Memory is deivided into 2 big parts ?

* Stack
* Heap

Stack

* It is responsible for holding references to heap objects and also storing value types( primitive types) which holds the value itself rather than reference to object from heap.
* The compiler executes the method’s body.Once the method completes and return the top of stack pops out and active scope changes.
* Each thread has its own stack memory.

Heap:

* Only one heap memory for each running JVM process.
* Shared part of memory regardless of how many threads are running.

Reference types :

[Reference (Java SE 20 & JDK 20) (oracle.com)](https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/ref/Reference.html) – Abstract sealed class

* Abstract base class for reference objects. This class defines the operations common to all reference objects. Because reference objects are implemented in close cooperation with the garbage collector, this class may not be subclassed directly.
* There 4 types of reference
  + Strong
  + Weak
  + Soft
  + Phanthom
* Strong reference :
  + The object cant be garbage collected if its reachable through any strong reference
  + Example : List<String> ltrString = new ArrayList();
* Soft Reference :

[SoftReference (Java SE 20 & JDK 20) (oracle.com)](https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/ref/SoftReference.html) -non sealed class

* + Used to implement memory -sensitive caches
  + These are going to be garbage collected when application is running low on memory.
  + Java guarantees soft references are cleaned up before it throws OutOfMemoryError.

SoftReference<StringBuilder> reference = new SoftReference<>(new StringBuilder());

* weak reference

Weak references are often used to implement canonicalizing mappings. A canonicalizing mapping is a map that only contains one instance of each object. For example, you could use a weak reference to implement a map of usernames to user objects. If two different users have the same username, the garbage collector will collect one of the user objects and the weak reference will point to the other user object. This ensures that the map only contains one instance of each user object.

* Phantom reference:

Similarly to weak references, [phantom references](https://www.baeldung.com/java-phantom-reference) don't prohibit the garbage collector from enqueueing objects for being cleared. The difference is **phantom references must be manually polled from the reference queue before they can be finalized**. That means we can decide what we want to do before they are cleared.

What is garbage collection

* Process in which programs try to free up memory space that no longer used by objects.
* High-level programming language have some sort of garbage collection built-in
  + Python
  + JavaScript
  + Java
  + C++
  + C#
  + Ruby
  + Swift
  + PHP
  + Go
  + Rust
* Low-level programming language may add garbage collection through libraries
  + X86 Assembly Language
  + ARM Assembly Language
  + MIPS Assembly Language
  + PowerPC Assembly language
  + SPARC Assembly Language
* In C programming, developers has to take care of memory allocation and deallocation using malloc() and dealloc() functions. But in C# do need to take care of GC .
* In C# , memory allocation of objects happens in managed heap, which is taken care by CLR -common language runtime.
* Memory allocation in heap is done by win32 dll in OS similarly to C.
* Memory mapping works based on LinkedList concepts. (one after the other)

Advantage of GC

* Its tool that saves time for programmers.
* It helps in memory leaks

Disadvantage of GC

* It has negative impact on performance
* While checking for object references and cleaning out memory it oftem requires program to pause.

Sufficient memory not available for creation of new objects and entire program terminates abnormally due to OutOfMemoryErrors.

Free() in C and delete() in C++ to perfrom Garbage collection.

* When java program runs on JVM , objects are created on heap, which is portion of memory dedicated to program.
* Heap memory has 2 types of objects
* - Live – The objects are being used and referenced from somewhere
* Dead – Objects are no longer used or refrenced from anywhere.

How to make eligible for Garbage collection ?

* By making reference null

|  |
| --- |
| * Student student = new Student(); * student = null; |

* By assigning a refrence to another

|  |
| --- |
| * Student studentOne = new Student(); * Student studentTwo = new Student(); * studentOne = studentTwo; // now the first object referred by studentOne is available for garbage collection |

* By using an anonymous object
  + Register(new Student();
* How garbage collection works?
* Implementation lives in JVM
* Each JVM can implement its own version of garbage collection
* Marking or identifying the unreachable objects and destroying them with compaction.
* GC works on concept of Garbage collection Roots(GC Roots ) to indentify live and dead objects .
* Examples of Grabage collection roots are
* - Classes loaded by system class loader (not custome class loaders)
* Live threads
* Local variables and parameters of currently executing methods
* Local variables and parameters of JNI methods
* Global JNI reference
* Objects used as monitor for synchronization
* Object held from garbage collection by JVM for its purpose

Phases of Garbage collection in Java

* In this step, the GC identifies all the live objects in memory by traversing the object graph.
* When GC visits an object, it marks it as accessible and thus alive. Every object the garbage collector visits is marked as alive. All the objects which are not reachable from GC Roots are garbage and considered as candidates for garbage collection.

**Sweep dead objects**

After marking phase, we have the memory space which is occupied by live (visited) and dead (unvisited) objects. The sweep phase releases the memory fragments which contain these dead objects.

### Compact remaining objects in memory

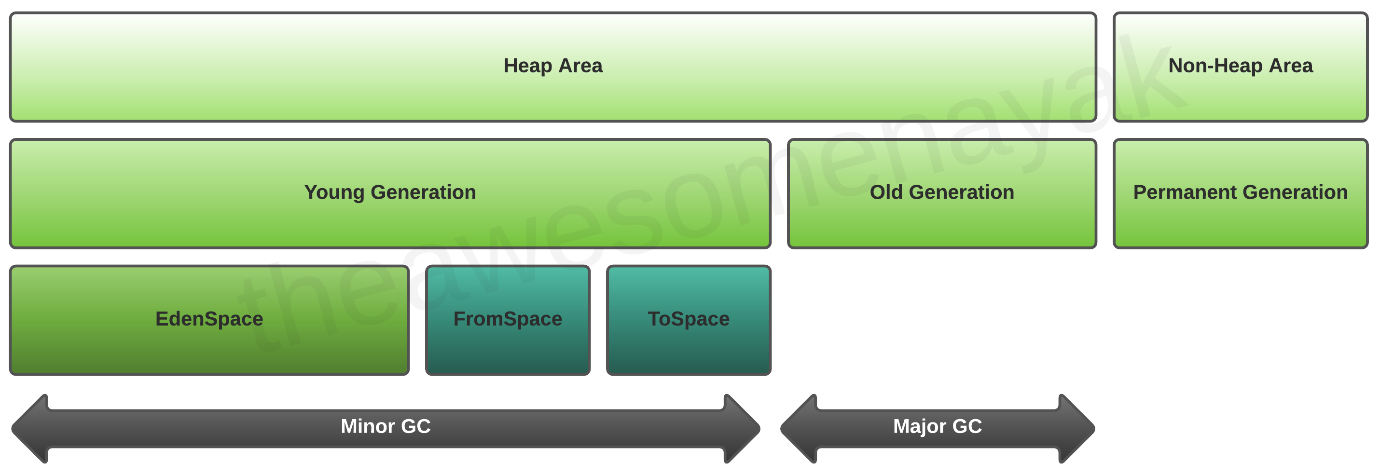
The dead objects that were removed during the sweep phase may not necessarily be next to each other. Thus, you can end up having fragmented memory space.

Memory can be compacted after the garbage collector deletes the dead objects, so that the remaining objects are in a contiguous block at the start of the heap.

The compaction process makes it easier to allocate memory to new objects sequentially.

Java cataegorizes the objects into generations and perfromsn the garbasge collection accordingly.

Heap memeory in JVM is divided into 3 secttions



The JVM heap is typically divided into three main areas: Young Generation, Old Generation, and Permanent Generation (or Metaspace, depending on the JVM version). The Young Generation is further divided into two Survivor Spaces and an Eden Space.

1. Eden Space: When objects are created in Java, they are initially allocated in the Eden Space, which is a part of the Young Generation. The Eden Space is usually where the majority of short-lived objects reside.
2. Survivor Spaces: The two Survivor Spaces (often called Survivor Space 0 and Survivor Space 1 or S0 and S1) are also part of the Young Generation. When a minor garbage collection (also known as minor GC or young GC) occurs, live objects from the Eden Space and one of the Survivor Spaces are moved to the other empty Survivor Space. The goal of this process is to identify and eliminate short-lived objects, leaving the long-lived ones to be moved to the Old Generation.
3. Old Generation: Objects that survive multiple rounds of minor GC are eventually promoted to the Old Generation. These are typically objects with longer lifetimes, and they are collected during a major garbage collection (full GC) cycle.

The division of the heap into these generations with separate Survivor Spaces allows the JVM to optimize garbage collection performance. Young Generation garbage collections are generally faster and more frequent, while Old Generation collections are more time-consuming and occur less often.

It's worth noting that JVM implementations can vary, and some may use different garbage collection algorithms or strategies. For example, the G1 (Garbage-First) Garbage Collector introduced in JDK 7 combines regions of the heap for young and old objects, removing the strict separation of Survivor Spaces. However, the concept of separating short-lived and long-lived objects remains fundamental to JVM garbage collection strategies.

Young Generation

Old Generation

Permanent Generation

MetaSpace

Types of garbage Collector

* Serial GC
* Parallel GC
* Parallel Old GC
* CMS (Concurrent Mark Sweep) GC
* G1 – Garbage First GC
* Epsilon Garbage collector
* Shenandoah
* ZGC

How to sleect right Garbage collector

Advantage of Garbage collector

Garbage collector Best practices

-Avoid Maual Triggers

Use tool for analysis

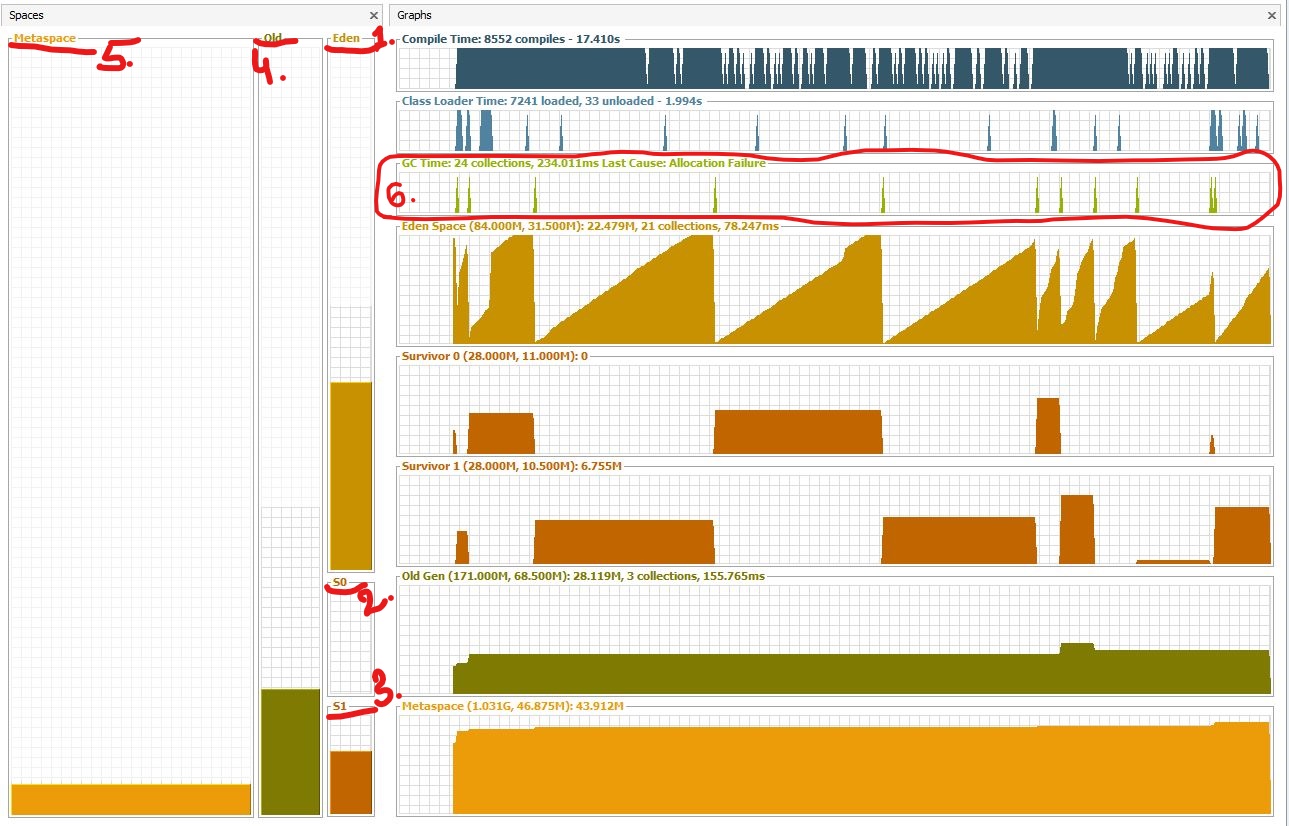
Defult setting as Good

Use JVM Flags for tuning

Select the right collector

Since this is a quite complex process, and it might affect you performance, it is implemented in a smart way. A so-called “Mark and Sweep” process is used for that. Java analyzes the variables from the stack and “marks” all the objects that need to be kept alive. Then, all the unused objects are cleaned up.

So actually, Java does not collect any garbage. In fact, the more garbage there is, and the fewer that objects are marked alive, the faster the process is. To make this even more optimized, heap memory actually consists of multiple parts. We can visualize the memory usage and other useful things with **JVisualVM**, a tool that comes with the Java JDK. The only thing you have to do is install a plugin named **Visual GC**, which allows you to see how the memory is actually structured. Let’s zoom in a bit and break down the big picture:

Heap memory generations

When an object is created, it is allocated on the **Eden(1)** space. Because the Eden space is not that big, it gets full quite fast. The garbage collector runs on the Eden space and marks objects as alive.

Once an object survives a garbage collecting process, it gets moved into a so-called survivor space **S0(2)**. The second time the garbage collector runs on the Eden space, it moves all surviving objects into the **S1(3)** space. Also, everything that is currently on **S0(2)** is moved into the **S1(3)** space.

If an object survives for X rounds of garbage collection (X depends on the JVM implementation, in my case it’s 8), it is most likely that it will survive forever, and it gets moved into the **Old(4)** space.

Taking everything said so far, if you look at the **garbage collector graph(6)**, each time it has run, you can see that the objects switch to the survivor space and that the Eden space gained space. And so on and so forth. The old generation can be also garbage collected, but since it is a bigger part of the memory compared to Eden space, it does not happen that often. The **Metaspace(5)** is used to store the metadata about your loaded classes in the JVM.

The presented picture is actually a Java 8 application. Prior to Java 8, the structure of the memory was a bit different. The metaspace is called actually the PermGen. space. For example, in Java 6, this space also stored the memory for the string pool. Therefore, if you have too many strings in your Java 6 application, it might crash.

### ****Garbage Collector Types****

Actually, the JVM has three types of garbage collectors, and the programmer can choose which one should be used. By default, Java chooses the garbage collector type to be used based on the underlying hardware.

**1. Serial GC** – A single thread collector. Mostly applies to small applications with small data usage. Can be enabled by specifying the command line option: -XX:+UseSerialGC

**2. Parallel GC** – Even from the naming, the difference between Serial and Parallel would be that Parallel GC uses multiple threads to perform the garbage collecting process. This GC type is also known as the throughput collector. It can be enabled by explicitly specifying the option: -XX:+UseParallelGC

**3. Mostly concurrent GC** – If you remember, earlier in this article, it was mentioned that the garbage collecting process is actually pretty expensive, and when it runs, all thread are paused. However, we have this mostly concurrent GC type, which states that it works concurrent to the application. However, there is a reason why it is “mostly” concurrent. It does not work 100% concurrently to the application. There is a period of time for which the threads are paused. Still, the pause is kept as short as possible to achieve the best GC performance. Actually, there are 2 types of mostly concurrent GCs:

**3.1 Garbage First** – high throughput with a reasonable application pause time. Enabled with the option: -XX:+UseG1GC

**3.2 Concurrent Mark Sweep** – The application pause time is kept to a minimum. It can be used by specifying the option: -XX:+UseConcMarkSweepGC. As of JDK 9, this GC type is deprecated.

## **Tips and Tricks**

* To minimize the memory footprint, limit the scope of the variables as much as possible. Remember that each time the top scope from the stack is popped up, the references from that scope are lost, and this could make objects eligible for garbage collecting.
* Explicitly refer to null obsolete references. That will make objects those refer to eligible for garbage collecting.
* Avoid finalizers. They slow down the process and they do not guarantee anything. Prefer phantom references for cleanup work.
* Do not use strong references where weak or soft references apply. The most common memory pitfalls are caching scenarios,when data is held in memory even if it might not be needed.
* JVisualVM also has the functionality to make a heap dump at a certain point, so you could analyze, per class, how much memory it occupies.
* Configure your JVM based on your application requirements. Explicitly specify the heap size for the JVM when running the application. The memory allocation process is also expensive, so allocate a reasonable initial and maximum amount of memory for the heap. If you know it will not make sense to start with a small initial heap size from the beginning, the JVM will extend this memory space. Specifying the memory options with the following options:
  + Initial heap size -Xms512m – set the initial heap size to 512 megabytes.
  + Maximum heap size -Xmx1024m – set the maximum heap size to 1024 megabytes.
  + Thread stack size -Xss1m – set the thread stack size to 1 megabytes.
  + Young generation size -Xmn256m – set the young generation size to 256 megabytes.
* If a Java application crashes with an OutOfMemoryError and you need some extra info to detect the leak, run the process with the –XX:HeapDumpOnOutOfMemory parameter, which will create a heap dump file when this error happens next time.
* Use the -verbose:gc option to get the garbage collection output. Each time a garbage collection takes place, an output will be generated.

## **Conclusion**

Knowing how memory is organized gives you the advantage of writing good and optimized code in terms of memory resources. In advantage, you can tune up your running JVM, by providing different configurations that are the most suitable for your running application. Spotting and fixing memory leaks is just an easy thing to do, if using the right tools.