## **1. Introduction**

The JVM ships with various options for garbage collection to support a variety of deployment options. With this, we get flexibility in choosing [which garbage collector](https://www.baeldung.com/jvm-garbage-collectors) to use for our application.

By default, the JVM chooses the most appropriate garbage collector based on the class of the host computer. However, sometimes, our application experiences major GC-related bottlenecks requiring us to take more control over which algorithm is used. The question is, **“how does one settle on a GC algorithm?”**

In this article, we attempt to answer that question.

## **2. What Is a GC?**

Java being a [garbage-collected language](https://www.baeldung.com/java-memory-management-interview-questions), we are shielded from the burden of manually allocating and deallocating memory to applications. The whole chunk of memory allocated to a JVM process by the OS is called the *heap*. The JVM then breaks this heap into two groups called generations. This breakdown enables it to apply a variety of techniques for efficient memory management.

The ***young (Eden) generation* is where newly created objects are allocated**. It’s usually small (100-500MB) and also has two *survivor spaces*. **The *old generation* is where older or aged objects are stored** — these are typically long-lived objects. This space is much larger than the young generation.

The collector continuously **tracks the fullness of the young generation** and triggers minor collections during which live objects are moved to one of the survivor spaces and dead ones removed. If an object has survived a certain number of minor GCs, the collector moves it to the old generation. **When the old space is considered full, a major GC happens** and dead objects are removed from the old space.

During each of these GCs, there are **stop-the-world** phases during which nothing else happens — the application can’t service any requests. We call this *pause time*.

## **3. Variables to Consider**

Much as GC shields us from manual memory management, it achieves this at a cost. **We should aim to keep the GC runtime overhead as low as possible**. There are several variables that can help us decide which collector would best serve our application needs. We’ll go over them in the remainder of this section.

### **3.1. Heap Size**

This is the total amount of working memory allocated by the OS to the JVM. Theoretically, **the larger the memory, the more objects can be kept before collection, leading to longer GC times**. The minimum and maximum heap sizes can be set using *-Xms=<n>* and *-Xmx=<m>* command-line options.

### **3.2. Application Data Set Size**

This is the total size of objects an application needs to keep in memory to work effectively. Since all new objects are loaded in the young generation space, this will definitely affect the maximum heap size and, hence, the GC time.

### **3.3. Number of CPUs**

This is the number of cores the machine has available. This variable directly affects which algorithm we choose. Some are only efficient when there are multiple cores available, and the reverse is true for other algorithms.

### **3.4. Pause Time**

The pause time is the duration during which the garbage collector stops the application to reclaim memory. This variable directly affects latency, so the goal is to limit the longest of these pauses.

### **3.5. Throughput**

By this, we mean the time processes spend actually doing application work. **The higher the application time vs. overhead time spent in doing GC work, the higher the throughput of the application**.

### **3.6. Memory Footprint**

This is the working memory used by a GC process. When a setup has limited memory or many processes, this variable may dictate scalability.

### **3.7. Promptness**

This is the time between when an object becomes dead and when the memory it occupies is reclaimed. It’s related to the heap size. In theory, the larger the heap size, the lower the promptness as it will take longer to trigger collection.

### **3.8. Java Version**

As new Java versions emerge, there are usually changes in the supported GC algorithms and also the default collector. We recommend starting off with the default collector as well as its default arguments. Tweaking each argument has varying effects depending on the chosen collector.

### **3.9. Latency**

This is the responsiveness of an application. GC pauses affect this variable directly.

## **4. Garbage Collectors**

Besides serial GC, all the other collectors are most effective when there’s more than one core available:

### **4.1. Serial *GC***

The serial collector uses a single thread to perform all the garbage collection work. It’s selected by default on certain small hardware and operating system configurations, or it can be explicitly enabled with the option *-XX:+UseSerialGC*.

Pros:

* Without inter-thread communication overhead, it’s relatively efficient.
* It’s suitable for client-class machines and embedded systems.
* It’s suitable for applications with small datasets.
* Even on multiprocessor hardware, if data sets are small (up to 100 MB), it can still be the most efficient.

Cons:

* It’s not efficient for applications with large datasets.
* It can’t take advantage of multiprocessor hardware.

### **4.2. Parallel/Throughput *GC***

This collector **uses multiple threads to speed up garbage collection**. In Java version 8 and earlier, it’s the default for server-class machines. We can override this default by using the *-XX:+UseParallelGC* option.

Pros:

* It can take advantage of multiprocessor hardware.
* It’s more efficient for larger data sets than serial GC.
* It provides high overall throughput.
* It attempts to minimize the memory footprint.

Cons:

* Applications incur **long pause times during stop-the-world operations**.
* It doesn’t scale well with heap size.

It’s best if we want more throughput and don’t care about pause time, as is the case with non-interactive apps like batch tasks, offline jobs, and web servers.

### **4.3. Concurrent Mark Sweep (CMS) GC**

We consider CMS a mostly concurrent collector. This means **it performs some expensive work concurrently with the application**. It’s designed for low latency by eliminating the long pause associated with the full GC of parallel and serial collectors.

We can use the option *-XX:+UseConcMarkSweepGC* to enable the CMS collector. The core Java team deprecated it as of Java 9 and completely removed it in Java 14.

Pros:

* It’s great for low latency applications as it minimizes pause time.
* It scales relatively well with heap size.
* It can take advantage of multiprocessor machines.

Cons:

* It’s deprecated as of Java 9 and removed in Java 14.
* It becomes relatively inefficient when data sets reach gigantic sizes or when collecting humongous heaps.
* It requires the application to share resources with GC during concurrent phases.
* There may be throughput issues as there’s more time spent overall in GC operations.
* Overall, it uses more CPU time due to its mostly concurrent nature.

### **4.4. G1 (Garbage-First) GC**

G1 uses multiple background GC threads to scan and clear the heap just like CMS. Actually, the core Java team designed G1 as an improvement over CMS, patching some of its weaknesses with additional strategies.

In addition to the incremental and concurrent collection, **it tracks previous application behavior and GC pauses to achieve predictability**. It then focuses on reclaiming space in the most efficient areas first — those mostly filled with garbage. We call it *Garbage-First* for this reason.

freestar

Since Java 9, G1 is the default collector for server-class machines. We can explicitly enable it by providing *-XX:+UseG1GC* on the command line.

Pros:

* It’s very efficient with gigantic datasets.
* It takes full advantage of multiprocessor machines.
* It’s the most efficient in achieving pause time goals.

Cons:

* It’s not the best when there are strict throughput goals.
* It requires the application to share resources with GC during concurrent collections.

G1 works best for applications with very strict pause-time goals and a modest overall throughput, such as real-time applications like trading platforms or interactive graphics programs.

### **4.5. Z Garbage Collector (ZGC)**

ZGC is a scalable low latency garbage collector. It manages to keep low pause times on even multi-terabyte heaps. It uses techniques including reference coloring, relocation, load barriers and remapping. It is a good fit for server applications, where large heaps are common and fast application response times are required.

It was introduced in Java 11 as an experimental GC implementation. We can explicitly enable it by providing *-XX:+UnlockExperimentalVMOptions -XX:+UseZGC* on the command line. For more detailed descriptions, please visit [our article on Z Garbage Collector.](https://www.baeldung.com/jvm-zgc-garbage-collector)

## **5. Conclusion**

For many applications, the choice of the collector is never an issue, as the **JVM default usually suffices**. That means the application can perform well in the presence of garbage collection with pauses of acceptable frequency and duration. However, this isn’t the case for a large class of applications, especially those with humongous datasets, many threads, and high transaction rates.

In this article, we’ve explored the garbage collectors supported by the JVM. We’ve also looked at key variables that can help us choose the right collector for the needs of our application.