Liberica JDK
Guide to JVM memory configuration options

Liberica JDK
Revision 2.0
July 2025

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

THIS INFORMATION MAY CHANGE WITHOUT NOTICE. TO THE EXTENT PERMITTED BY APPLICABLE LAW, BELLSOFT PROVIDES THIS DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL BELLSOFT BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF BELLSOFT IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Alpaquita, Liberica and BellSoft are trademarks or registered trademarks of BellSoft Corporation. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates. Other trademarks are the property of their respective owners and are used only for identification purposes.

Contents

1. Introduction

Reducing the memory footprint and increasing the performance of the application requires meticulous optimizations with due consideration of all variables. This document contains an overview of the Garbage Collection options in JDK and the most important JVM flags related to memory management.

2. Heap and RAM size options

Heap size options

The heap is where your object data is stored. This area is then managed by the garbage collector selected at startup. Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation.

JVM parameter	Description
-Xms	Sets the initial heap size
-Xmx	Sets the maximum heap size
-XX:MinHeapFreeRatio	Sets the minimum percentage of free space after garbage collection
-XX:MaxHeapFreeRatio	Sets the maximum percentage of free space after garbage collection
-XX:MaxDirectMemorySize	Sets the limit for the memory allocated to direct byte buffers

In some cases, setting the maximum and minimum Java heap size is enough to optimize JVM memory footprint. The optimal heap size depends on your application, so you should experiment with the values before settling on a final number.

Setting min. and max. proportion of heap free after GC helps to avoid unnecessary expansion and shrinking of free space and release the unused memory without affecting the performance significantly.

For instance, if you set `-XX:MinHeapFreeRatio=40` and `-XX:MaxHeapFreeRatio=70`, then the generation expands if the free space percentage goes below 40% and contracts if the free space exceeds 70%.

Direct byte buffers are used by the JVM to perform native I/O operations. As opposed to non-direct byte buffers stored in the heap, direct ones reside outside the heap and therefore are not affected by heap size parameters or garbage collection. By default, the JVM chooses the size of the direct size buffers

automatically based on the available memory, so setting the `-XX:MaxDirectMemorySize` helps to prevent excessive resource consumption.

Options for limiting total RAM consumption

JVM parameter	Description
-XX:MaxRAM	Sets the max. amount of total memory used by the JVM
-XX:MaxRAMFraction	Sets the RAM limit for JVM in fractions
-XX:MaxRAMPercentage	Sets the RAM limit for JVM per cent

The JVM flags adjusting the heap size do not affect the total memory consumption by the JVM. To limit the total RAM consumption, use MaxRam flags. The heap size will be adjusted accordingly. For instance, if you have 1 GB of memory, setting `-XX:MaxRAMPercentage=50` (or `-XX:MaxRAMFraction=2`) will make the JVM allocate approx. 500 MB to heap.

These arguments are especially useful in the case of containerized applications, where they help to adjust the heap size based on the available container memory.

3. Garbage Collection mechanism

Overview

Garbage collection (GC) is an important part of JVM that has effect on the overall performance of an application. Garbage collection is a process of freeing up memory by deleting unused objects from the heap. An object is considered eligible for GC when it becomes unreachable, meaning there are no references to it. GC in JDK is automatic. There are ways of invoking the GC manually, but it is not the best practice.

How does Garbage Collection work

When the application starts, the heap is almost empty. While the threads are working, the heap is filling up, until it reaches the state where the new objects cannot be created any more. It triggers garbage collection.

All garbage collectors pause the application threads for some time (Stop-theWorld), scan the application thread stacks, and mark all objects that are directly accessible from those stacks. Then they mark all reachable objects. When all reachable objects are marked, the rest is treated as garbage and removed. Concurrent collectors resume the stopped application threads as soon as possible and do part of the GC work along with application.

There is always a tradeoff between application pause time and GC throughput. Doing all GC tasks when application threads are paused gives a better throughput, but longer stop-the-world time.

Despite that all GCs perform the steps above, the implementation details are different and each implementation has its strength and weakness.

Advantages of automatic Garbage Collection

- No need for manual memory allocation/deallocation, which saves the developer’s time and minimizes bugs related to human error;
- Efficient memory usage as the heap gets cleaned as soon as it fills up;

- Reduced risk of memory leaks — most of these cases are successfully handled by the collector.

Disadvantages of automatic Garbage Collection

- Lack of control over memory management leaves little space for improving the resource usage;
- While some applications may benefit from automatic GC, larger enterprise apps may require manual GC tuning to achieve maximal performance;
- Automatic garbage collection still allows incorrectly designed applications to create memory leaks, that are hard to debug.

Manual invocation of GC

There is a way of invoking the GC manually by calling `System.gc()`.

> **Important:**
>
> Explicitly calling `System.gc()` in your code is an unrecommended practice.

`System.gc()` requests to trigger Full GC as soon as possible, which means the JVM is notified about the garbage collections request, but the JVM decides by itself when to perform garbage collection depending on lots of parameters and may induce Full GC when the whole heap gets cleaned, which might have disastrous impact on performance. Use the `-XX:+DisableExplicitGC` flag to prohibit the explicit execution of GC. It also does not allow you to trigger GC through code with calling `System.gc()`. See GC management for GC-specific flags and options.

Types of Garbage Collectors in JDK

JDK provides numerous opportunities for GC tuning with various GC implementations with their own strengths. The choice of a garbage collector depends on the defined goals.

For instance, Serial GC can be suitable for memory and CPU constraint devices, but there will be long pauses in application work especially if a significant amount of memory is involved. Parallel GC gives you the balance between pause and throughput. After selecting the collector, you can start the tuning process based on GC capabilities. Garbage collection adjustment is a balancing act: large heap means longer pauses, short pauses lead to a more frequent GC activation and so on.

Below you will find the summary of key garbage collectors available in JVM with distinguishing features, usage suggestions, and references to JDK Enhancement Proposals, known as JEPs.

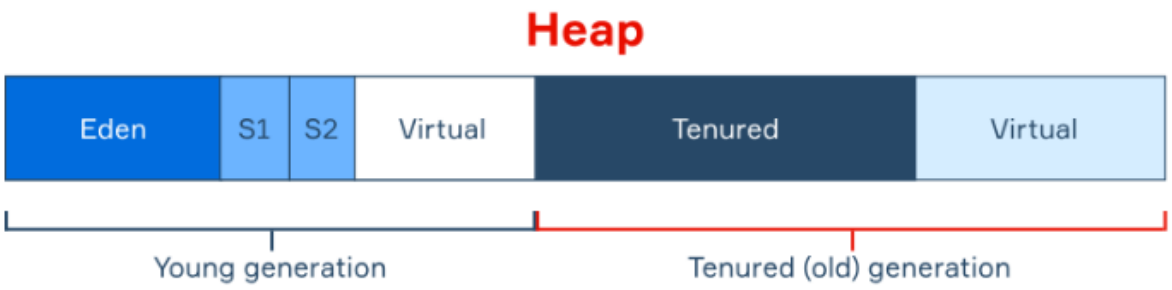
Serial Garbage Collector

Serial GC is the oldest and simplest GC implementation in JDK. It is utilized in single-threaded environments as it freezes all threads while it performs the collection and works in one thread itself.

Serial GC is suitable for client-side applications without low pause requirements. Note that this GC is used automatically if the RAM limit is set to less than 1792 MB or there are less than 2 CPUs. Otherwise, to enable Serial GC, use the following command:

```
java -XX:+UseSerialGC -jar yourApp.java
```

GC Workings



The heap is split into two areas, young gen and old gen. The young gen also contains a survivor space. The algorithm is based on the "die young" idea - most objects won't survive more than one GC. When an object of regular size is created, it is placed to Eden. When Eden becomes full, minor collection is started. The VM stops application threads, scans threads and marks all objects reachable from the thread root and from each other. Then live objects are copied to one of the survival spaces. Objects that survive more than MaxTenuringThreshold GC cycles are promoted to Tenured space (old gen). If the object is too large to fit into Eden space it will be allocated directly to the Tenured space. When the tenured space fills up, the FullGC happens. Application threads stop for significant time until all garbage is evacuated and the entire heap is compacted.

Usage

The Serial GC can be suitable for memory and CPU constraint devices. On today's hardware, the Serial GC can efficiently manage a lot of applications with a few hundred MBs of JDK heap, with relatively short pauses (around a couple of seconds for full garbage collections).

Another popular use for the Serial GC is in environments where a high number of JVMs run on the

same machine (in some cases, more JVMs than available processors!). In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs, even if the garbage collection might last longer.

Embedded hardware and container environment with minimal memory and few cores are other areas where the Serial GC can be quite useful.

JEPs

- JEP 366: Deprecate the combination of the Parallel Scavenge and Serial Old garbage collection algorithms.

Parallel Garbage Collector

ParallelGC is the evolution of SerialGC destined for a multiprocessor environment. It shares the same generational principles but with two significant additions: ParallelGC can use multiple threads to collect both young (Eden) and old (Tenured) space and have the ergonomics algorithms that automatically tune internal GC parameters in order to achieve application goals: Maximum pause time (can be set using `-XX:MaxGCPauseMillis=<N>`); Throughput - gc time to app time ratio (can be set using `-XX:GCTimeRatio=<N>`); Footprint - fit the heap specified by `-Xmx`.

By default, the number of threads in the collection is calculated based on the processor number. The number of garbage collector threads can be controlled with the `-XX:ParallelGCThreads` command-line option. The following command enables the Parallel GC.

```
java -XX:+UseParallelGC -jar yourApp.java
```

GC Workings

ParallelGC follows the same principles as SerialGC, dividing the heap into young (Eden) and old (Tenured) generations. The garbage collection process occurs during a Stop-the-World (STW) pause, during which all application threads are paused. The GC may trigger either minor collection, where only young gen is collected or full collection where GC collects the entire heap.

The GC scans thread roots, marks objects directly reachable from the thread stacks, and then marks all reachable objects.

While the marking of live objects is still performed by a single thread, multiple threads handle the promotion of objects from the young generation to the old generation, enhancing efficiency. The old generation is divided into multiple sections - promotion buffers, where each GC thread copies objects into its dedicated buffer. This structure enables multiple threads to operate concurrently

without locks or interference.

During FullGC, the old generation scanned and collected as well as the young one. Although the old generation collection itself is handled by a single thread, the compaction process utilizes multiple threads to optimize performance. However, the use of multiple threads for compaction can be disabled using the `-XX:-UseParallelOldGC` flag.

Usage

The Parallel GC can use multiple CPUs to increase throughput. This collector should be used when a lot of work needs to be done and long pauses are acceptable. For example, batch processing like printing reports or bills or performing a large number of database queries.

Concurrent Mark Sweep Garbage Collector

> **Important:**
>
> CMS Garbage Collector was deprecated since JDK 9 in favor of a more advanced G1 Garbage Collector according to JEP 291 and removed in later versions as specified in JEP 363. CMS is not recommended for production applications.

Similar to ParallelGC, CMS follows a two-generation memory layout and employs the same algorithm for minor (young gen) collections. However, its major (old gen) collection process differs significantly. In CMS, minor and major collections are always separate, meaning the young generation is not collected during a major collection cycle. Instead, CMS periodically scans the old generation in the background. When fragmentation in the old generation exceeds a certain threshold, a major collection is triggered.

During a major collection, CMS briefly pauses all application threads to scan thread roots and mark objects directly reachable from the thread stacks. After this initial marking phase, application threads resume while CMS continues marking all reachable objects in the background. Since new objects may be allocated during this concurrent marking phase, CMS performs an additional short pause at the end to rescan roots and ensure accuracy. Once the application resumes, the garbage is evacuated in the background, minimizing pauses but increasing CPU load.

To enable CMS GC, run the following command:

```
java -XX:+UseConcMarkSweepGC -jar yourApp.java
```

This collector was deprecated since JDK 9, so you get the following warning when trying to run it with

JDK 11:

```
java -XX:+UseConcMarkSweepGC --version
OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in
version 9.0 and will likely be removed in a future release.
```

It has already been removed from the code base in newer versions, and the following message appears in JDK 17:

```
Unrecognized VM option 'UseConcMarkSweepGC'
```

JEPs

- JEP 214: Remove GC Combinations Deprecated in JDK 8.
- JEP 291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector.
- JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector.

G1 Garbage Collector

G1 GC is the default GC implementation designed to replace CMS GC with low latency in mind. It is fit for any application, but shows the best performance with server-class applications running in a multiprocessor environment with a large heap. G1 GC utilizes the Garbage-First approach by dividing the heap into multiple regions and performing the global marking phase to determine the liveliness of objects.After learning which heap regions are mostly filled with garbage, it first collects garbage in those regions to free up a lot of memory.

G1 GC copies objects from one or several memory regions into a single region, which enables it to compact memory. The compaction is performed in parallel on multiprocessor machines, thus reducing pause times and increasing throughput. In addition, the developers can adjust the maximum pause time and pause time intervals.

To enable G1 GC, run the following command:

```
java -XX:+UseG1GC -jar yourApp.java
```

GC Workings

G1 GC divides the heap into the number of equivalent regions, each of which can be either young gen (containing Eden and survivor) or old gen, depending on the application needs. As for all previous collectors, space reclamation efforts are concentrated on young gen with occasional collection of the old-gen. G1 reclaims space mostly by using evacuation: live objects found within

selected memory areas to collect are copied into new memory areas, compacting them in the process. After an evacuation has been completed, the space previously occupied by live objects is reused for allocation by the application.

G1 Garbage collection consist of several phases:

1. **Young-only phase**: This phase starts with a few Normal young collections that promote objects into the old generation. The transition between the young-only phase and the space-reclamation phase starts when the old generation occupancy reaches a certain threshold, the Initiating Heap Occupancy threshold. At this time, G1 schedules a Concurrent Start young collection instead of a Normal young collection.

2. **Concurrent Start**: This type of collection starts the marking process in addition to performing a Normal young collection. Concurrent marking determines all currently reachable (live) objects in the old generation regions to be kept for the following space-reclamation phase. While collection marking is still in progress, Normal young collections may occur. Marking finishes with two special stop-the-world pauses: Remark and Cleanup.

3. **Remark**: This pause finalizes the marking itself, performs global reference processing and class unloading, reclaims completely empty regions and cleans up internal data structures. Between Remark and Cleanup, G1 calculates information to later be able to reclaim free space in selected old generation regions concurrently, which will be finalized in the Cleanup pause.

4. **Cleanup**: This pause determines whether a space-reclamation phase will actually follow. If a space-reclamation phase follows, the young-only phase completes with a single Prepare Mixed young collection.

5. **Space-reclamation phase**: This phase consists of multiple Mixed collections that in addition to young generation regions, also evacuate live objects of sets of old generation regions. The space-reclamation phase ends when G1 determines that evacuating more old generation regions wouldn't yield enough free space worth the effort.

After space-reclamation, the collection cycle restarts with another young-only phase. As backup, if the application runs out of memory while gathering liveness information, G1 performs an in-place stop-the-world full heap compaction (Full GC) like other collectors.

G1 performs garbage collections and space reclamation in stop-the-world pauses. Live objects are typically copied from source regions to one or more destination regions in the heap, and existing references to these moved objects are adjusted.

Usage

The G1 GC is a server-class garbage collector, targeted at multiprocessor machines with large memory. It meets garbage collection (GC) pause time goals with a high probability, while

achieving high throughput. The G1 collector:

- Can operate concurrently with applications threads.
- Compact free space without lengthy GC induced pause times.
- Provides more predictable GC pause durations.
- Has less throughput than ParallelGC.

G1 provides a solution for users running applications that benefit from low GC pause despite decreased throughput.

Applications running today with older garbage collectors would benefit from switching to G1 if the application has one or more of the following traits.

- Full GC durations are too long or too frequent.
- The rate of object allocation rate or promotion varies significantly.
- Undesired long garbage collection or compaction pauses.

JEPs

- JEP 156: G1 GC: Reduce need for full GCs.
- JEP 192: Reduce the Java heap live-data set by enhancing the G1 Garbage Collector.
- JEP 248: Make G1 the Default Garbage Collector.
- JEP 307: Parallel Full GC for G1 improves the latency of G1 GC during the full collection.
- JEP 344: Make G1 mixed collections abortable if they might exceed the pause target.
- JEP 345: Improve G1 performance on large machines by implementing NUMA-aware memory allocation.
- JEP 346: Enhance the G1 garbage collector to automatically return Java heap memory to the operating system when idle.
- JEP 423: Reduce latency by implementing region pinning in G1.
- JEP 475: Simplify the implementation of the G1 garbage collector's barriers, which record information about application memory accesses, by shifting their expansion from early in the C2 JIT's compilation pipeline to later.

Z Garbage Collector

ZGC is a concurrent, single-generation (since JDK 21), region-based, NUMA-aware, compacting collector. Stop-the-world phases are limited to root scanning, so GC pause times do not increase with the size of the heap or the live set. ZGC uses a few high bits of each object to keep its reachability status and access barriers to update these bits.

Z Garbage Collector (ZGC) was introduced in JDK 11 as an experimental feature and obtained production status starting with JDK 15.

It is a scalable low-latency collector that performs the expensive work concurrently and does not stop the application threads for more than 10 ms. The most important parameter is the max. heap size (`-Xmx`): it should be able to accommodate the live-set of the app and provide enough room for allocations. To use ZGC, run the following command:

```
java -XX:+UseZGC -jar yourApp.java
```

For versions up to JDK 15 ZGC is experimental and the command is slightly different:

```
java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -jar yourApp.java
```

GC Workings

A core principle in ZGC is the use of load barriers in combination with colored object pointers (i.e., colored oops). This is what enables ZGC to do concurrent operations, such as object relocation, while Java application threads are running. From a Java thread's perspective, the act of loading a reference field in a Java object is subject to a load barrier. In addition to an object address, a colored object pointer contains information used by the load barrier to determine if some action needs to be taken before allowing a Java thread to use the pointer. For example, the object might have been relocated, in which case when the load barrier triggers, the object pointer will be updated as required.

Usage

ZGC has been designed to be adaptive and to require minimal manual configuration. During the execution of the Java program, ZGC dynamically adapts to the workload by resizing generations, scaling the number of GC threads, and adjusting tenuring thresholds. The main tuning knob is to increase the maximum heap size.

ZGC is suitable for applications which require low latency. Pause times are independent of the heap size that is being used. ZGC works well with heap sizes from a few hundred megabytes to 16TB.

JEPs

- JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental).
- JEP 351: Enhance ZGC to return unused heap memory to the operating system.
- JEP 364: Port the ZGC garbage collector to macOS.
- JEP 365: Port the ZGC garbage collector to Windows.
- JEP 376: Move ZGC thread-stack processing from safepoints to a concurrent phase.
- JEP 377: Change the Z Garbage Collector from an experimental feature into a product feature.
- JEP 439: Generational ZGC reduces the GC CPU overhead by making Z GC maintain young and old objects separately, thus collecting young objects more frequently.
- JEP 474: Switch the default mode of the Z Garbage Collector (ZGC) to the generational mode. Deprecate the non-generational mode, with the intent to remove it in a future release.
- JEP 490: Remove the non-generational mode of the Z Garbage Collector (ZGC).

Shenandoah Garbage Collector

This is a region-based low-pause parallel and concurrent GC algorithm targeted at large heap applications. It performs garbage collection concurrently with the running Java application thus reducing pause times which are independent of the application's live data size. The GC provides minimal possible pauses on huge heaps at the cost of extra memory and cpu consumption.

Shenandoah works with all LTS releases and a current JDK release and supports a wide range of platforms. The OpenJDK community continuously backports improvements and bug fixes to previous supported JDK versions.

To enable Shenandoah GC, run the following command:

```
java -XX:+UseShenandoahGC -jar yourApp.java
```

GC Workings

Shenandoah GC uses some extra pointers to mark live objects and perform compacting without stopping the application. The GC relies on access barriers to maintain object state, i.e. part of the GC work is done by the application thread along with application tasks.

The heap is broken up into equal sized regions. A region may contain newly allocated objects,

long-lived objects, or a mix of both. Any subset of the regions may be chosen to be collected during a GC cycle.

GC phases:

1. Initial Marking - Stops the world, scans the heap.
2. Concurrent Marking - Traces the heap marking the live objects, updates any references to regions evacuated in the previous GC cycle.
3. Final Marking: Stops the world, re-scans the root set, copies and updates roots to point to region copies. Initiates concurrent compaction. Frees any fully evacuated regions from previous compaction.
4. Concurrent Compaction: Evacuates live objects from targeted regions.

Each GC cycle consists of two stop the world phases and two concurrent phases.

Usage

Shenandoah's performance has improved considerably in the last ten-plus years of development, and it is considered a mature product supported in production environments.

You can use non-generational Shenandoah in huge workloads with neither random nor too much generational overload. For example, use it for applications with constant allocation of objects.

Shenandoah GC usage may be beneficial in some container applications, but we recommend testing it first in your environment, because it depends on the use case.

JEPs

- JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)
- JEP 379: Change the Shenandoah garbage collector from an experimental feature into a product feature.

Epsilon Garbage Collector

Epsilon GC is the most peculiar of all Garbage Collectors because it does not collect any garbage. Its primary purpose is to allocate memory. Once the available heap is exhausted and the application tries to allocate more memory than allowed (that is, set by `-Xmx`), the JVM shuts down with an `OutOfMemoryError`.

This no-ops GC was introduced in JDK 11 as an experimental feature, but it was decided to keep it experimental to avoid accidental Epsilon GC enabling in production. So, to use Epsilon GC, you need to explicitly enable experimental features first:

```
java -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -jar yourApp.java
```

Usage

Despite the fact that Epsilon GC doesn't collect any garbage, there are several use cases where it can be useful:

- Performance testing with Epsilon GC may reveal how fast the application runs without garbage collection and whether there are performance bottlenecks, which can be clearly seen without GC-induced performance artifacts.
- Applications that create all necessary objects at start and don't produce any garbage as well as short-lived applications that don't have time to spend all available resources may run faster without garbage collection.

Otherwise, it is not recommended to use Epsilon GC to avoid unexpected application behavior and crashes.

JEPs

- JEP 318: Epsilon: A No-Op Garbage Collector (Experimental).

Selecting GC

Garbage Collection and JDK versions

The choice of a garbage collector relies, among other things, on the JDK version you use.

> **Note:**
>
> For the full list of GCs supported in different environments, see Appendix: Standard JDK and JDK Performance Edition

First of all, a certain collector may not be available with your version:

- CMS GC was removed from JDK 14, so it is absent from newer JDK releases;
- ZGC appeared in JDK 11 as an experimental feature and it is now available in JDK 8 and all later versions;
- Shenandoah GC was introduced in JDK 12 as an experimental feature and became ready for production-use in JDK 15, so it is absent from older JDK versions;
- Epsilon GC was introduced in JDK 11 and is not available in previous versions.

Secondly, the performance of a chosen GC implementation also depends on the JDK version because there have been many enhancements introduced to Garbage Collection up until now, and the improvement process is ongoing. As of January 2025 Java Bug System contained more than 2,000 resolved and integrated fixes and enhancements to G1 GC alone.

Therefore, upgrading the JDK version is key to using the full potential of JDK garbage collectors. But if your enterprise workloads are based on JDK 8 or 11 and migration is off the table for now, you can use Liberica JDK Performance Edition that couples JDK 8 or 11 and JVM 17 or JVM 21. So technically, you stay on JDK 8 or 11, but enjoy the performance of version 17 and 21, including new and improved GC implementation. See Liberica JDK Performance Edition for more information.

Parameters to select GC

The following table lists parameters that enable a certain GC.

JVM parameter	Description
-XX:+UseSerialGC	Enables Serial Garbage Collector
-XX:+UseParallelGC	Enables Parallel Garbage Collector
-XX:+UseConcMarkSweepGC	Enables Concurrent Mark Sweep Garbage Collector (available up to JDK 8 only)
-XX:+UseG1GC	Enables G1 Garbage Collector

JVM parameter	Description
-XX:+UseZGC (since JDK 15)	Enables Z Garbage Collector (available since JDK 11)
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC (since JDK 11 up to 15)	
-XX:+UseShenandoahGC	Enables Shenandoah Garbage Collector (absent in Oracle JDK, available in major OpenJDK distributions)

The default garbage collection settings are enough for many applications. If you would like to enhance some KPIs, try switching to another collector, which default settings are more beneficial to your app, without delving into the intricacies of GC tuning. For example, if your system is limited to a single CPU, choose the Serial GC. If more than one CPU is active and sufficient memory is allocated, use the G1 GC. JDK can enable G1 GC automatically if resources reach certain limits.

The following table summarizes available GCs wih their main characteristics and possible use cases.

Characteristic	Serial GC	Parallel GC	CMS GC	G1 GC	Z GC	Shenandoah GC	Epsilon GC
Heap size	Small	Medium to Large	Medium to Large	Medium to Large	Very large	Very large	N/A
Latency	High	Moderate	Moderate	Moderate	Low	Low	N/A
Throughput	Low	High	Moderate	High	High	High	N/A

Characteristic	Serial GC	Parallel GC	CMS GC	G1 GC	Z GC	Shenandoah GC	Epsilon GC
Use cases	Client and Embedded applications in the single processor environment	Client, Embedded, and Server applications in the multiprocessor environment with focus on throughput	Server application in the multiprocessor environment that can afford sharing the resources with GC	Server-side applications in the multiprocessor environment with a large heap	Latency-sensitive applications, Applications with a very large heap (terabytes)	Latency-sensitive applications, Applications with a very large heap (terabytes)	Performance testing, Short-lived applications, Apps not producing garbage

4. GC management

Each GC comes with numerous settings that enable the developers to adjust latency, throughput, or memory.

Common parameters for all GCs

- `-XX:+AlwaysPreTouch` – By default, the OS allocates physical pages to the JVM on demand as they are accessed, which can cause latency spikes when pages are first touched. Enabling this option forces the JVM to pre-touch (write to) every page in the heap at startup, ensuring all memory is allocated upfront. While this may increase startup time, it is recommended, especially when the JVM is the primary or sole process running on the OS.
- `-XX:+UseStringDeduplication` – Scans the heap for duplicate strings and eliminates redundancy, reducing memory usage at the expense of additional CPU overhead. This feature is not supported on certain platforms, including ARM32 and PPC32.
- `-XX:+DisableExplicitGC` – Prevents the JVM from executing System.gc() calls, ensuring that garbage collection is managed entirely by the JVM's built-in logic.
- `-XX:ParallelGCThreads` (ParGC/G1GC Only) - This option specifies the number of threads the JVM should use for parallel garbage collection.
- `-XX:ConcGCThreads` (G1GC/Shenandoah GC/ZGC Only) – This option defines the number of threads to be used for concurrent garbage collection tasks, allowing GC work to occur alongside application threads. This can help reduce GC pause times, though it may impact application thread performance due to the additional GC threads running in parallel.

Key GC specific parameters

SerialGC/ParallelGC

- `-XX:NewRatio` - Controls the size ratio between the young generation (where new objects are allocated) and the old generation (where long-lived objects are promoted) in the heap. This is an important parameter that should be adjusted based on the characteristics of your application.

Increase it if the application creates many small, short-lived objects.

- `-XX:SurvivorRatio` - Controls the ratio between the sizes of the Survivor spaces (S0 and S1) in the young generation. The survivor spaces hold objects that survive garbage collection in the young generation before being promoted to the old generation.
- `-XX:MaxTenuringThreshold` - Sets the maximum number of times an object can survive in the young generation before being promoted to the old generation. The threshold defines how many garbage collection cycles an object can remain in the survivor spaces before being moved to the old generation.
- `-XX:+UseAdaptiveSizePolicy` - Allows the JVM to automatically adjust the sizes of memory regions (such as the young generation, old generation, and survivor spaces) based on the application's runtime behavior. However, this dynamic resizing incurs additional overhead. If your application's behavior is stable over time, it's often better to manually set the necessary parameters and disable this option, especially on small boards.

G1 GC

- `-XX:InitiatingHeapOccupancyPercent` - Sets the threshold (as a percentage of the total heap size) for when the G1 garbage collector should start a concurrent mark cycle. The JVM will trigger this cycle once the heap occupancy exceeds the specified percentage.
- `-XX:G1NewSizePercent` – Sets the percentage of the total heap size to be allocated to the young generation when using the G1 garbage collector. It controls how much of the heap is initially reserved for new objects, which are allocated in the young generation.
- `-XX:G1HeapWastePercent` – Specifies the maximum allowable percentage of the G1 garbage collector's heap that can be wasted (i.e., unused or fragmented) before G1 decides to trigger a full GC to reclaim space.
- `-XX:MaxGCPauseMillis` - Sets a target for the maximum pause time (in milliseconds) that the JVM should aim for during garbage collection. The JVM will attempt to adjust garbage collection behavior to meet this pause time goal, though it may not always be achievable, depending on the heap size, system resources, and workload.
- `-XX:G1HeapRegionSize` - Defines the size of each heap region managed by the G1 garbage collector. The heap is divided into multiple regions, and G1 handles these regions independently for more efficient garbage collection. By default, the JVM selects the region size based on the total heap size. For applications with many threads and a large heap, having more regions may be beneficial. Conversely, on systems with limited cores and applications with fewer threads, reducing the number of regions might improve performance.

ZGC

ZGC does not have any tunable parameters.

Shenandoah GC

- `-XX:ShenandoahGCHeuristics` – Selects the garbage collection strategy (heuristics) for the Shenandoah GC, determining how it balances CPU usage, pause times, and memory reclamation. Different heuristics optimize for various workloads:
  - adaptive (default) – Adjusts dynamically based on runtime behavior.
  - aggressive – Triggers GC more frequently to keep pause times low but increases CPU usage.
  - static – Uses fixed GC thresholds without adaptive tuning.
  - compact – Focuses on defragmentation of the heap to improve memory efficiency.
- `-XX:+ShenandoahCompact` – Enables heap compaction in the Shenandoah garbage collector, helping to reduce memory fragmentation by relocating objects and freeing up contiguous space.

Example setup

The following examples provide settings for the most commonly used GCs divided into small or large-scale applications.

Serial GC small board

```
-XX:+UseSerialGC
-Xmx=512m
-Xms=512m
-XX:NewRatio=2
-XX:SurvivorRatio=8
-XX:MaxTenuringThreshold=10
-XX:+AlwaysPreTouch
-XX:+DisableExplicitGC
```

G1 GC Small board

```
-XX:+UseG1GC
-Xmx=512m
-Xms=512m
-XX:InitiatingHeapOccupancyPercent=30
-XX:MaxGCPauseMillis=100
-XX:G1HeapRegionSize=64m
-XX:ParallelGCThreads=1
-XX:ConcGCThreads=1
-XX:+DisableExplicitGC
-XX:+AlwaysPreTouch
-XX:+UseStringDeduplication
```

Parallel GC server

```
-Xms384g
-Xmx384g
-XX:+UseParallelGC
-XX:ParallelGCThreads=18
-XX:NewRatio=8
-XX:SurvivorRatio=130
-XX:MaxTenuringThreshold=15
-XX:-UseAdaptiveSizePolicy
-XX:+UseStringDeduplication
```

G1 GC server

```
-Xms384g
 -Xmx384g
-XX:+UseG1GC
-XX:MaxGCPauseMillis=200
-XX:G1HeapRegionSize=32m
-XX:InitiatingHeapOccupancyPercent=40
-XX:G1NewSizePercent=20
-XX:SurvivorRatio=6
-XX:ParallelGCThreads=18
-XX:ConcGCThreads=8
-XX:G1HeapWastePercent=5
```

```
-XX:+UseStringDeduplication
```

5. Logging

Before adjusting garbage collector settings, learn to understand its behavior. GC logs are text files that provide exhaustive information about GC work: total GC time, memory reclamation and allocation, etc.

For JDK8 and below, the following options control GC logging:

JVM parameter	Description
-XX:PrintGC	Enables basic logging
-XX:+PrintGCDetails	Activates detailed logging
-XX:NumberOfGCLogFiles	Sets the limit for the number of GC logs
-XX:+UseGCLogFileRotation	Renames, archives, compresses or deletes log files when they are large and new logs are written directly to a new log file.
-XX:GCLogFileSize	Sets a maximum size of each Garbage Collector (GC) log.

JDK9 expands the unified logging framework for GC logging (JEP 271) so the logging option above is superseded with the `-Xlog` parameter. To learn more about the new logging syntax, run:

```
-Xlog:help
```

6. How to handle OutOfMemoryError

JVM parameter	Description
-XX:+HeapDumpOnOutOfMemoryError	Dumps heap into a file in the case of OutOfMemoryError
-XX:HeapDumpPath	Specifies the path for the file with heap data
-XX:OnOutOfMemoryError="< cmd args >;< cmd args >"	Specifies actions to be performed in the case of `OutOfMemoryError`

`OutOfMemoryError` leads to the application crash and is hard to troubleshoot. The above parameters provide the developers with a lot of information related to the error, so it is easier to detect memory leaks.

7. Working with Strings

JVM parameter	Description
-XX:+UseStringDeduplication	Removes duplicate strings during GC (with G1 GC only)
-XX:+UseStringCache	Caches commonly allocated strings in the String pool
-XX:+UseCompressedStrings	Uses a *byte[]* for Strings that can be represented as pure ASCII
-XX:+OptimizeStringConcat	Optimizes String concatenation operations when possible

`java.lang.String` is the most commonly used Java class. No wonder that Strings take up a significant part of the application memory. We can release the resources by removing duplicate strings and optimizing the String operations with the above parameters.

8. Other useful parameters

JVM parameter	Description
-XX:LargePageHeapSizeThreshold	Uses large pages if max. heap is at least as big as the specified value
-XX:LargePageSizeInBytes	Sets the large page size for the heap
-XX:+UseCompressedOops	Enables the use of compressed pointers (32-bit instead of 64-bit) for heaps less than 32 GB
-XX:+TieredCompilation	Disables intermediate compilation tiers
-XX:TieredStopAtLevel=1	Uses only the C1 compiler
-XX:ThreadStackSize	Sets the size of thread stack space

The `-XX:LargePageHeapSizeThreshold` and `-XX:LargePageSizeInBytes` flags enable the developers to operate with large pages (a technique to reduce the pressure on the processors Translation-Lookaside Buffer caches) and make better use of virtual hardware resources.

The `-XX:+TieredCompilation` and `-XX:TieredStopAtLevel=1` can be used with Serial GC to turn off the optimizing compiler and reduce memory footprint in some cases. Use them when memory consumption is the only important KPI.

Memory to thread stacks is allocated outside the heap, so it is not affected by heap size parameters. The `-XX:ThreadStackSize` flag enables the developers to reduce the size of thread stacks.

Appendix A: GC support in different environments

Standard JDK

The following tables list supported GCs depending on the JDK version, OS, and CPU type.

Mark	Legend
+	supported
-	not supported
E	supported as experimental feature: requires `-XX:+UnlockExperimentalVMOptions` vm option to be unlocked
N	build is not available

jdk6	CMS	G1	Parallel	Serial
Linux-x86	+	+	+	+
Linux-x86_64	+	+	+	+
Windows-x86	+	+	+	+
Windows-x86_64	+	+	+	+

jdk7	CMS	G1	Parallel	Serial
Linux-x86	+	+	+	+
Linux-x86_64	+	+	+	+
macOS-x86_64	+	+	+	+
Windows-x86	+	+	+	+
Windows-x86_64	+	+	+	+

jdk8	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-arm	+	-	-	+	+	-	-
Linux-aarch64	+	-	+	+	+	-	-
Linux-ppc32	+	-	-	+	+	-	-
Linux-ppc64le	+	-	+	+	+	-	-
Linux-x86	+	-	+	+	+	-	-
Linux-x86_64	+	-	+	+	+	-	-
macOS-aarch64	+	-	+	+	+	-	-
macOS-x86_64	+	-	+	+	+	-	-
Solaris-sparcv9	+	-	+	+	+	-	-
Solaris-x86_64	+	-	+	+	+	-	-

jdk8	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Windows-x86	+	-	+	+	+	-	-
Windowx-x86_64	+	-	+	+	+	-	-

jdk11	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-arm	+	E	+	+	+	-	-
Linux-aarch64	+	E	+	+	+	+	-
Linux-ppc64le	+	E	+	+	+	-	-
Linux-s390	+	E	+	+	+	-	-
Linux-x86	+	E	+	+	+	+	-
Linux-x86_64	+	E	+	+	+	+	E
macOS-aarch64	+	E	+	+	+	+	-
macOS-x86_64	+	E	+	+	+	+	-
Solaris-sparcv9	+	E	+	+	+	-	-
Solaris-x86_64	+	E	+	+	+	+	-
Windows-aarch64	+	E	+	+	+	+	-
Windows-x86	+	E	+	+	+	+	-
Windowx-x86_64	+	E	+	+	+	+	-

jdk17	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-arm	-	E	+	+	+	-	-
Linux-aarch64	-	E	+	+	+	+	+
Linux-ppc64le	-	E	+	+	+	+	+
Linux-s390	-	E	+	+	+	-	-
Linux-x86	-	E	+	+	+	+	-
Linux-x86_64	-	E	+	+	+	+	+
macOS-aarch64	-	E	+	+	+	+	+
macOS-x86_64	-	E	+	+	+	+	+
Windows-aarch64	-	E	+	+	+	+	+
Windows-x86	-	E	+	+	+	+	-

jdk17	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Windowx-x86_64	-	E	+	+	+	+	+[1]

jdk21	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-arm	-	E	+	+	+	-	-
Linux-aarch64	-	E	+	+	+	+	+
Linux-ppc64le	-	E	+	+	+	+	+
Linux-riscv	-	E	+	+	+	-	-
Linux-s390	-	E	+	+	+	-	-
Linux-x86	-	E	+	+	+	+	-
Linux-x86_64	-	E	+	+	+	+	+
macOS-aarch64	-	E	+	+	+	+	+
macOS-x86_64	-	E	+	+	+	+	+

jdk21	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Windows-aarch64	-	E	+	+	+	+	+
Windows-x86	-	E	+	+	+	+	-
Windwx-x86_64	-	E	+	+	+	+	+ [1]

JDK Performance Edition

The following tables list supported GCs depending on the JDK version, OS, and CPU type.

jdk8perf	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-arm	-	E	+	+	+	-	-
Linux-aarch64	-	E	+	+	+	+	+
Linux-ppc32	N	N	N	N	N	N	N
Linux-ppc64le	-	E	+	+	+	+	+
Linux-x86	-	E	+	+	+	+	-
Linux-x86_64	-	E	+	+	+	+	+

jdk8perf	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
macOS-aarch64	-	E	+	+	+	+	+
macOS-x86_64	-	E	+	+	+	+	+
Solaris-sparcv9	N	N	N	N	N	N	N
Solaris-x86_64	N	N	N	N	N	N	N
Windows-x86	-	E	+	+	+	+	-
Windowx-x86_64	-	E	+	+	+	+	+ [1]

jdk11perf	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-arm	-	E	+	+	+	-	-
Linux-aarch64	-	E	+	+	+	+	+
Linux-ppc64le	-	E	+	+	+	+	+
Linux-s390	-	E	+	+	+	-	-
Linux-x86	-	E	+	+	+	+	-
Linux-x86_64	-	E	+	+	+	+	+

jdk11perf	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
macOS-aarch64	-	E	+	+	+	+	+
macOS-x86_64	-	E	+	+	+	+	+
Solaris-sparcv9	N	N	N	N	N	N	N
Solaris-x86_64	N	N	N	N	N	N	N
Windows-aarch64	-	E	+	+	+	+	+
Windows-x86	-	E	+	+	+	+	-
Windowx-x86_64	-	E	+	+	+	+	+ [1]

jdk17perf	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-arm	-	E	+	+	+	-	-
Linux-aarch64	-	E	+	+	+	+	+
Linux-ppc64le	-	E	+	+	+	+	+
Linux-s390	-	E	+	+	+	-	-

jdk17perf	CMS	Epsilon	G1	Parallel	Serial	Shenandoah	Z
Linux-x86	-	E	+	+	+	+	-
Linux-x86_64	-	E	+	+	+	+	+
macOS-aarch64	-	E	+	+	+	+	+
macOS-x86_64	-	E	+	+	+	+	+
Windows-aarch64	-	E	+	+	+	+	+
Windows-x86	-	E	+	+	+	+	-
Windowx-x86_64	-	E	+	+	+	+	+ [1]

1. Supported on Windows version 10 or later (2019 or later for server version).

Liberica JDK
Guide to JVM memory configuration options

bell//soft