**[What is EnumMap in Java – Example Tutorial](http://javarevisited.blogspot.in/2012/09/what-is-enummap-in-java-example-tutorial.html" \o "What is EnumMap in Java – Example Tutorial)**

**What is EnumMap in Java**

EnumMap in Java is added on JDK  5 release along with other important features like  [Autoboxing](http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html), [varargs](http://javarevisited.blogspot.sg/2011/09/variable-argument-in-java5-varargs.html) and [Generics](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html). EnumMap is specialized Map implementation designed and optimized for using [Java Enum](http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html) as key. Since enum can represent a type (like class or interface)  in Java and it can also override [equals() and hashCode()](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) , It can be used inside [HashMap](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) or any other collection but using  EnumMapbrings implementation specific benefits which is done for enum keys, In short EnumMap is optimized Map implementation exclusively for enum keys . As per javadoc Enum is implemented using Arrays and common operations result in constant time. So if you are thinking of an high performance Map, EnumMap could be decent choice for enumeration data. We have already seen many examples of Java enum in our article [10 Examples of enum in Java](http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html)  and using [Enum as thread-safe Singleton](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html). In this Java tutorial we will see simple examples of using EnumMap in Java. On related note *Difference between EnumMap and HashMap* is also getting popular as one of advanced [Java collection interviews questions](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html) and most of Java IDE like Eclipse and Netbeans will also suggest to use EnumMap if you use Enum keys along with HashMap as part of there code improvement suggestion. 

**Important points of EnumMap in Java:**

Here is few important points to remember about EnumMap in Java which is also useful while using EnumMap in code to avoid any compile time or logical errors :

1. All keys used in EnumMap must be  from same [Enum type](http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html) which is specified while creating EnumMap in Java. For example if you can not use different enum instances from two different enum.

2. EnumMap is [ordered collection](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) and they are maintained in the natural order of their keys( natural order of keys means  the order on which enum constant are declared inside enum type ). you can verify this while Iterating over an EnumMap in Java.

3. Iterators of EnumMap are [fail-fast Iterator](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) , much like of [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) and doesn't throw ConcurrentModificationException and may not show effect of any modification on EnumMap during Iteration process.

4. You can not insert null keys inside EnumMap in Java.  EnumMap doesn't allow null key and throw [NullPointerException](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html), at same time null values are permitted.

5. EnumMap is not synchronized and it has to be [synchronized](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) manually before using it in a concurrent or multi-threaded environment. like synchronized Map in Java  you can also make EnumMap synchronized by using Collections.synchronizedMap() method and as per javadoc this should be done while creating EnumMap in java to avoid accidental non synchronized access.

6. EnumMap is likely give better performance than HashMap in Java. So prefer EnumMap if you are going to use enum keys.

How to use EnumMap in Java – EnumMap Example

In this section we will see *How to use EnumMap in Java* with simple examples like creating EnumMap, putting objects into EnumMap, getting objects from EnumMap,  finding size of EnumMap, [Iterating](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) over EnumMap, printing EnumMap in console , checking if EnumMap contains a particular key and value or not etc. All of these operations are similar to other Map implementation like [HashMap](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) and doesn’t require special knowledge but It’s good to remember all points specific to EnumMap as discussed in previous section to avoid any error.

**import** java.util.EnumMap;  
**import** java.util.Iterator;

/\*\*  
 \* **Java program to demonstrate How to use EnumMap in Java**  
 \* If Key Object is Enum than it’s best to EnumMap to get better performance.

 \* Most of IDE like Netbeans and Eclipse suggest you to use EnumMap instead of HashMap

 \* or any other Map implementation when key object is Enum.

 \*

 \* @author Javarevisited  
 \*/  
  
**public** **class** EnumMapExample{  
    
    **public** **enum** STATE{  
        NEW, RUNNING, WAITING, FINISHED;  
    }  
  
    **public** **static** **void** main(**String** args[]) {  
        
        *// Java EnumMap Example 1: creating EnumMap in java with key as enum type STATE*  
        **EnumMap**<STATE, **String**> stateMap = **new** **EnumMap**<STATE, **String**>(STATE.**class**);  
        
        *// Java EnumMap Example 2:*  
        *//putting values inside EnumMap in Java*  
        *//we are inserting Enum keys on different order than their natural order*  
        stateMap.put(STATE.RUNNING, "Program is running");  
        stateMap.put(STATE.WAITING, "Program is waiting");  
        stateMap.put(STATE.NEW, "Program has just created");  
        stateMap.put(STATE.FINISHED, "Program has finished");  
        
        *// Java EnumMap Example 3:*  
        *//printing size of EnumMap in java*  
        **System**.out.println("Size of EnumMap in java: " + stateMap.size());  
        
        *// Java EnumMap Example 5:*  
        *//printing Java EnumMap , should print EnumMap in natural order*  
        *//of enum keys (order on which they are declared)*  
        **System**.out.println("EnumMap: " + stateMap);  
        
        *// Java EnumMap Example 5:*  
        *//retrieving value from EnumMap in java*  
        **System**.out.println("EnumMap key : " + STATE.NEW +" value: " + stateMap.get(STATE.NEW));  
        
        *// Java EnumMap Example 6:*  
        *//Iterating over Java EnumMap*  
        **Iterator**<STATE> enumKeySet = stateMap.keySet().iterator();  
        while(enumKeySet.hasNext()){  
            STATE currentState = enumKeySet.next();  
            **System**.out.println("key : " + currentState + " value : " + stateMap.get(currentState));  
        }  
        
        *//Java EnumMap Example 7: checking if EnumMap contains a particular key*  
        **System**.out.println("Does stateMap has :" + STATE.NEW + " : "   
                            +  stateMap.containsKey(STATE.NEW));  
        
        *//Java EnumMap Example 8: checking if EnumMap contains a particular value*  
        **System**.out.println("Does stateMap has :" + STATE.NEW + " : " + stateMap.containsValue(**null**));  
  
    }  
    
}  
  
Output:  
Size of **EnumMap** in java: 4  
**EnumMap**: {NEW=Program has just created, RUNNING=Program is running, WAITING=Program is waiting, FINISHED=Program has finished}  
  
**EnumMap** key : NEW value: Program has just created  
key : NEW value : Program has just created  
key : RUNNING value : Program is running  
key : WAITING value : Program is waiting  
key : FINISHED value : Program has finished  
Does stateMap has :NEW : **true**  
Does stateMap has :NEW : **false**

In summary if you are using enum keys or can use enum keys *prefer EnumMap over HashMap* or any other Map implementation because EnumMap is specialized Map implementation for enum and provides better performance than general map. In this Java tutorial we have seen **What is EnumMap in Java**, important points about EnumMap in Java and How to use EnumMap with some how to type of examples

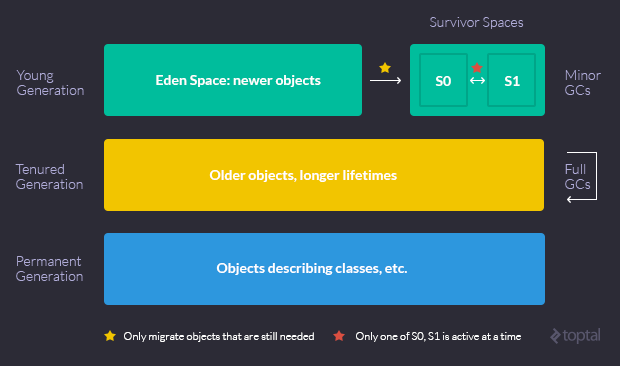
## How Does Garbage Collection Work in the JVM?

Before we start our analysis of an application with a memory leak issue, let’s first look at how garbage collection works in the JVM.

The JVM uses a form of garbage collector called a **tracing collector**, which essentially operates by pausing the world around it, marking all root objects (objects referenced directly by running threads), and following their references, marking each object it sees along the way.

Java implements something called a **generational** garbage collector based upon the generational hypothesis assumption, which states that **the majority of objects that are created are quickly discarded**, and **objects that are not quickly collected are likely to be around for a while**.

Based on this assumption, [Java partitions objects into multiple generations](http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html#1.1.%20Generations%7Coutline). Here’s a visual interpretation:



* **Young Generation** - This is where objects start out. It has two sub-generations:
  + **Eden Space** - Objects start out here. Most objects are created and destroyed in the Eden Space. Here, the GC does Minors GCs, which are optimized garbage collections. When a Minor GC is performed, any references to objects that are still needed are migrated to one of the survivors spaces (S0 or S1).
  + **Survivor Space (S0 and S1)** - Objects that survive Eden end up here. There are two of these, and only one is in use at any given time (unless we have a serious memory leak). One is designated as empty, and the other as live, alternating with every GC cycle.
* **Tenured Generation** - Also known as the old generation (old space in Fig. 2), this space holds older objects with longer lifetimes (moved over from the survivor spaces, if they live for long enough). When this space is filled up, the GC does a **Full GC**, which costs more in terms of performance. If this space grows without bound, the JVM will throw an OutOfMemoryError - Java heap space.
* **Permanent Generation** - A third generation closely related to the tenured generation, the permanent generation is special because it holds data required by the virtual machine to describe objects that do not have an equivalence at the Java language level. For example, objects describing classes and methods are stored in the permanent generation.

Java is smart enough to apply different garbage collection methods to each generation. The young generation is handled using a **tracing, copying collector** called the **Parallel New Collector**. This collector stops the world, but because the young generation is generally small, the pause is short.

For more information about the JVM generations and how them work in more detail visit the [Memory Management in the Java HotSpot™ Virtual Machine](http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf) documentation.

## Detecting a Memory Leak

To find and eliminate a memory leak, you need the proper tools. It’s time to detect and remove such a leak using the [Java VisualVM](http://visualvm.java.net/intro.html).

### Remotely Profiling the Heap with Java VisualVM

VisualVM is a tool that provides a visual interface for viewing detailed information about Java technology-based applications while they are running.

With VisualVM, you can view data related to local applications and those running on remote hosts. You can also capture data about JVM software instances and save the data to your local system.

In order to benefit from all of Java VisualVM’s features, you should run the Java Platform, Standard Edition (Java SE) version 6 or above.

# Visualizing Garbage Collection Algorithms

Most developers take automatic garbage collection for granted. It’s just another amazing feature provided by our language run-times to make our jobs easier.

But if you try to peek inside a modern garbage collector, it’s very difficult to see how they actually work. There are thousands of implementation details that will confuse you unless you already have a good understanding of what it’s trying to do and how they can go fantastically wrong.

I’ve built a toy with five different garbage collection algorithms. Small animations were created from the run-time behavior. You can find larger animations and the code to create them at [github.com/kenfox/gc-viz](https://github.com/kenfox/gc-viz). It surprised me how much a simple animation reveals about these important algorithms.

## Cleanup At The End: aka No GC

The simplest possible way of cleaning up garbage is to just wait until a task is done and dispose of everything at once. This is a surprisingly useful technique, especially if you have a way of breaking up a task into pieces. The Apache web server, for example, creates a small pool of memory per request and throws the entire pool away when the request completes.

The small animation to the right represents a running program. The entire image represents the program’s memory. Memory starts out colored black, which means it isn’t used. Areas that flash bright green or yellow are memory reads or writes. The color decays over time so you can see how memory was used, but also see current activity. If you watch carefully, you can see patterns emerge where the program begins to ignore some memory. Those areas have become garbage — they are not used and not reachable by the program. Everything else that isn’t garbage is “live”.

The program easily fits in memory without needing to worry about cleaning up garbage while the program is running. I’m going to stick with this simple program for the rest of the examples.

## Reference Counting Collector

Another simple solution is to keep a count of how many times you are using a resource (an object in memory, in this case) and dispose of it when the count drops to zero. This is the most common technique that developers use when they add garbage collection to an existing system —it’s the only garbage collector that easily integrates with other resource managers and existing code bases. Apple learned this lesson after releasing a mark-sweep collector for Objective-C. It caused so many problems that they retracted the feature and replaced it with an automated reference counting collector that works much better with existing code.

The animation shows the same program as above, but this time it will try to dispose of garbage by keeping a reference count on each object in memory. The red flashes indicate reference counting activity. A very useful property of reference counting is that garbage is detected as soon as possible — you can sometimes see a flash of red immediately followed by the area turning black.

Unfortunately reference counting has a lot of problems. Worst of all, it can’t handle cyclic structures. These are very common — anything with a parent or reverse reference creates a cycle which will leak memory. Reference counting also has very high overhead — you can see in the animation that red flashes are constantly happening even when memory use is not increasing. Arithmetic is fast on a modern CPU, but memory is slow, and the counters are being loaded and saved to memory often. All these counter updates also make it difficult to have read-only or thread-safe data.

Reference counting is an amortized algorithm (the overhead is spread over the run-time of the program), but it’s an accidentally amortized algorithm that can’t guarantee response times. For example, say a program is working with a very large tree structure. The last piece of the program that uses the tree will trigger the disposal of the entire tree, which Murphy will guarantee happens when the user least desires the delay. None of the other algorithms here are amortized either though, so accidentally amortized may be a feature depending on your data. (All of these algorithms do have concurrent or partially-concurrent variations, but those are beyond the capabilities of my toy program to demonstrate.)

## Mark-Sweep Collector

Mark-sweep eliminates some of the problems of reference count. It can easily handle cyclic structures and it has lower overhead since it doesn’t need to maintain counts.

It gives up being able to detect garbage immediately. You can see that in the animation where there’s a period of activity without any red flashes, then suddenly a bunch of red flashes indicate where it is marking live objects. After marking is finished, it sweeps over all of memory and disposes of garbage. You can see that in the animation too — several areas turn black all at once instead of more spread out over time in the reference counting approach.

Mark-sweep requires more implementation consistency than reference counting and is more difficult to retrofit into existing systems. The mark phase requires being able to traverse all live data, even data encapsulated within an object. If an object doesn’t provide traversal, it’s probably too risky to attempt to retrofit mark-sweep into the code. The other weakness of mark-sweep is the fact the sweep phase must sweep over all of memory to find garbage. For systems that do not generate much garbage, this is not an issue, but modern functional programming style generates enormous amounts of garbage.

## Mark-Compact Collector

One thing you may have noticed in the previous animations is that objects never move. Once an object is allocated in memory, it stays in the same place even if memory turns into a fragmented sea of islands surrounded by black. The next two algorithms change that, but with completely different approaches.

Mark-compact disposes of memory, not by just marking it free, but by moving objects down into the free space. Objects always stay in the same memory order — an object allocated before another object will always be lower in memory — but gaps caused by disposed objects will be closed up by objects moving down.

The crazy idea of moving objects means that new objects can always just be created at the end of used memory. This is called a “bump” allocator and is as cheap as stack allocation, but without the limitations of stack size. Some systems using bump allocators don’t even use call stacks for data storage, they just allocate call frames in the heap and treat them like any other object.

Another benefit, sometimes more theory than practice, is that when objects are compacted like this, programs have better memory access patterns that are friendly to modern hardware memory caches. It’s far from certain you will see this benefit, though — the memory allocators used in reference counting and mark-sweep are complex, but also very well debugged and very efficient.

Mark-compact is a complex algorithm requiring several passes over all allocated objects. In the animation you can see the red flashes of live object marking followed by lots of reads and writes as destinations are computed, objects are moved and finally references are fixed to point to where objects have moved. The main benefit of all this complexity is operating under extremely low memory overhead. Oracle’s Hotspot JVMuses several different garbage collection algorithms. The tenured object space uses mark-compact.

## Copying Collector

The last algorithm I’ve animated is the foundation of most high-performance garbage collection systems. It’s a moving collector like mark-compact, but it’s incredibly simple. It uses two memory spaces and simply copies live objects back and forth between them. In practice, there are more than two spaces and the spaces are used for different generations of objects. New objects are created in one space, get copied to another space if they survive, and finally copied to a tenured space if they are very long-lived. If you hear a garbage collector described as generational or ephemeral, it’s usually a multi-space copy collector.

Other than simplicity and flexibility, the main advantage of this algorithm is that it only spends time on live objects. There is no separate mark phase that must be later swept or compacted. Objects are immediately copied during live object traversal, and object references are patched up by following a broken-heart reference where the object used to be.

In the animation, you can see there are several collections where almost all the data is copied from one space to the other. This is a terrible situation for this algorithm and shows one of the reasons why people talk about tuning garbage collectors. If you can size your memory and tune your allocations so that most objects are dead when the collection begins, then you get a fantastic combination of safe functional programming style and high performance.

# Mark-And-Sweep (Garbage Collection Algorithm)

The mark-and-sweep algorithm is called a tracing garbage collector because is traces out the entire collection of objects that are directly or indirectly accessible by the program.

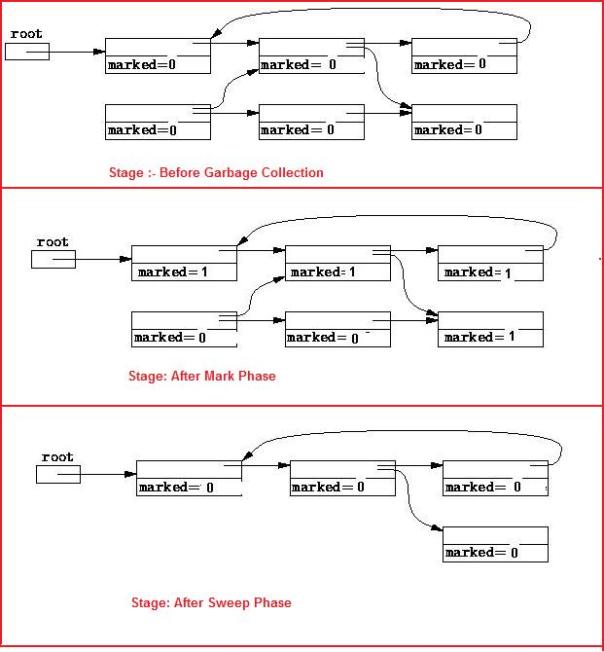
**Root:**  
The objects that a program can access directly are those objects which are referenced by local variables on the processor stack as well as by any static variables that refer to objects. In the context of garbage collection, these variables are called the roots.

**Live Objects:**  
An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object. An accessible object is said to be live . Conversely, an object which is not live is garbage.

The mark-and-sweep algorithm is divided into two phases:

**Mark phase:**  
The garbage collector traverses the graph of references from the root nodes and marks each heap object it encounters. Each object has an extra bit: the mark bit – initially the mark bit is 0. It is set to 1 for the reachable objects in the mark phase.

**Sweep phase:**

The GC scans the heap looking for objects with mark bit 0 – these objects have not been visited in the mark phase – they are garbage. Any such object is added to the free list of objects that can be reallocated. The objects with a mark bit 1 have their mark bit reset to 0.[](https://sandeepin.files.wordpress.com/2011/12/markandsweepexampleupdated11.jpg)  
The algorithm can be expressed as follows:

|  |  |
| --- | --- |
| 1  2  3 | for each root variable a      mark (a);   sweep (); |
| 1  2  3  4  5 | void mark (Object a)      if (!a.marked)          a.marked = true;          for each Object b referenced by a              mark (b); |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | void sweep ()      for each Object a in the heap          if (a.marked)              a.marked = false          else              heap.release (a); |

**Advantages**  
— It is able to reclaim garbage that contains cyclic references.  
— There is no overhead in storing and manipulating reference count fields.  
— Objects are not moved during GC – no need to update the references to objects.

**Disadvantages**  
— It may increase heap fragmentation.  
— It does work proportional to the size of the entire heap.  
— The program must be halted while garbage collection is being performed.

# How Java Garbage Collection Works?

This tutorial is to understand the basics of Java garbage collection and how it works. This is the second part in the garbage collection tutorial series. Hope you have read [introduction to Java garbage collection](http://javapapers.com/java/java-garbage-collection-introduction/), which is the first part.

Java garbage collection is an automatic process to manage the runtime memory used by programs. By doing it automatic JVM relieves the programmer of the overhead of assigning and freeing up memory resources in a program.

## Java Garbage Collection GC Initiation

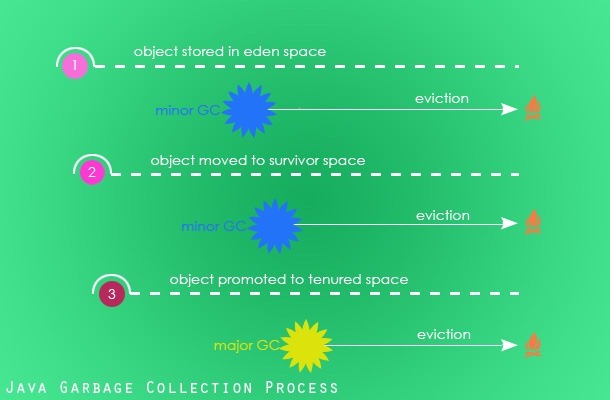
Being an automatic process, programmers need not initiate the garbage collection process explicitly in the code. System.gc() and Runtime.gc() are hooks to request the JVM to initiate the garbage collection process.

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/" \o "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)

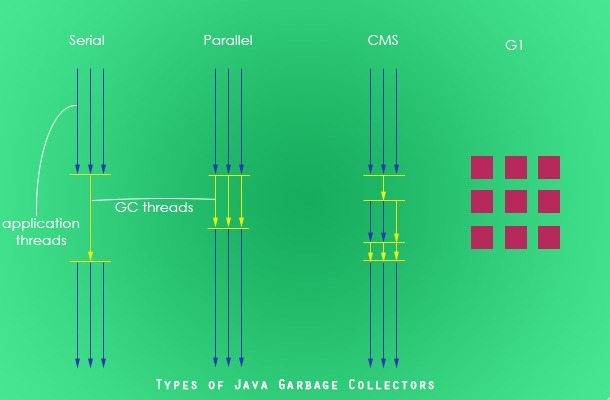
Next is the third part of the garbage collection tutorial series and we will see about the different [types of Java garbage collectors](http://javapapers.com/java/types-of-java-garbage-collectors/) available.

# Types of Java Garbage Collectors

In this tutorial we will go through the various type of Java garbage collectors available. Garbage collection is an automatic process in Java which relieves the programmer of object memory allocation and de-allocation chores. This is the third part in the garbage collection tutorial series. In the previous part 2 we saw about [how garbage collection works in Java](http://javapapers.com/java/how-java-garbage-collection-works/), it is an interesting read and I recommend you to go through it. In the part 1 [introduction to Java garbage collection](http://javapapers.com/java/java-garbage-collection-introduction/), we saw about the JVM architecture, heap memory model and surrounding Java terminologies.

Java has **four types of garbage collectors**,

1. [Serial Garbage Collector](http://javapapers.com/java/types-of-java-garbage-collectors/#serial-garbage-collector)
2. [Parallel Garbage Collector](http://javapapers.com/java/types-of-java-garbage-collectors/#parallel-garbage-collector)
3. [CMS Garbage Collector](http://javapapers.com/java/types-of-java-garbage-collectors/#cms-garbage-collector)
4. [G1 Garbage Collector](http://javapapers.com/java/types-of-java-garbage-collectors/#g1-garbage-collector)



Each of these four types has its own advantages and disadvantages. Most importantly, we the programmers can choose the type of garbage collector to be used by the JVM. We can choose them by passing the choice as JVM argument. Each of these types differ largely and can provide completely different application performance. It is critical to understand each of these types of garbage collectors and use it rightly based on the application.

## 1. Serial Garbage Collector

Serial garbage collector works by holding all the application threads. It is designed for the single-threaded environments. It uses just a single thread for garbage collection. The way it works by freezing all the application threads while doing garbage collection may not be suitable for a server environment. It is best suited for simple command-line programs.

Turn on the -XX:+UseSerialGC JVM argument to use the serial garbage collector.

## 2. Parallel Garbage Collector

Parallel garbage collector is also called as throughput collector. It is the default garbage collector of the JVM. Unlike serial garbage collector, this uses multiple threads for garbage collection. Similar to serial garbage collector this also freezes all the application threads while performing garbage collection.

## 3. CMS Garbage Collector

Concurrent Mark Sweep (CMS) garbage collector uses multiple threads to scan the heap memory to mark instances for eviction and then sweep the marked instances. CMS garbage collector holds all the application threads in the following two scenarios only,

1. while marking the referenced objects in the tenured generation space.
2. if there is a change in heap memory in parallel while doing the garbage collection.

In comparison with parallel garbage collector, CMS collector uses more CPU to ensure better application throughput. If we can allocate more CPU for better performance then CMS garbage collector is the preferred choice over the parallel collector.

Turn on the XX:+USeParNewGC JVM argument to use the CMS garbage collector.

## 4. G1 Garbage Collector

G1 garbage collector is used for large heap memory areas. It separates the heap memory into regions and does collection within them in parallel. G1 also does compacts the free heap space on the go just after reclaiming the memory. But CMS garbage collector compacts the memory on stop the world (STW) situations. G1 collector prioritizes the region based on most garbage first.

Turn on the –XX:+UseG1GC JVM argument to use the G1 garbage collector.