler Profiler MBeans Buffer Pools JConsole Plugins Visual GC Tracer
Monitor
CPU Memory Classes Threads
Perform GC Heap Dump
CPU
Heap PermGen
Classes
Threads
CPU usage GC activity
Heap size Used heap
Total loaded classes Shared loaded classes
Live threads Daemon threads


garbagecollection.GCObjectsDemo (pid 7396)

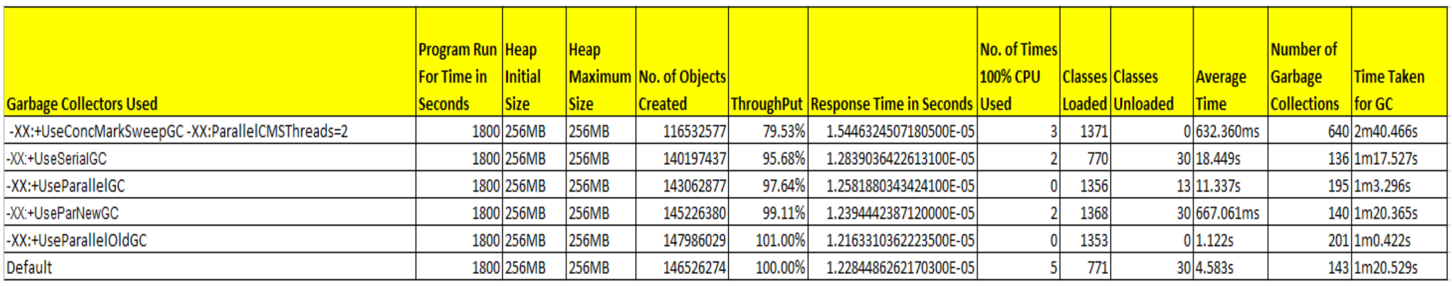
Refresh rate: Auto msec.

Compile Time: 436 compiles - 351.214ms
Class Loader Time: 1371 loaded, 0 unloaded - 632.360ms
GC Time: 640 collections, 2m 40.466s Last Cause: CMS Initial Mark
Eden Space (68.312M, 68.312M): 51.876M, 69 collections, 1m 0.945s
Survivor 0 (8.500M, 8.500M): 0
Survivor 1 (8.500M, 8.500M): 8.500M
Old Gen (170.688M, 170.688M): 16.682M, 571 collections, 1m 39.520s
Perm Gen (64.000M, 12.000M): 4.818M

Parameters
Tenuring Threshold: 1 Max Tenuring Threshold: 6 Desired Survivor Size: 4456448 Current Survivor Size: 8912896

Appendix 2

The results of the experiment:



Garbage Collectors Used	Program Run For Time in Seconds	Heap Initial Size	Heap Maximum Size	No. of Objects Created	ThroughPut	Response Time in Seconds	No. of Times 100% CPU Used	Classes Loaded	Classes Unloaded	Average Time	Number of Garbage Collections	Time Taken for GC
-XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2	1800	256MB	256MB	116532577	79.53%	1.5446324507180500E-05	3	1371	0	632.360ms	640	2m40.466s
-XX:+UseSerialGC	1800	256MB	256MB	140197437	95.68%	1.2839036422613100E-05	2	770	30	18.449s	136	1m17.527s
-XX:+UseParallelGC	1800	256MB	256MB	143062877	97.64%	1.2581880343424100E-05	0	1356	13	11.337s	195	1m3.296s
-XX:+UseParNewGC	1800	256MB	256MB	145226380	99.11%	1.2394442387120000E-05	2	1368	30	667.061ms	140	1m20.365s
-XX:+UseParallelOldGC	1800	256MB	256MB	147986029	101.00%	1.2163310362223500E-05	0	1353	0	1.122s	201	1m0.422s
Default	1800	256MB	256MB	146526274	100.00%	1.2284486262170300E-05	5	771	30	4.583s	143	1m20.529s

15. Bibliography

- "Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine." *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*. N.p., n.d. Web. 28 May 2013.
- "How to Handle Java Finalization's Memory-Retention Issues." *How to Handle Java Finalization's Memory-Retention Issues*. N.p., n.d. Web. 22 July 2013.
- "Java Garbage Collection Basics." *Java Garbage Collection Basics*. N.p., n.d. Web. 26 June 2013.
- "Question of the Month: 1.4.1 Garbage Collection Algorithms, January 29th, 2003." *Question of the Month: 1.4.1 Garbage Collection Algorithms, January 29th, 2003*. N.p., n.d. Web. 23 July 2013.
- "14) Garbage Collection." *Java Tutorial Hub*. N.p., n.d. Web. 23 May 2013.
- "Calculating Throughput and Response Time » Javidjamae.com." *Calculating Throughput and Response Time » Javidjamae.com*. N.p., n.d. Web. 21 May 2013.
- "Formative Writing for Student Learning." *Turnitin*. N.p., n.d. Web. 25 July 2013.
- "Garbage Collection Ergonomics." *Garbage Collection Ergonomics*. N.p., n.d. Web. 22 May 2013.
- "Garbage Collection: Serial vs. Parallel vs. Concurrent-Mark-Sweep." *Who Are We?* N.p., n.d. Web. 25 June 2013.
- "How to Handle Java Finalization's Memory-Retention Issues." *How to Handle Java Finalization's Memory-Retention Issues*. N.p., n.d. Web. 22 July 2013.

- "Java Garbage Collection Basics." *Java Garbage Collection Basics*. N.p., n.d. Web. 22 May 2013.
- "Java Memory Management." *JavaWorld*. N.p., n.d. Web. 21 May 2013.
- "Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning." *Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning*. N.p., n.d. Web. 23 June 2013.
- "Java SE at a Glance." *Java SE*. N.p., n.d. Web. 19 June 2013.
- "Java VisualVM." *VisualVM*. N.p., n.d. Web. 24 July 2013.
- "NetBeans IDE." *Welcome to NetBeans*. N.p., n.d. Web. 24 July 2013.
- "NetBeans IDE." *Welcome to NetBeans*. N.p., n.d. Web. 24 July 2013.
- "Understanding Java Garbage Collection." *CUBRID*. N.p., n.d. Web. 26 May 2013.
- "What the Frequency of the Garbage Collection in Java?" - *Stack Overflow*. N.p., n.d. Web. 22 May 2013.