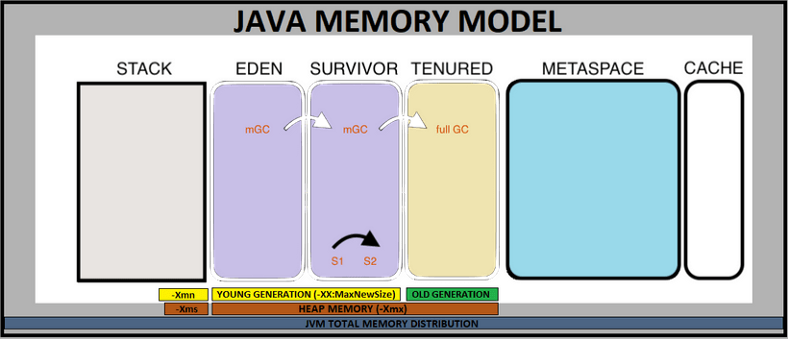
**[Top Performance issues every Java developer/architect must know — part](https://medium.com/p/fc1ad6e1644b?source=post_stats_page-------------------------------------)3 — Memory**

[[](https://javarevisited.blogspot.com/2019/04/top-5-courses-to-learn-jvm-internals.html)](https://javarevisited.blogspot.com/2019/04/top-5-courses-to-learn-jvm-internals.html)

Hello everyone. This article is a continuation of the following articles

**part 1 — Database problems**

**[Top Performance issues every developer/architect must know — part 1](https://dineshchandgr.medium.com/top-performance-issues-every-developer-architect-must-know-part-1-fc1ad6e1644b" \t "_blank)**

[Hello everyone. This is one of the interesting articles I am writing as it talks about the performance issue in an…](https://dineshchandgr.medium.com/top-performance-issues-every-developer-architect-must-know-part-1-fc1ad6e1644b" \t "_blank)

[dineshchandgr.medium.com](https://dineshchandgr.medium.com/top-performance-issues-every-developer-architect-must-know-part-1-fc1ad6e1644b" \t "_blank)

**part 2 — Concurrency problems**

**[Top Performance issues every developer/architect must know — part 2 — Concurrency](https://dineshchandgr.medium.com/top-performance-issues-every-developer-architect-must-know-part-2-concurrency-a15bd0b2b3b6" \t "_blank)**

[Hello everyone. This article is a continuation of the part 1…](https://dineshchandgr.medium.com/top-performance-issues-every-developer-architect-must-know-part-2-concurrency-a15bd0b2b3b6" \t "_blank)

[dineshchandgr.medium.com](https://dineshchandgr.medium.com/top-performance-issues-every-developer-architect-must-know-part-2-concurrency-a15bd0b2b3b6" \t "_blank)

If you have not read it, please read that and come back here to have a continuation.

The performance issues were classified into different segments as follows

* Database problems
* [Concurrency problems](https://enlear.academy/how-to-avoid-a-deadlock-while-writing-concurrent-programs-java-example-988bb07db25f)
* **Memory problems**

I have covered the **Database problems in part 1, the Concurrency problems in part 2,**and let me cover the Memory problems in Java in this part 3 article.

**Memory problems**

As we all know in [C/C++](https://medium.com/javarevisited/top-10-courses-to-learn-c-for-beginners-best-and-free-4afc262a544e), programmers have to manage memory. It's pretty powerful but burdens us with a lot of responsibility. There was always the risk of memory leaks but Java was sophisticated in managing memory for developers and enabled us to build stable and secure applications. Every sophistication comes with some automation and in Java, we have the Garbage Collection mechanism, which is a burden if not used properly

There are 2 major issues we have to check in terms of memory. They are

**1.**[**Garbage Collection**](https://medium.com/javarevisited/7-best-courses-to-learn-jvm-garbage-collection-and-performance-tuning-for-experienced-java-331705180686)

**2. Memory Leaks**

**1. Garbage Collection**

To understand when and why garbage collection pauses the JVM, we need to understand a little bit about how garbage collection works in Java. Different JVM implementations and different JVM garbage collection strategies manage heaps differently. *Garbage Collection* would deal with finding and deleting the garbage from the memory. The algorithm tracks each and every object available in the JVM heap space and removes the unused ones.

Basically, *GC* works in two simple steps, known as Mark and Sweep:

* **Mark —**this is where the garbage collector identifies which pieces of memory are in use and which aren’t.
* **Sweep —**this step removes objects identified during the “mark” phase.

[[](https://www.java67.com/2019/08/best-books-to-learn-java-virtual-machine-in-depth.html)](https://www.java67.com/2019/08/best-books-to-learn-java-virtual-machine-in-depth.html)

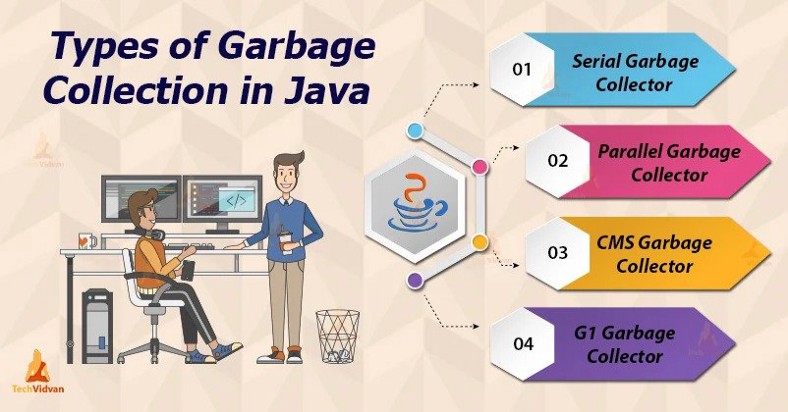
***Stop the World Garbage Collection***

A Major garbage collection, also known as a stop-the-world (STW) collection freezes all threads in the JVM, performs a mark-sweep collection across the entire heap, and then performs compaction. This is a very effective garbage collection, but it is both very slow and impactful to your users because no threads are able to run during this collection. So it “stops the JVM world” for however long it takes to run. And the amount of time will be directly proportional to the size of your heap. In a modest 2–3 gigabyte heap this might be 3–5 seconds, but if you are running a 30-gigabyte head this could be more on the order of 30 seconds.

There are strategies to minimize the impact of stop-the-world collections, such as the Concurrent Mark Sweep (CMS) garbage collection strategy, which maintains an additional thread that is constantly marking and sweeping objects, and the new G1 garbage collection strategy, which attempts to collect popular sections of the heap, but neither of these completely eliminates stop-the-world compactions.

**Types of GC alogorithms**

JVM has the following types of *GC* implementations:

[[](https://www.java67.com/2020/02/50-garbage-collection-interview-questions-answers-java.html)](https://www.java67.com/2020/02/50-garbage-collection-interview-questions-answers-java.html)

* **Serial Garbage Collector**

This 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 multithreaded applications, like server environments.

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

java -XX:+UseSerialGC -jar Application.java

* **Parallel Garbage Collector**

It’s the default *GC* of the *JVM until Java 8*and unlike *Serial Garbage Collector*, it **uses multiple threads for managing heap space,** but it also freezes other application threads while performing *GC*. This performs a Major GC

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

java -XX:+UseParallelGC -jar Application.java

* **CMS Garbage Collector**

**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.

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

To enable the *CMS Garbage Collector*, we can use the following flag:

java -XX:+UseParNewGC -jar Application.java

* **G1 Garbage Collector**

*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 has replaced the *CMS* collector since it’s more performance efficient and it is the default GC from Java 9 onwards.

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 empty. 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.

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

java -XX:+UseG1GC -jar Application.java

**Issues with GC**

1. The symptoms of stop-the-world garbage collection include periodic CPU spikes and you might observe that your application is performing well most of the time but once or twice an hour your SLAs set off alerts.

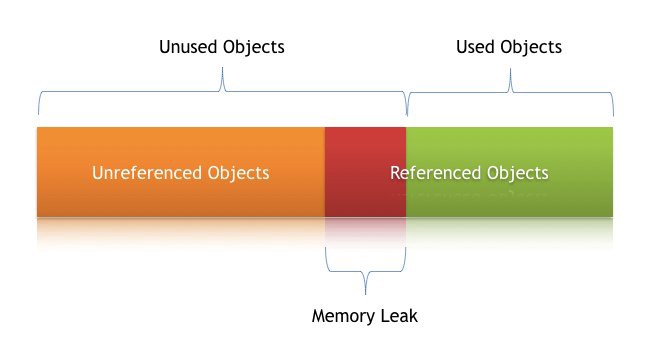
2. If you have a performance monitoring solution in place then you can also observe an increase in the frequency of major garbage collection occurrences as well as a pattern of increase in your old generation followed by a dip.

3. Remember that objects in the old generation can only be reclaimed by major garbage collections, so whenever you see the old generation shrink, it means that a major garbage collection occurred.

**Monitoring and Resolution**

* Enable verbose garbage collection logging during Java startup
* Use a performance monitoring tool that shows you major garbage collection occurrence counts and look for frequent increases
* Use a performance monitoring tool that shows you the behavior of your CPU and heap usage and look for CPU spikes and frequent increases and dips in heap usage
* Cycle your JVMs: if you are running in an elastic environment, such as in a cloud-like Amazon’s EC2, and regularly scale up and down throughout the day, you might consider cycling down older machines first. In this capacity, you can maintain a much larger heap and shut down the JVM before a major garbage collector ever has a chance to run.

**2. Memory Leaks**

[[](https://javarevisited.blogspot.com/2012/10/10-garbage-collection-interview-question-answer.html)](https://javarevisited.blogspot.com/2012/10/10-garbage-collection-interview-question-answer.html)

Memory leaks in Java are much more of a reference management issue than they are true memory leaks.

Java memory leaks occur when a Java application inadvertently maintains a reference to an object that it does not ever intend to use again. There is no definitive way for the garbage collector to assess the intentions of the developer who built the application, so if a reference to an object is maintained, the garbage collector will assume that someone intended to use it at some point in the future.

Unfortunately, this tends to occur in code that is frequently executed, causing the JVM to eventually exhaust its memory and throw the dreaded OutOfMemoryError, which usually means you have to restart your JVM.

**Symptoms**

The symptom of a Java memory leak is a gradual increase in memory usage until the heap eventually runs out of memory. Unfortunately, there is no definitive way of differentiating between simply running out of memory and a memory leak, but there are ways to infer what is happening, which are summarized below.

Depending on the severity of your memory leak and how quickly it takes down your JVM, the impact can be severe. When a JVM runs out of memory, it must be killed and restarted before it can service any additional requirements. From an operations perspective, you will find yourself switching between machines and restarting JVMs potentially several times a day

**Troubleshoot**

Identifying the root cause of a memory leak is challenging but fortunately, there are some behaviors of memory leaks. Memory leaks can only occur inside containers that support unbounded growth. Java natively supports unbounded growth through its Collection classes: it maintains lists, maps, and sets supported by linked lists, arrays, trees, hash tables, and more, which can all grow without bound.

Consider a common scenario that leads to a memory leak: on a per-request basis, the application adds an object to a collection but does not remove it from the collection when it’s done.

Because this occurs as a result of user interaction and the user continues to interact with your application, the end result is a memory leak. The good thing is that because of this behavior, intelligent analysis can be performed to detect this behavior and point to suspected memory leaks.

**How to Avoid Memory Leaks**

Avoiding memory leaks requires close scrutiny of code where memory leaks can occur, namely around Java collection classes. Additionally, memory leak-like symptoms can occur around session management: memory used by user sessions will eventually be reclaimed by the garbage collector, but if the heap exhausts its memory before the sessions expire, then session memory can cause **OutOfMemoryErrors**.

1. Share common memory leak scenarios, such as managing data in Java Collection classes, with your developers and ask them to review their code for memory leak

2. Employ a Java memory profiler to analyze code during development. Execute a load test against the database and tune your database to optimally support these queries.

3. Monitor the application in production to detect potential memory leaks (including capabilities for observing changes in collection classes)

4. Configure your JVM to capture a heap dump when an [OutOfMemoryError](https://javarevisited.blogspot.com/2019/04/top-5-courses-to-learn-jvm-internals.html" \t "_blank)occurs and analyze that heap dump using a tool like Eclipse Memory Analyzer

Hope you have enjoyed reading this article. Please follow me for more content