**Java garbage collection**

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

Garbage collection (GC) is the process that aims to free up occupied memory that is no longer referenced by any reachable Java object, and is an essential part of the Java virtual machine's (JVM's) dynamic memory management system. In a typical garbage collection cycle all objects that are still referenced, and thus reachable, are kept. The space occupied by previously referenced objects is freed and reclaimed to enable new object allocation.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### Advantage of Garbage Collection

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By annonymous object etc.

### 1) By nulling a reference:

1. Employee e=new Employee();
2. e=null;



### 2) By assigning a reference to another:

1. Employee e1=new Employee();
2. Employee e2=new Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

### 3) By annonymous object:

1. new Employee();

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. protected void finalize(){}



#### Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. public static void gc(){}



#### Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

### Simple Example of garbage collection in java

1. public class TestGarbage1{
2. public void finalize(){System.out.println("object is garbage collected");}
3. public static void main(String args[]){
4. TestGarbage1 s1=new TestGarbage1();
5. TestGarbage1 s2=new TestGarbage1();
6. s1=null;
7. s2=null;
8. System.gc();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGarbage1)

object is garbage collected

object is garbage collected

#### Note: Neither finalization nor garbage collection is guaranteed

**Garbage Collection : Destroying Object in Java Programming**

1. Object is Created Using **new Operator**.
2. Object gets memory inside “**Heap**“.
3. In C++ after Using Object, **Memory for Object gets de-allocated using delete()**.
4. In Java **De-allocation of Object can be done automatically**.
5. Automatic Technique used in Java Programming Which is used to de-allocate memory is called as “**Garbage Collection**“.
6. Java is Smart Enough to identify the **Unused objects or Useless Objects**.

**Explanation :**

1. Garbage Collection is done **Automatically by JVM**.
2. As soon as compiler detects that – Object is no **longer needed inside program** , Garbage Collection Algorithm gets executed automatically to free up memory from the heap so that free memory may be used by other objects .
3. Different Java Run times may have **different approaches for Garbage Collection**.

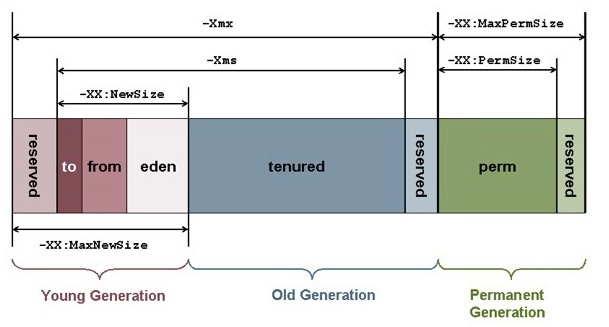
## Important points about Garbage Collection in Java

This article is in continuation of my previous articles How Classpath works in Java and How to write Equals method in Java and before moving ahead let's recall few important points about garbage collection in Java.

1) Objects are created on the heap in Java irrespective of their scope e.g. local or member variable. while it's worth noting that class variables or static members are created in method area of [Java memory space](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html) and both heap and method area is shared between different thread.  
  
2) Garbage collection is a mechanism provided by Java Virtual Machine to reclaim heap space from objects which are eligible for Garbage collection.  
  
3) Garbage collection relieves Java programmer from memory management which is an essential part of C++ programming and gives more time to focus on business logic.  
  
4) Garbage Collection in Java is carried by a daemon thread called Garbage Collector.  
  
5) Before removing an object from memory garbage collection thread invokes [finalize() method](http://javarevisited.blogspot.com/2012/03/finalize-method-in-java-tutorial.html) of that object and gives an opportunity to perform any sort of cleanup required.  
  
6) You as Java programmer can not force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.  
  
7) There are methods like System.gc() and Runtime.gc() which is used to send request of Garbage collection to JVM but it’s not guaranteed that garbage collection will happen.  
  
8) If there is no memory space for creating a new object in Heap Java Virtual Machine throws OutOfMemoryError or [java.lang.OutOfMemoryError heap space](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)  
  
9) J2SE 5(Java 2 Standard Edition) adds a new feature called Ergonomics goal of ergonomics is to provide good performance from the JVM with a minimum of command line tuning.

## When an Object becomes Eligible for Garbage Collection

## An object becomes eligible for Garbage collection or GC if it's not reachable from any live threads or by any static references. In other words, you can say that an object becomes eligible for garbage collection if its all references are null. Cyclic dependencies are not counted as the reference so if object A has a reference to object B and object B has a reference to Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection. Generally, an object becomes eligible for garbage collection in Java on following cases: 1) All references to that object explicitly set to null e.g. object = null 2) The object is created inside a block and reference goes out scope once control exit that block. 3) Parent object set to null if an object holds the reference to another object and when you set container object's reference null, child or contained object automatically becomes eligible for garbage collection. 4) If an object has only [lived weak references](http://javarevisited.blogspot.sg/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html) via WeakHashMap it will be eligible for garbage collection. Heap Generations for Garbage Collection in Java

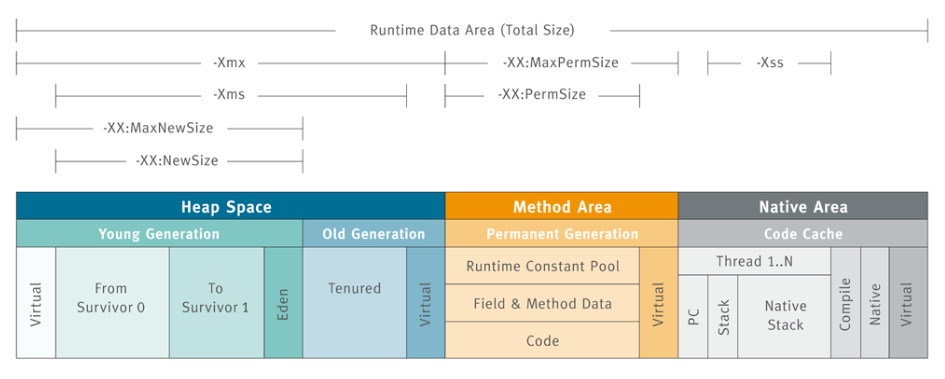
Java objects are created in Heap and Heap is divided into three parts or generations for the sake of garbage collection in Java, these are called as Young generation, Tenured or Old Generation and Perm Area of the heap.  New Generation is further divided into three parts known as Eden space, Survivor 1 and Survivor 2 space. When an object first created in heap its gets created in new generation inside Eden space and after subsequent minor garbage collection if an object survives its gets moved to survivor 1 and then survivor 2 before major garbage collection moved that object to old or tenured generation.  
  


### Permanent generation of Heap or Perm Area of Heap is somewhat special and it is used to store Metadata related to classes and method in JVM, it also hosts String pool provided by JVM as discussed in my string tutorial [why String is immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html). There are many opinions around whether garbage collection in Java happens in perm area of Java heap or not, as per my knowledge this is something which is JVM dependent and happens at least in Sun's implementation of JVM. You can also try this by just creating millions of String and watching for Garbage collection or OutOfMemoryError. Types of Garbage Collector in Java

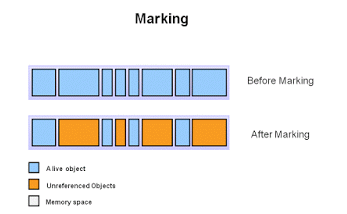
Java Runtime (J2SE 5) provides various types of Garbage collection in Java which you can choose based on your application's performance requirement. Java 5 adds three additional garbage collectors except serial garbage collector. Each is generational garbage collector which has been implemented to increase the throughput of the application or to reduce garbage collection pause times.  
  
1) Throughput Garbage Collector: This garbage collector in Java uses a parallel version of the young generation collector. It is used if the -XX:+UseParallelGC option is passed to the runtime via [JVM command line options](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html) . The tenured generation collector is same as the serial collector.

2) Concurrent low pause Collector: This Collector is used if the -Xingc or -XX:+UseConcMarkSweepGC is passed on the command line. This is also referred as Concurrent Mark Sweep Garbage collector. The concurrent collector is used to collect the tenured generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection. A parallel version of the young generation copying collector is used with the concurrent collector. Concurrent Mark Sweep Garbage collector is most widely used garbage collector in java and it uses an algorithm to first mark object which needs to collect when garbage collection triggers.  
  
3) The Incremental (Sometimes called train) low pause collector: This collector is used only if -XX:+UseTrainGC is passed on the command line. This garbage collector has not changed since the java 1.4.2 and is currently not under active development. It will not be supported in future releases so avoid using this and please see 1.4.2 GC Tuning document for information on this collector.  
  
An important point to note is that -XX:+UseParallelGC should not be used with -XX:+UseConcMarkSweepGC. The argument passing in the J2SE platform starting with version 1.4.2 should only allow the legal combination of command line options for garbage collector but earlier releases may not find or detect all illegal combination and the results for illegal combination are unpredictable. It’s not recommended to use this garbage collector in java.

## JVM Parameters for Garbage Collection in Java

Garbage collection tuning is a long exercise and requires a lot of profiling of application and patience to get it right. While working with High volume low latency Electronic trading system I have worked with some of the project where we need to increase the performance of Java application by profiling and finding what causing full GC and I found that Garbage collection tuning largely depends on application profile, what kind of object application has and what are their average lifetime etc.  
  
For example, if an application has too many short lived object then making Eden space wide enough or larger will reduce the number of minor collections. you can also control the size of both young and Tenured generation using JVM parameters for example setting -XX:NewRatio=3 means that the ratio of the young and tenured generation is 1:3 , you got to be careful on sizing this generation.  
  
  
As making young generation larger will reduce the size of the tenured generation which will force the Major collection to occur more frequently which pause application thread during that duration results in degraded or reduced throughput. The parameters NewSize and MaxNewSize are used to specify the young generation size from below and above. Setting these equal to one another fixes the young generation.  
  
In my opinion, before doing garbage collection tuning detailed understanding of how garbage collection works in Java is a must and I would recommend reading Garbage collection document provided by Sun Microsystems for detail knowledge of garbage collection in Java. Also to get a full list of JVM parameters for a particular Java Virtual machine please refer official documents on garbage collection in Java. I found this link quite helpful though http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html  
  


### Full GC and Concurrent Garbage Collection in Java

The concurrent garbage collector in java uses a single garbage collector thread that runs concurrently with the application threads with the goal of completing the collection of the tenured generation before it becomes full. In normal operation, the [concurrent garbage collector](http://javarevisited.blogspot.com/2012/10/10-garbage-collection-interview-question-answer.html) is able to do most of its work with the application threads still running, so only brief pauses are seen by the application threads.  
  
As a fallback, if the concurrent garbage collector is unable to finish before the tenured generation fills up, the application is paused and the collection is completed with all the application threads stopped. Such Collections with the application stopped are referred as full garbage collections or full GC and are a sign that some adjustments need to be made to the concurrent collection parameters.  
  


### Always try to avoid or minimize full garbage collection or Full GC because it affects the performance of Java application. When you work in finance domain for an electronic trading platform and on a high volume low latency systems performance of Java application becomes extremely critical you definitely like to avoid full GC during the trading hours. Summary on Garbage collection in Java

## 1) Java Heap is divided into three generation for the sake of garbage collection. These are a young generation, tenured or old generation, and Perm area. 2) New objects are created by young generation and subsequently moved to the old generation. 3) String pool is created in [PermGen area of Heap](http://javarevisited.blogspot.com/2012/01/tomcat-javalangoutofmemoryerror-permgen.html#uds-search-results), garbage collection can occur in perm space but depends upon JVM to JVM. By the way from JDK 1.7 update, String pool is moved to heap area where objects are created. 4) Minor garbage collection is used to move an object from Eden space to survivor 1 and survivor 2 space and major collection is used to move an object from young to tenured generation. 5) Whenever Major garbage collection occurs application threads stop during that period which will reduce application’s performance and throughput. 6) There are few performance improvements has been applied in garbage collection in java 6 and we usually use JRE 1.6.20 for running our application. 7) JVM command line options –Xmx and -Xms is used to setup starting and max size for Java Heap. The ideal ratio of this parameter is either 1:1 or 1:1.5 based on my experience, for example, you can have either both –Xmx and –Xms as 1GB or –Xms 1.2 GB and 1.8 GB. 8) There is no manual way of doing garbage collection in Java, but you can use various reference classes e.g. [WeakReference or SoftReference](http://javarevisited.blogspot.com/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html) to assist garbage collector.  Java Garbage Collection GC Initiation

Being an automatic process, programmers need not initiate the garbage collection process explicitly in the code. System.gc() and Runtime.gc() are hooks to request the JVM to initiate the garbage collection process.

Though this request mechanism provides an opportunity for the programmer to initiate the process but the onus is on the JVM. It can choose to reject the request and so it is not guaranteed that these calls will do the garbage collection. This decision is taken by the JVM based on the eden space availability in heap memory. The JVM specification leaves this choice to the implementation and so these details are implementation specific.

Undoubtedly we know that the garbage collection process cannot be forced. I just found out a scenario when invoking System.gc() makes sense. Just go through this article to know about this corner case when [System.gc() invocation is applicable](http://javapapers.com/core-java/system-gc-invocation-a-suitable-scenario/).

## Java Garbage Collection Process

Garbage collection is the process of reclaiming the unused memory space and making it available for the future instances.



**Eden Space:** When an instance is created, it is first stored in the eden space in young generation of heap memory area.

NOTE: If you couldn’t understand any of these words, I recommend you to go through the [garbage collection introduction tutorial](http://javapapers.com/java/java-garbage-collection-introduction/) which goes through the memory mode, JVM architecture and these terminologies in detail.

**Survivor Space (S0 and S1):** As part of the minor garbage collection cycle, objects that are live (which is still referenced) are moved to survivor space S0 from eden space. Similarly the garbage collector scans S0 and moves the live instances to S1.

Instances that are not live (dereferenced) are marked for garbage collection. Depending on the garbage collector (there are four types of garbage collectors available and we will see about them in the next tutorial) chosen either the marked instances will be removed from memory on the go or the eviction process will be done in a separate process.

**Old Generation:** Old or tenured generation is the second logical part of the heap memory. When the garbage collector does the minor GC cycle, instances that are still live in the S1 survivor space will be promoted to the old generation. Objects that are dereferenced in the S1 space is marked for eviction.

**Major GC:** Old generation is the last phase in the instance life cycle with respect to the Java garbage collection process. Major GC is the garbage collection process that scans the old generation part of the heap memory. If instances are dereferenced, then they are marked for eviction and if not they just continue to stay in the old generation.

**Memory Fragmentation:** Once the instances are deleted from the heap memory the location becomes empty and becomes available for future allocation of live instances. These empty spaces will be fragmented across the memory area. For quicker allocation of the instance it should be defragmented. Based on the choice of the garbage collector, the reclaimed memory area will either be compacted on the go or will be done in a separate pass of the GC.

## Finalization of Instances in Garbage Collection

Just before evicting an instance and reclaiming the memory space, the Java garbage collector invokes the finalize() method of the respective instance so that the instance will get a chance to free up any resources held by it. Though there is a guarantee that the finalize() will be invoked before reclaiming the memory space, there is no order or time specified. The order between multiple instances cannot be predetermined, they can even happen in parallel. Programs should not pre-mediate an order between instances and reclaim resources using the finalize() method.

* Any uncaught exception thrown during finalize process is ignored silently and the finalization of that instance is cancelled.
* JVM specification does not discuss about garbage collection with respect to weak references and claims explicitly about it. Details are left to the implementer.
* Garbage collection is done by a daemon thread.

## When an object becomes eligible for garbage collection?

* Any instances that cannot be reached by a live thread.
* Circularly referenced instances that cannot be reached by any other instances.

There are [different types of references in Java](http://javapapers.com/core-java/java-weak-reference/). Instances eligibility for garbage collection depends on the type of reference it has.

|  |  |
| --- | --- |
| **Reference** | **Garbage Collection** |
| Strong Reference | Not eligible for garbage collection |
| Soft Reference | Garbage collection possible but will be done as a last option |
| Weak Reference | Eligible for Garbage Collection |
| Phantom Reference | Eligible for Garbage Collection |

During compilation process as an optimization technique the Java compiler can choose to assign null value to an instance, so that it marks that instance can be evicted.

class Animal {

public static void main(String[] args) {

Animal lion = new Animal();

System.out.println("Main is completed.");

}

protected void finalize() {

System.out.println("Rest in Peace!");

}

}

In the above class, lion instance is never uses beyond the instantiation line. So the Java compiler as an optimzation measure can assign lion = null just after the instantiation line. So, even before SOP’s output, the finalizer can print ‘Rest in Peace!’. We cannot prove this deterministically as it depends on the JVM implementation and memory used at runtime. But there is one learning, compiler can choose to free instances earlier in a program if it sees that it is referenced no more in the future.

* One more excellent example for when an instance can become eligible for garbage collection. All the properties of an instance can be stored in the register and thereafter the registers will be accessed to read the values. There is no case in future that the values will be written back to the instance. Though the values can be used in future, still this instance can be marked eligible for garbage collection. Classic isn’t it?
* It can get as simple as an instance is eligible for garbage collection when null is assigned to it or it can get complex as the above point. These are choices made by the JVM implementer. Objective is to leave as small footprint as possible, improves the responsiveness and increase the throughput. In order to achieve this the JVM implementer can choose a better scheme or algorithm to reclaim the memory space during garbage collection.
* When the finalize() is invoked, the JVM releases all synchronize locks on that thread.

### Example Program for GC Scope

Class GCScope {

GCScope t;

static int i = 1;

public static void main(String args[]) {

GCScope t1 = new GCScope();

GCScope t2 = new GCScope();

GCScope t3 = new GCScope();

// No Object Is Eligible for GC

t1.t = t2; // No Object Is Eligible for GC

t2.t = t3; // No Object Is Eligible for GC

t3.t = t1; // No Object Is Eligible for GC

t1 = null;

// No Object Is Eligible for GC (t3.t still has a reference to t1)

t2 = null;

// No Object Is Eligible for GC (t3.t.t still has a reference to t2)

t3 = null;

// All the 3 Object Is Eligible for GC (None of them have a reference.

// only the variable t of the objects are referring each other in a

// rounded fashion forming the Island of objects with out any external

// reference)

}

protected void finalize() {

System.out.println("Garbage collected from object" + i);

i++;

}

### Example Program for GC OutOfMemoryError

Garbage collection does not guarantee safety from out of memory issues. Mindless code will lead us to OutOfMemoryError.

import java.util.LinkedList;

import java.util.List;

public class GC {

public static void main(String[] main) {

List l = new LinkedList();

// Enter infinite loop which will add a String to the list: l on each

// iteration.

do {

l.add(new String("Hello, World"));

} while (true);

}

}

Output:

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

at java.util.LinkedList.linkLast(LinkedList.java:142)

at java.util.LinkedList.add(LinkedList.java:338)

at com.javapapers.java.GCScope.main(GCScope.java:12)

## Garbage collection and the Java platform memory model

When you specify the startup option -Xmx on the command line of your Java application (for instance: java -Xmx:2g MyApp) memory is assigned to a Java process. This memory is referred to as the Java heap (or just heap). This is the dedicated memory address space where all objects created by your Java program (or sometimes the JVM) will be allocated. As your Java program keeps running and allocating new objects, the Java heap (meaning that address space) will fill up.

Eventually, the Java heap will be full, which means that an allocating thread is unable to find a large-enough consecutive section of free memory for the object it wants to allocate. At that point, the JVM determines that a garbage collection needs to happen and it notifies the garbage collector. A garbage collection can also be triggered when a Java program calls System.gc(). Using System.gc() does not guarantee a garbage collection. Before any garbage collection can start, a GC mechanism will first determine whether it is safe to start it. It is safe to start a garbage collection when all of the application's active threads are at a safe point to allow for it, e.g. simply explained it would be bad to start garbage collecting in the middle of an ongoing object allocation, or in the middle of executing a sequence of optimized CPU instructions (see my previous article on compilers), as you might lose context and thereby mess up end results.

A garbage collector should never reclaim an actively referenced object; to do so would break the [Java virtual machine specification](http://docs.oracle.com/javase/specs/jvms/se7/html/index.html). A garbage collector is also not required to immediately collect dead objects. Dead objects are eventually collected during subsequent garbage collection cycles. While there are many ways to implement garbage collection, these two assumptions are true for all varieties. The real challenge of garbage collection is to identify everything that is live (still referenced) and reclaim any unreferenced memory, but do so without impacting running applications any more than necessary. A garbage collector thus has two mandates:

1. To quickly free unreferenced memory in order to satisfy an application's allocation rate so that it doesn't run out of memory.
2. To reclaim memory while minimally impacting the performance (e.g., latency and throughput) of a running application.

### Reference counting collectors

Reference counting collectors keep track of how many references are pointing to each Java object. Once the count for an object becomes zero, the memory can be immediately reclaimed. This immediate access to reclaimed memory is the major advantage of the reference-counting approach to garbage collection. There is very little overhead when it comes to holding on to un-referenced memory. Keeping all reference counts up to date can be quite costly, however.

The main difficulty with reference counting collectors is keeping the reference counts accurate. Another well-known challenge is the complexity associated with handling circular structures. If two objects reference each other and no live object refers to them, their memory will never be released. Both objects will forever remain with a non-zero count. Reclaiming memory associated with circular structures requires major analysis, which brings costly overhead to the algorithm, and hence to the application.

### Tracing collectors

Tracing collectors are based on the assumption that all live objects can be found by iteratively tracing all references and subsequent references from an initial set of known to be live objects. The initial set of live objects (called root objects or just roots for short) are located by analyzing the registers, global fields, and stack frames at the moment when a garbage collection is triggered. After an initial live set has been identified, the tracing collector follows references from these objects and queues them up to be marked as live and subsequently have their references traced. Marking all found referenced objects live means that the known live set increases over time. This process continues until all referenced (and hence all live) objects are found and marked. Once the tracing collector has found all live objects, it will reclaim the remaining memory.

Tracing collectors differ from reference-counting collectors in that they can handle circular structures. The catch with most tracing collectors is the marking phase, which entails a wait before being able to reclaim non-referenced memory.

Tracing collectors are most commonly used for memory management in dynamic languages; they are by far the most common for the Java language and have been commercially proven in production environments for many years. I'll focus on tracing collectors for the remainder of this article, starting with some of the algorithms that implement this approach to garbage collection.

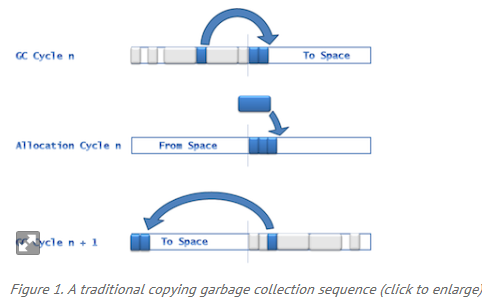
## Tracing collector algorithms

Copying and mark-and-sweep garbage collection are not new, but they're still the two most common algorithms that implement tracing garbage collection today.

### Copying collectors

Traditional copying collectors use a from-space and a to-space -- that is, two separately defined address spaces of the heap. At the point of garbage collection, the live objects within the area defined as from-space are copied into the next available space within the area defined as to-space. When all the live objects within the from-space are moved out, the entire from-space can be reclaimed. When allocation begins again it starts from the first free location in the to-space.

In older implementations of this algorithm the from-space and to-space switch places, meaning that when the to-space is full, garbage collection is triggered again and the to-space becomes the from-space, as shown in Figure 1.



implementations of the copying algorithm allow for arbitrary address spaces within the heap to be assigned as to-space and from-space. In these cases they do not necessarily have to switch location with each other; rather, each becomes another address space within the heap.

One advantage of copying collectors is that objects are allocated together tightly in the to-space, completely eliminating fragmentation. Fragmentation is a common issue that other garbage collection algorithms struggle with; something I'll discuss later in this article.

#### Downsides of copying collectors

Copying collectors are usually stop-the-world collectors, meaning that no application work can be executed for as long as the garbage collection is in cycle. In a stop-the-world implementation, the larger the area you need to copy, the higher the impact on your application performance will be. This is a disadvantage for applications that are sensitive to response time. With a copying collector you also need to consider the worst-case scenario, when everything is live in the from-space. You always have to leave enough headroom for live objects to be moved, which means the to-space must be large enough to host everything in the from-space. The copying algorithm is slightly memory inefficient due to this constraint.

### Mark-and-sweep collectors

Most commercial JVMs deployed in enterprise production environments run mark-and-sweep (or marking) collectors, which do not have the performance impact that copying collectors do. Some of the most famous marking collectors are CMS, G1, GenPar, and DeterministicGC (see [Resources](http://www.javaworld.com/#resources)).

A mark-and-sweep collector traces references and marks each found object with a "live" bit. Usually a set bit corresponds to an address or in some cases a set of addresses on the heap. The live bit can, for instance, be stored as a bit in the object header, a bit vector, or a bit map.

After everything has been marked live, the sweep phase will kick in. If a collector has a sweep phase it basically includes some mechanism for traversing the heap again (not just the live set but the entire heap length) to locate all the non-marked chunks of consecutive memory address spaces. Unmarked memory is free and reclaimable. The collector then links together these unmarked chunks into organized free lists. There can be various free lists in a garbage collector -- usually organized by chunk sizes. Some JVMs (such as JRockit Real Time) implement collectors with heuristics that dynamically size-range lists based on application profiling data and object-size statistics.

When the sweep phase is complete allocation will begin again. New allocation areas are allocated from the free lists and memory chunks could be matched to object sizes, object size averages per thread ID, or the application-tuned TLAB sizes. Fitting free space more closely to the size of what your application is trying to allocate optimizes memory and could help reduce fragmentation.

#### Downsides of mark-and-sweep collectors

The mark phase is dependent on the amount of live data on your heap, while the sweep phase is dependent on the heap size. Since you have to wait until both the mark and sweep phases are complete to reclaim memory, this algorithm causes pause-time challenges for larger heaps and larger live data sets.

One way that you can help heavily memory-consuming applications is to use GC-tuning options that accommodate various application scenarios and needs. Tuning can, in many cases, help at least postpone either of these phases from becoming a risk to your application or service-level agreements (SLAs). (An SLA specifies that the application will meet certain application response times -- i.e., latency.) Tuning for every load change and application modification is a repetitive task, however, as the tuning is only valid for a specific workload and allocation rate.

## Implementations of mark-and-sweep

There are at least two commercially available and proven approaches for implementing mark-and-sweep collection. One is the parallel approach and the other is the concurrent (or mostly concurrent) approach.

### Parallel collectors

Parallel collection means that resources assigned to the process are used in parallel for the purpose of garbage collection. Most commercially implemented parallel collectors are monolithic stop-the-world collectors -- all application threads are stopped until the entire garbage collection cycle is complete. Stopping all threads allows all resources to be efficiently used in parallel to finish the garbage collection through the mark and sweep phases. This leads to a very high level of efficiency, usually resulting in high scores on throughput benchmarks such as [SPECjbb](http://www.spec.org/jbb2005/). If throughput is essential for your application, the parallel approach is an excellent choice.

http://www.javaworld.com/article/2078645/java-se/jvm-performance-optimization-part-3-garbage-collection.html?page=3