Hi everyone.

**Memory management**

Today our topic is Garbage collection in Java. But before going to that let's begin with a quick reminder of memory management in different programming languages.

What's it about?

Obviously that each entity used in code like a variable, array, etc. requires some memory to be allocated for it. But after it's not needed anymore allocated memory should be released, otherwise, it will end up really soon.

We have different types of memory management in various languages, but in short, we can say that some languages use automatic memory management usually with some kind of garbage collectors, and others need to be controlled manually when in program code memory should be allocated and deallocated for each entity.

**What is garbage?**

As you may know in Java when you programming something you don't have to manually allocate and deallocate memory for all variables and objects that your program creates and use during its runtime.

This is because Java uses an automatic memory management system called a garbage collector that is a part of JVM where all programs run.

And let's begin with the definition of garbage. In short, garbage is considered objects that are not reachable from Garbage Collector roots.

**As a matter of fact, all GC implementations in the HotSpot JVM are tracing collectors.** GC identifies all live objects by traversing the objects graph. In addition, objects visited and marked as alive won't be garbage collected. To be able to traverse the graph, starting points are necessary. Thus, **GC roots are starting points for tracing collectors**.

**GC roots**

**System\Application Classloader**

But what are those roots? Let's discuss.

The first one is the so-called system aka application class loader. When the java program starts, compiled classes are loaded by the classloaders, your classes are usually loaded by the Application classloader.

There can be custom classloaders which are also can be considered roots but unlike system classloaders, they can be garbage collected, and then all their roots are gone.

So, classes loaded by the application classloader and their static variables are roots.

**Active Java Threads**

The active threads, local variables in thread stack. The object that has reference in the thread stack is defined as reachable and not considered for garbage collection. Regularly this is the place where your program produces the objects that will be garbage collected in the future when threads complete their work and stack is drained and there are no live links to early created objects.

**JNI References**

**JNI (Java Native Interface)** is a foreign function interface programming framework that enables Java code running in a Java virtual machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++, and assembly.

I found a small demo out in the wild that will crash JVM with OOM(Out of memory). It was specially made for that, here the java program calls the "C" program that keeps creating objects forever.

So, native code Java objects created for JNI calls; contains local variables, parameters to JNI methods, and global JNI references and are considered as roots.

Furthermore, there is no documentation per JVM about which specific objects are GC roots. And they may vary but we discussed the main GC roots.

**Garbage collection.**

Let's head to garbage collection itself.

No matter what implementation of the garbage collector we use, to clean up the memory, a short pause needs to happen. Those pauses are also called stop-the-world events or STW in short.

The first step of the cycle starts when your [JVM threads](https://sematext.com/glossary/jvm-threads/) are started and your business code is working. This is where your application code is running. At a certain point in time, an event happens that triggers garbage collection. To clear the memory, application threads have to be stopped. This is where the work of your application stops and the next steps start. The garbage collector marks objects that are no longer used and reclaims the memory. Finally, an optional step of heap resizing may happen if possible. Then the circle starts again, application threads are started.

The full cycle of garbage collection is called the **epoch**.

The key when running JVM applications and tuning the garbage collector is to keep the application threads running for as long as possible. That means that the pauses caused by the garbage collector should be minimal.

**Serial GC.**

Serial is the simplest GC implementation, as it basically works with a single thread. As a result, this GC implementation freezes all application threads when it runs. Therefore, it's not a good idea to use it in multi-threaded applications, like server environments.

**Parralel GC.**

It's the default GC of the JVM, and sometimes called Throughput Collectors. Unlike Serial Garbage Collector, it uses multiple threads for managing heap space, but it also freezes other application threads while performing GC.

If we use this GC, we can specify maximum garbage collection threads and pause time, throughput, and footprint (heap size)

**CMS GC.**

The Concurrent Mark Sweep (CMS) implementation uses multiple garbage collector threads for garbage collection. It's designed for applications that prefer shorter garbage collection pauses, and can afford to share processor resources with the garbage collector while the application is running.

Simply put, applications using this type of GC respond slower on average, but don't stop responding to perform garbage collection.

But this Garbage collector is considered as a derpecated and should be replaced with G1 GC.

**Generations.**

The second thing that we need to talk about is generations. Java garbage collectors are generational, which means that they work under certain principles:

* Young data will not survive long
* Data that is old will continue to persist in memory

That’s why JVM heap memory is divided into generations:

* **Young generation** is divided into two sections called**Eden space**and **Survivor space**
* Old generation, or **Tenured space**.

**G1 GC.**

However, the G1 garbage collector goes a step further and divides the heap into something called **regions**. A **region** is a small, independent heap that can be dynamically set to be of Eden, Survivor or Tenured type:

Such architecture allows for different operations. First of all, because the generations are divided they can be collected in portions that affect latency, making the garbage collector faster for old generation space. Such heaps can be easily defragmented and dynamically resized. The cost of maintaining such heap architecture is higher compared to traditional heap architecture. It requires more CPU and memory.

When performing garbage collections, G1 shows a concurrent global marking phase (i.e. phase 1, known as Marking) to determine the liveness of objects throughout the heap.

After the mark phase is complete, G1 knows which regions are mostly garbage. It collects in these areas first, which usually yields a significant amount of free space (i.e. phase 2, known as Sweeping). That's why this method of garbage collection is called Garbage-First.

G1 (Garbage First) Garbage Collector is designed for applications running on multi-processor machines with large memory space. It's available from the JDK7 Update 4 and in later releases.

G1 collector will replace the CMS collector since it's more performance efficient.

To enable the G1 Garbage Collector, we can use the following argument:

java -XX:+UseG1GC -jar Application.java

**Demo**

**Comparison.**

Let's head next and try to take a closer look at different garbage collectors. We will run the same application that just constantly produces integer arrays and right after will erase the link to it, so it can be considered garbage.

To monitor the garbage collector's work we will use visualVM with vusialGC plugin.

First of all, let's try to look at different graphs, and see what they describe. This graph with pikes is our heap, it says that we have 250 MB of memory allocated for our heap, be we don't fill it more than 75 MB of it.

Because of the help of a garbage collector, such nice and smooth pikes usually mean that we don't have problems with memory leaks, and the garbage collector is able to perform stable cleaning each time required.

Now let us compare them. What difference we may find?

I'll open the visualGC plugin tab which is a plugin for visualVM.

The first difference that I see is the time of "stop the world" pauses between serial and parallel GC. This is because parallel GC uses multiple threads for collection.

What else? You may notice that serial GC survivor spaces look pretty the same, they don't have any spikes, etc. Unlike parallel, because parallel doesn't synchronize cleaning between threads it just divides the heap, and each thread cleans its space, which causes defragmentation of memory.

Then let's see what is happening with the G1 collector.

As you may see it uses a bit more heap space, that happens because of the way it handles it, remember about regions, they require more space. We don't see any space allocated in the first survivor region since G1 use only three generations (Eden, Survivor, Tenured). And tenured will be mostly empty in this demo since there are no objects that will last long enough.

**Bad practice.**

**Automated garbage collection is good but it’s not bulletproof.**

•System.gc()

Let us talk about how can we make life harder for us and for garbage collectors, it is possible to throw away the advantages that automation memory management gives to us.

For example, if we have problems with a growing heap during runtime and we do not want to find out the reason we may use "System.gc" calls and hope that problem evaporates. But it may not. And it's not a durable solution to resolve memory usage problems.

•finalize()

Method finalize, could be easily used to make your GC work harder. Whenever a class' finalize() method is overridden, then objects of that class aren't instantly garbage collected. Instead, the GC queues them for finalization, which occurs at a later point in time.

•Static variables

As you remember static variables are GC roots, so let's fill them with all those tons of objects that produced during runtime, and make it impossible to collect them.

•Unclosed resources

Whenever we make a new connection or open a stream, the JVM allocates memory for these resources. A few examples of this include database connections, input streams, and session objects.

Forgetting to close these resources can block the memory, thus keeping them out of the reach of the GC. This can even happen in case of an exception that prevents the program execution from reaching the statement that's handling the code to close these resources.

•Inner Classes That Reference Outer Classes

Every non-static Inner Class has, by default, an implicit reference to its containing class. If we use this inner class' object in our application, then even after our containing class' object goes out of scope, it won't be garbage collected.

•Improper equals() and hashCode() Implementations

When defining new classes, a very common oversight is not writing proper overridden methods for the equals() and hashCode() methods.

HashSet and HashMap use these methods in many operations, and if they're not overridden correctly, they can become a source for potential memory leak problems.

Like an example hibernate that help you work with databases heavily relies on equals and hashcode methods, if entities used by hibernate do not override equals and hashcode hibernate can start pile them up in cash thinking they all unique.

**Monitoring tools**

To continue the topic of what could go wrong with your garbage collection, I'd like to describe a few ways to monitor and diagnose problems.

Usually, if such a problem occurs it could be hard to find the reason, it will be not like in the examples that we saw earlier. So you may need to diagnose your GC, heap, etc. Let's talk about some tools and technics that may help.

First of all, you may use additional JVM options, to enable garbage collection logs. It's built-in and easy to use, but you may have to figure out what exactly options you need for your current situation. I guess most used is heap dump on out of memory error. If your app keeps running in out-of-memory error this dump will help to figure out the reason.

Then jstat, it's command line tools that are built in some JDK, but not all of them, it depends.

cd C:\Program Files\Java\jdk1.8.0\_251\bin

jps

jstat -gcutil -t 11016 1000 10

Then we may use some tools with a user interface which usually helps to understand provided statistics of our JVM, heap, and garbage collection process. But it could be hard to monitor that way applications running on remote servers. You will have to additionally open ports on a remote server, provide configuration for JVM, etc.

**Tuning**

Ok, we've been talking about issues related to memory leaks in java, and about monitoring tools that could help us to resolve them, but actually, it's quite rare to resolve such problems related to garbage collection. I hope it will be rare for you :)

Usually, if an engineer needs to do something with GC it's tuning. It could be related to a specific program that you currently working on.

Like you really need to make stop the world pauses as short as possible. Or you found out that for some generation you need to add extra space in a heap like you have a lot of tenured\old objects in your app during runtime but they are stable, they not changing their number after reaching some threshold and to reduce the load on garbage collector you may increase such space for old objects.

So, after you decide that you need to adjust some settings for GC you will

start by looking at how your application behaves, what events fill up the memory space, and what space is filled. Remember that:

* Assigned objects in the Eden generation are moved to Survivor space
* Assigned objects in the Survivor space are moved to Tenured generation if the counter is high enough or the counter is increased.
* Assigned objects in the Tenured generation are ignored most of the time, garbage collection goes there really rare.

You need to be sure you understand what is happening inside your application’s heap, and keep in mind what causes the garbage collection events. That will help you understand your application’s memory needs and how to improve garbage collection.

There are various ways to tune garbage collectors, usually, everyone starts with simple tuning like heap adjustment. Because it often helps without the need for further investigation. And initial heap size often can be not configured to fit it current environment.

Then there are options to choose the GC that will be used, and depending on that chose a lot of additional adjustments. You can control generations, thresholds, number of threads that will be used by garbage collectors, etc., etc. I'm not going to describe them all because they all may differ for the garbage collectors and what problems need to be solved. So just keep in mind that garbage collectors are a highly tunable tool.