Algoritham of GC

1) **Mark-and-Sweep Algorithm**:

* **Mark Phase**: The GC identifies and marks all live objects (objects that are still reachable from the root references).
* **Sweep Phase**: The GC traverses the heap and deallocates memory occupied by unmarked objects.

2) **Mark-and-Compact Algorithm**:

* **Mark Phase**: Identical to the Mark-and-Sweep algorithm.
* **Compact Phase**: After sweeping, the memory is compacted by moving live objects together, which helps reduce fragmentation.

3) **Copying Algorithm**:

* The Young Generation is divided into two halves (Eden and Survivor Spaces). Objects are initially allocated in Eden. During garbage collection, live objects are copied from Eden to one of the Survivor Spaces. After several collections, objects are promoted to the Old Generation.

TYPES OF GC

Java provides several types of garbage collectors (GCs), each designed to address different performance and application requirements. The choice of garbage collector can significantly impact the performance characteristics of your Java application. Here’s an overview of the main types:

**1. Serial Garbage Collector**

* **Description**: The Serial Garbage Collector uses a single thread for garbage collection. It’s the simplest GC algorithm and is suitable for single-threaded applications or applications with small heaps.
* **Characteristics**:
  + **Single-threaded**: All garbage collection work is done by one thread.
  + **Stop-the-world**: When garbage collection occurs, it pauses all application threads.
* **Use Case**: Small applications or applications running on single-core machines.
* **JVM Option**: -XX:+UseSerialGC

**2. Parallel Garbage Collector (Throughput Collector)**

* **Description**: The Parallel Garbage Collector uses multiple threads to perform garbage collection, which can improve throughput by utilizing multiple CPU cores. It is the default garbage collector in many JVM implementations.
* **Characteristics**:
  + **Multi-threaded**: Utilizes multiple threads for minor GC (Young Generation) and may use a single thread for major GC (Old Generation).
  + **Stop-the-world**: Like the Serial GC, it pauses all application threads during collection.
* **Use Case**: Applications that benefit from high throughput and can tolerate moderate pause times.
* **JVM Option**: -XX:+UseParallelGC or -XX:+UseParallelOldGC (for the Old Generation).

**3. Concurrent Mark-Sweep (CMS) Collector**

* **Description**: The CMS Garbage Collector is designed to minimize pause times by performing most of its work concurrently with the application threads. It aims to provide a more responsive experience by reducing the time spent in stop-the-world pauses.
* **Characteristics**:
  + **Concurrent**: Performs much of the garbage collection work concurrently with application threads.
  + **Stop-the-world**: There are still some stop-the-world pauses, but they are shorter compared to other collectors.
  + **Final Remark**: In CMS, there can be some fragmentation in the Old Generation.
* **Use Case**: Applications with low-latency requirements and acceptable throughput.
* **JVM Option**: -XX:+UseConcMarkSweepGC

**4. Garbage-First (G1) Collector**

* **Description**: The G1 Garbage Collector is designed for applications with large heaps and aims to balance pause times and throughput. It divides the heap into regions and collects them in a way that prioritizes regions with the most garbage.
* **Characteristics**:
  + **Region-based**: The heap is divided into fixed-size regions. G1 prioritizes regions with the most garbage.
  + **Predictable**: Offers more control over GC pause times with tuning options.
* **Use Case**: Applications with large heaps where pause-time predictability is important.
* **JVM Option**: -XX:+UseG1GC

**5. Z Garbage Collector (ZGC)**

* **Description**: The ZGC is designed to provide extremely low pause times even for very large heaps. It is a low-latency collector that aims to keep GC pauses short and predictable.
* **Characteristics**:
  + **Low-latency**: Designed to handle large heaps with minimal pause times.
  + **Concurrent**: Most of the work is done concurrently with application threads.
* **Use Case**: Applications with very large heaps and strict low-latency requirements.
* **JVM Option**: -XX:+UseZGC

**6. Shenandoah Garbage Collector**

* **Description**: Shenandoah is another low-latency garbage collector that aims to minimize GC pause times. It’s designed to handle large heaps with reduced pause times similar to ZGC.
* **Characteristics**:
  + **Low-latency**: Provides short and predictable pause times.
  + **Concurrent**: Performs most of the garbage collection work concurrently with application threads.
* **Use Case**: Applications with large heaps and a need for low pause times.
* **JVM Option**: -XX:+UseShenandoahGC

**7. Epsilon Garbage Collector**

* **Description**: The Epsilon Garbage Collector is a no-op garbage collector that does not actually perform garbage collection. It is intended for testing and scenarios where garbage collection is not needed (e.g., when the application manages memory manually).
* **Characteristics**:
  + **No GC**: Does not perform any garbage collection. It’s useful for scenarios where you want to measure application performance without the impact of GC.
* **Use Case**: Testing and applications where garbage collection is not required.
* **JVM Option**: -XX:+UseEpsilonGC

**Choosing a Garbage Collector**

Selecting the right garbage collector depends on your application's needs:

* **Throughput**: For applications where high throughput is more important than low latency (e.g., batch processing), the Parallel GC is a good choice.
* **Low Latency**: For applications where minimizing pause times is critical (e.g., real-time applications), consider CMS, G1, ZGC, or Shenandoah.
* **Large Heaps**: For applications with very large heaps, G1, ZGC, or Shenandoah may be appropriate.

Tuning garbage collection often involves testing and profiling to find the best configuration for your specific workload and performance goals.