Garbage Collection in Java

Garbage Collection (GC) in Java is a process by which the Java Virtual Machine (JVM) automatically reclaims memory that is no longer in use, freeing developers from manually managing memory. Here's a detailed look at various aspects of Java's garbage collection:

1. Making an Object Eligible for GC

An object becomes eligible for garbage collection when there are no more references to it, meaning that it cannot be reached or used by any part of the program. Here's how you can make an object eligible for GC:

- **Remove References:** Set all references to the object to `null`. For example:

```
MyObject obj = new MyObject();
obj = null;  // The object created is now eligible for GC
```

- **Scope Limitation:** Allow the object to go out of scope, such as by exiting a method where the object was created.

```
public void myMethod() {
        MyObject obj = new MyObject();
        // obj is eligible for GC once myMethod exits

    }
```

- **Use Data Structures:** Remove references from data structures like lists or maps.

```
    List<MyObject> list = new ArrayList<>();
    MyObject obj = new MyObject();
     list.add(obj);
      list.remove(obj);  // obj is now eligible
for GC if there are no other references
```

2. Requesting JVM to Run Garbage Collector

While the JVM's garbage collector runs automatically, you can request it to run explicitly, though it's not guaranteed that it will execute immediately.

- **Using `System.gc()` Method:**

```
System.gc();
```

This method is a hint to the JVM that it may be a good time to run the garbage collector. However, the JVM is free to ignore this request.

- **Using `Runtime.getRuntime().gc()` Method:**

```
Runtime.getRuntime().gc();
```

This is equivalent to `System.gc()` and also serves as a hint to the JVM.

Note: Forcing garbage collection is generally not recommended in production code as it can affect performance and is often unnecessary. Relying on the JVM's automatic GC is usually preferable.

3. How and When to Use Finalization

Finalization is a mechanism that allows objects to clean up resources before they are garbage collected. However, it is not often used in modern Java due to potential performance impacts and unpredictability.

- **Using `finalize()` Method:**

```
public class MyResource {
            @Override
            protected void finalize() throws Throwable {
                try {
                    // Clean up resources here
                } finally {
                    super.finalize();
                }
            }
        }
```

When to Use Finalization:

- **Resource Cleanup:** When an object needs to release non-memory resources (e.g., file handles, network connections) before it is collected.

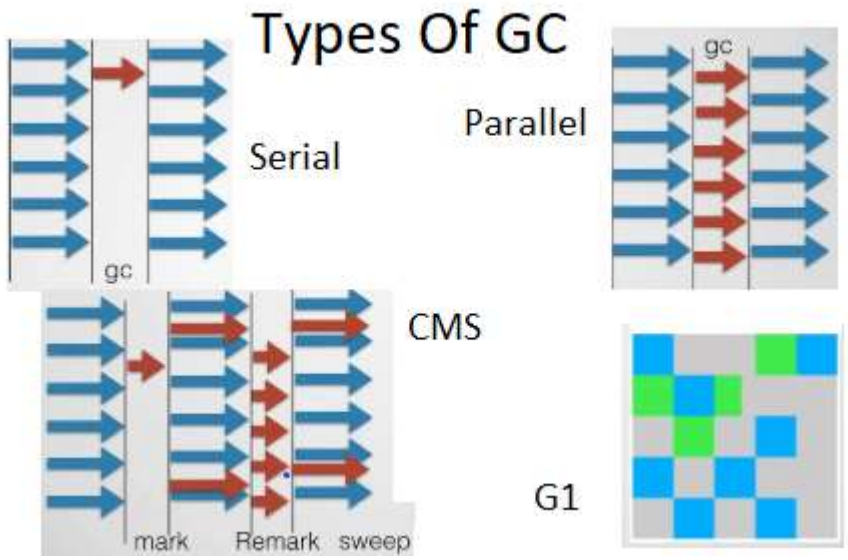
- **Legacy Code:** In some older codebases where finalization was more common.

Note: Finalization is considered unreliable because:

- The timing of finalization is unpredictable.
- The finalize() method may not run promptly, which can lead to resource leaks.

As of Java 9, the finalize() method is deprecated in favor of using try-with-resources and AutoCloseable for resource management.

4. Types of JVM Garbage Collectors

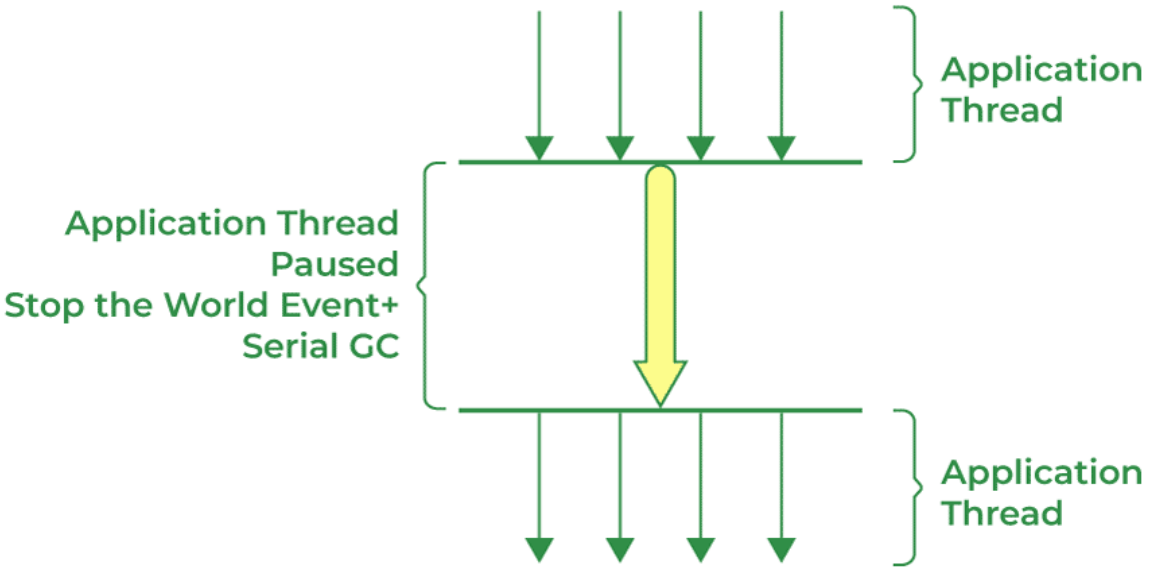


Types of Garbage Collectors in HotSpot Java

There are mainly 4 algorithms for garbage collectors in HotSpot Java:

1. Serial Garbage Collection

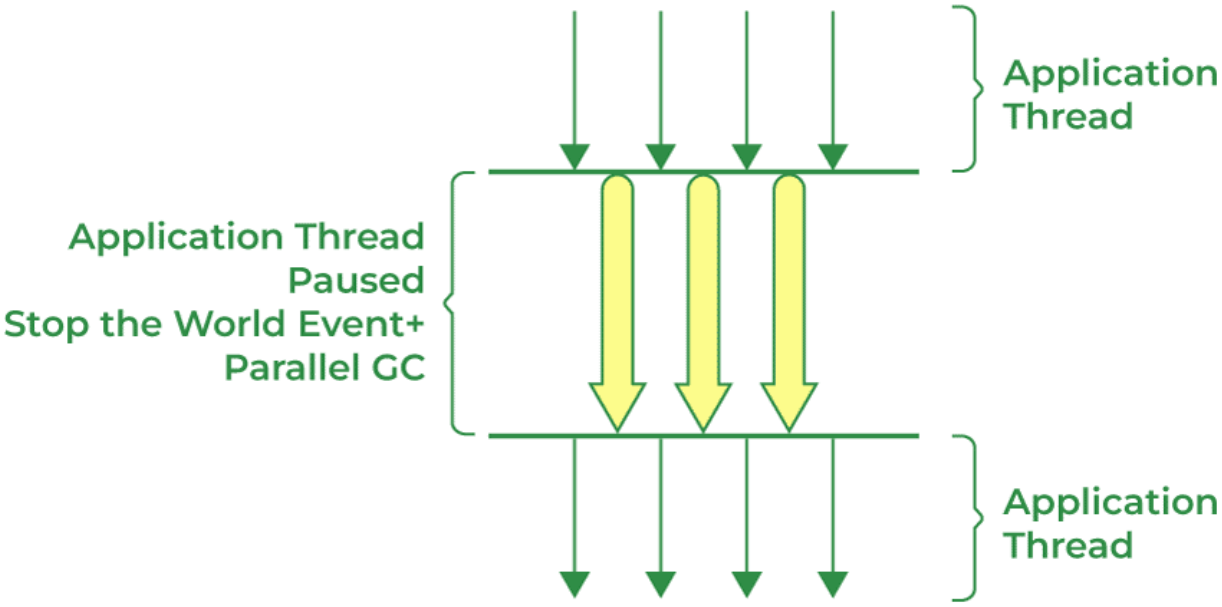
It is used where just one thread executed the GC (garbage collection). This collector freezes all application threads whenever it's working. If we want to use the serial garbage collector, execute the -XX:+UseSerialGC JVM argument to activate it.



Serial GC

2. Parallel Garbage Collection

Where multiple minor threads are executed simultaneously. This collector also freezes all application threads whenever it's working. If you want to use the parallel garbage collector, execute the -XX:+UseParallelGC JVM argument to activate it.

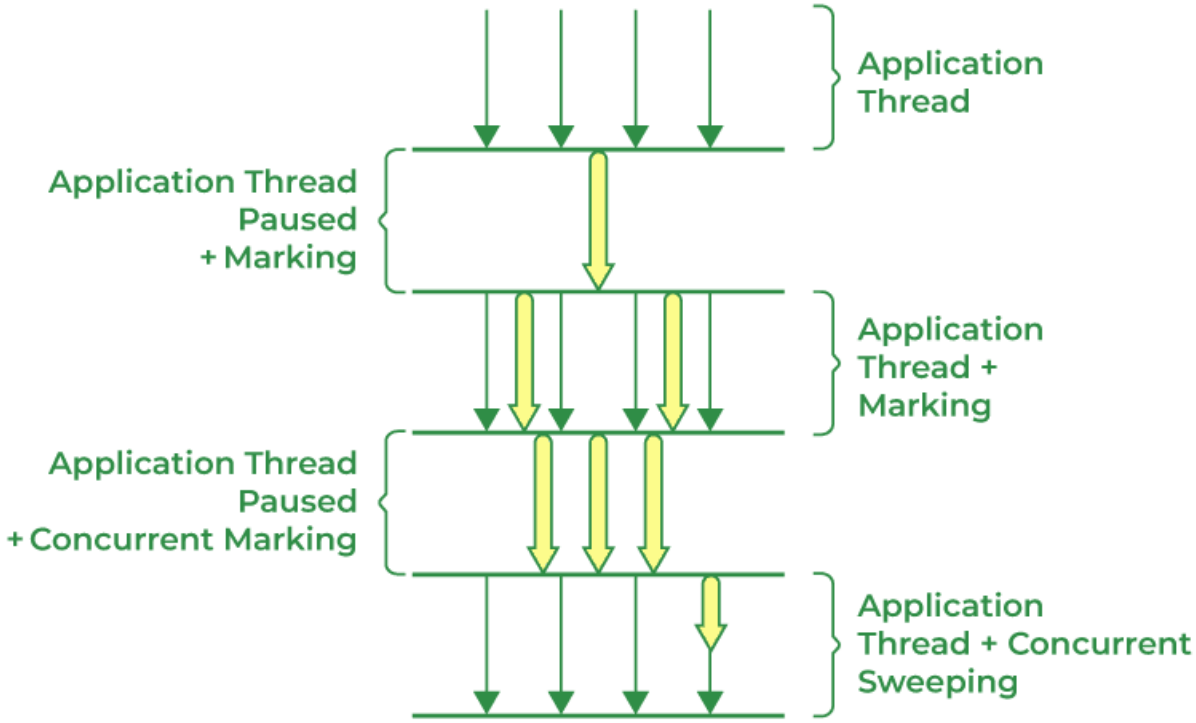


Parallel GC

3. Concurrent Mark Sweep(CMS) Garbage Collection

It is similar to parallel, also allows the execution of some application threads, and reduces the frequency of stop-the-world GC. It has 2 phases: **Mark Phase and Sweep Phase**. The Mark phase algorithm marks all the live objects in heap memory (those that have a reference) as "live" and others as "dead" while putting the applications on pause. The Sweep phase algorithm empties the space occupied by objects marked "dead". In the Mark phase, multiple minor threads are executed simultaneously to mark the live objects. In the Sweep phase, multiple minor threads are executed simultaneously to sweep or delete the dead objects parallel to application threads. For this reason, it uses more CPU in comparison to other GC. It is also known as the concurrent low pause collector. If you want to use a CMS garbage collector, execute the -XX:+USeParNewGC JVM argument to activate it. We can also set the

number of GC threads by using the -XX:ParallelCMSThreads=<n> JVM argument.



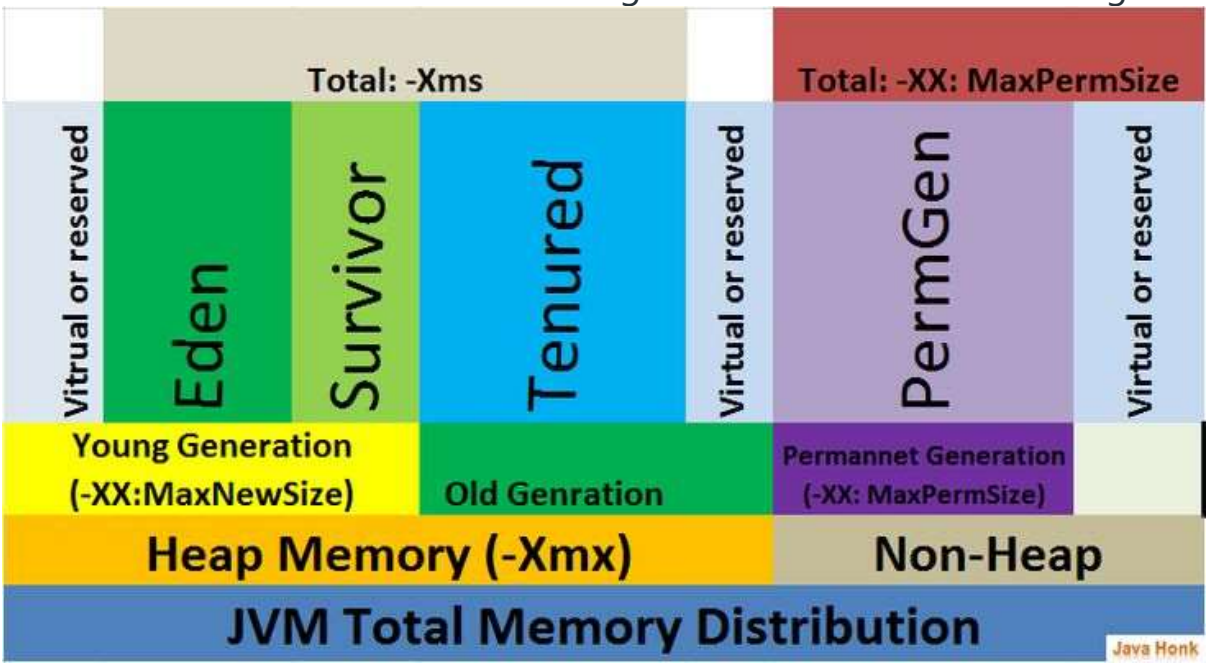
CMS Garbage Collection

***Note**: The Concurrent Mark Sweep (CMS) collector is deprecated as of JDK 9, with the recommendation to use the G1 collector instead.*

4. Generation First(G1) Garbage Collection

This GC is used if we have huge memory(>4GB). It partitions the heap into a set of equal size regions (1MB to 32MB - depending on the size of the heap) and uses multiple threads to scan them and mark the objects as live or dead. After marking, G1 knows which regions contain the most garbage objects. In the next step, it sweeps the objects in the region containing most garbage objects. G1 works concurrently with the application. This divides young and old generations of the heap into

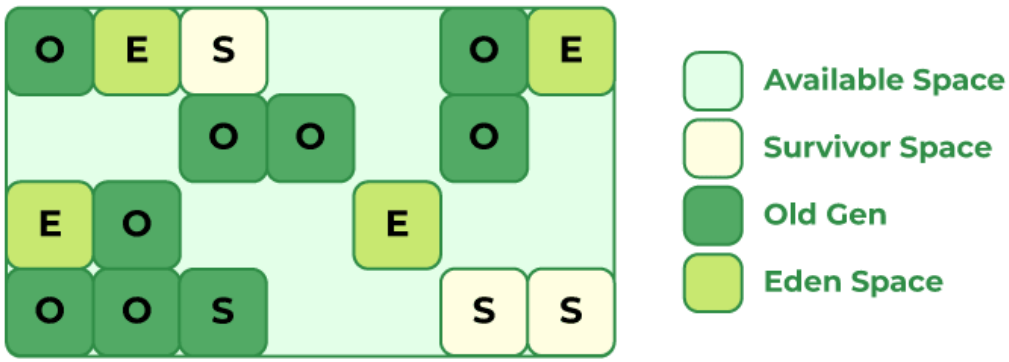
the following regions:



- **Eden region** in memory where the objects are typically allocated when they are created.
- **Survivor Region** contains the objects that have survived from the Young garbage collection or Minor garbage collection.
- **Tenured Region** is where long-lived objects are stored. Objects are eventually moved to this space if they survive a certain number of garbage collection cycles. This space is much larger than the Eden space and the garbage collector checks it less often.
- **Humongous**: It is used if the object size is large.
- **Available**: It represents the unoccupied space.

An optimization is present in Java 8 (G1 Collector String deduplication). Since strings (and their internal char[] arrays) takes much of our heap, a new optimization has been made that enables the G1 collector to identify strings that are duplicated more than once across heap memory and correct them to point to the same internal char[] array, to avoid multiple copies of the same string from residing inefficiently

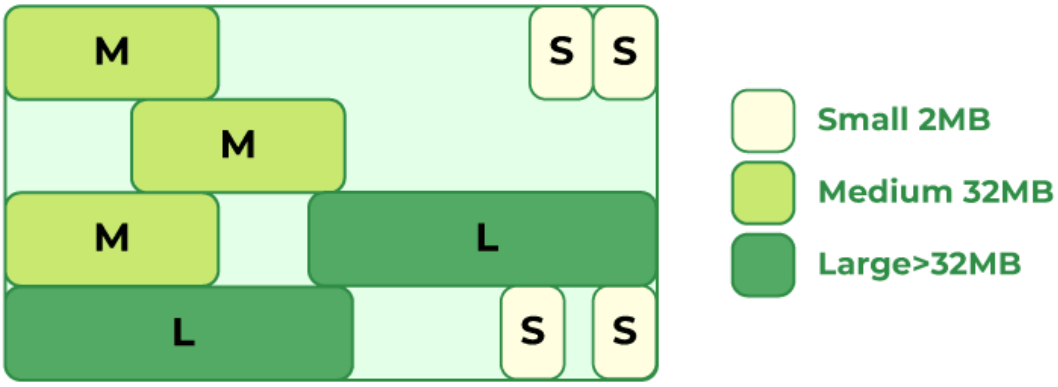
within the heap. If you want to use G1 GC use -XX:+UseStringDeduplicationJVM argument.



G1 GC Space Description

5. Z Garbage Collector

It's the latest GC that can be used with Java 11 by explicitly setting it, and in Java 15 and beyond this is set as default. It's a highly scalable, low-latency GC. It performs very expensive tasks concurrently without stopping the working of threads for more than 10ms. It can work on large heap memory (up to TBs). If you want to use ZGC, use XX:+UnlockExperimentalVMOptions -XX:+UseZGC.



ZGC Heap Regions

Methods for Running Garbage Collector in Java

In Java, GC runs by itself. But there are some methods we can use in Java:

- System.gc()
- Runtime.getRuntime().gc()

Best Practices to Use Garbage Collection

Collector choice is a primary concern as the wrong collector can hinder performance and render the application useless. For **backend purposes**, the **CMS collector** can prove to be useful in spite of having the disadvantage of stopping the application itself during garbage collection. However, the **frontend** of the application must not implement CMS because the UI must always be responsive. This calls for **G1 collectors** which can run concurrently with the application in itself. **ZGC** is the newest and provides very low latency and high throughput which can be used by applications that require low latency and uses large-size Heap. Whenever you are using garbage collections in Java it is important to note that you can never predict when the garbage collector will run. You can try to explicitly call the garbage collector by **System.gc()** and **Runtime.gc()**. However, even after explicitly calling the Garbage collector there is **no guarantee** that it will actually work.

Here’s a full example demonstrating how to work with Java’s garbage collection features, including making objects eligible for GC, requesting GC, and using finalization. I'll also provide examples of different garbage collectors, but remember that the selection of a garbage collector is typically done via JVM flags, not in code.

Example: Garbage Collection in Java

1. Making an Object Eligible for GC

This example demonstrates creating an object and making it eligible for garbage collection:

```
package com.mphasis.generics;
public class GCExample {
    public static void main(String[] args) {
        // Create an object and print its reference
        MyClass obj = new MyClass("Object 1");
        System.out.println("Created: " + obj);

        // Make the object eligible for GC
        obj = null;

        // Suggest the JVM run garbage collection
        System.gc();

        // Wait for some time to allow GC to complete
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Create another object
        MyClass obj2 = new MyClass("Object 2");
        System.out.println("Created: " + obj2);
    }
}

class MyClass {
    private String name;

    public MyClass(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "MyClass{name='" + name + "'}";
    }

    @Override
    protected void finalize() throws Throwable {
```

```
        System.out.println("Finalizing: " + name);
        super.finalize();
    }
}
```

Explanation

- **Making Object Eligible for GC:** The object obj is assigned null, which removes the reference to it and makes it eligible for garbage collection.
- **Requesting GC:** System.gc() is called to suggest that the JVM perform garbage collection. This is not guaranteed but serves as a hint.
- **Finalization:** The finalize() method is overridden to print a message when the object is finalized before being collected by the garbage collector.

Running with Different Garbage Collectors

1. Serial Garbage Collector:

```
java -XX:+UseSerialGC GCExample
```

2. Parallel Garbage Collector:

```
java -XX:+UseParallelGC GCExample
```

3. Concurrent Mark-Sweep (CMS) Garbage Collector:

```
java -XX:+UseConcMarkSweepGC GCExample
```

4. G1 Garbage Collector:

```
java -XX:+UseG1GC GCExample
```

5. Z Garbage Collector (ZGC):

```
java -XX:+UseZGC GCExample
```

6. Shenandoah Garbage Collector:

```
java -XX:+UseShenandoahGC GCExample
```

Notes:

- **System.gc():** This is a hint to the JVM and doesn't guarantee immediate garbage collection.
- **finalize():** The finalize() method is deprecated in Java 9 and removed in Java 18. Use try-with-resources and AutoCloseable for resource management in modern Java.
- **GC Types:** The garbage collector you use can be specified through JVM flags, as shown above. The choice of GC depends on your application's requirements regarding throughput, pause times, and heap size.