javapapers

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, 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

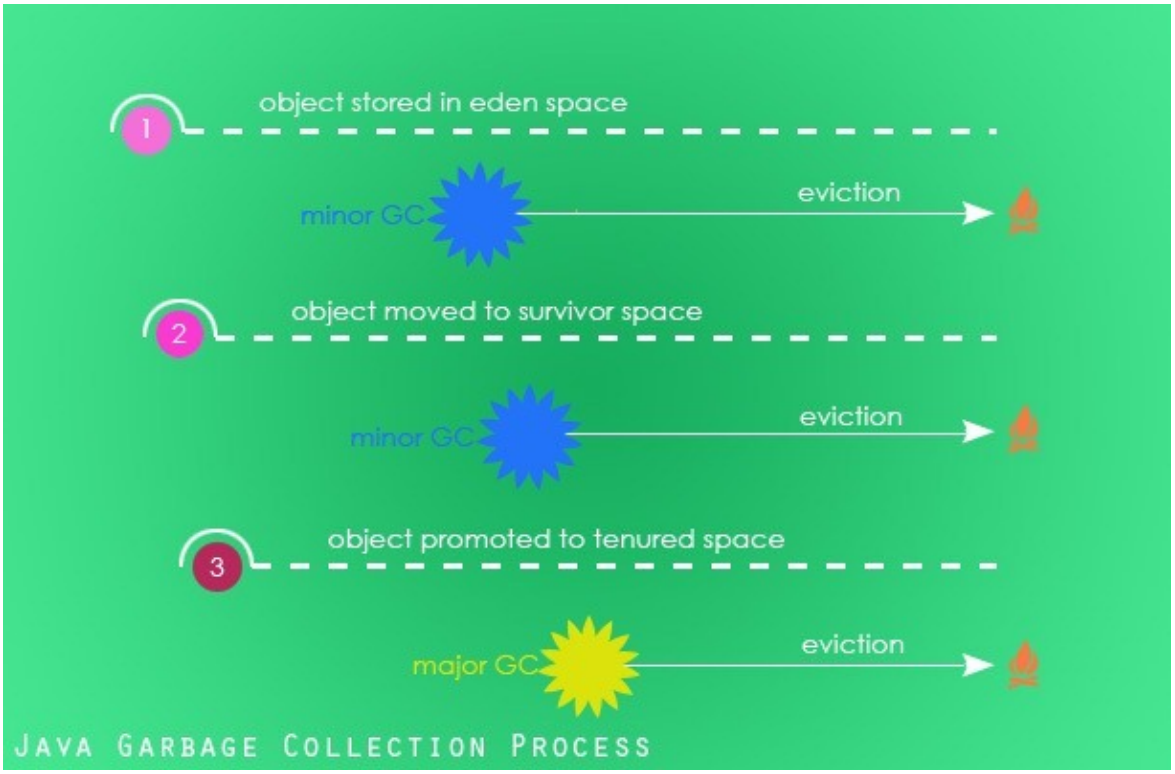
Being an automatic process, programmers need not initiate the garbage collection process explicitly in the code. `System.gc()` and `Runtime.gc()` are hooks to request the JVM to initiate the garbage collection process.

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.

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 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. 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 referen
ce to t1)

        t2 = null;
        // No Object Is Eligible for GC (t3.t.t still has a refer
ence to t2)
```

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 sp
ace
    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 available.

This Java tutorial was added on 12/10/2014.

Java Garbage Collection Introduction Types of Java Garbage Collectors

Comments on "How Java Garbage Collection Works?" Tutorial:

Java Garbage Collection Introduction says: *12/10/2014 at 10:32 pm*

[…] How Java Garbage Collection Works? […]

Types of Java Garbage Collectors says: *12/10/2014 at 11:15 pm*

[…] is the third part in the garbage collection tutorial series. In the previous part 2 we saw about how garbage collection works in Java, it is an interesting read and I recommend you to go through it. In the part 1 introduction to Java […]

Stefan says: *14/10/2014 at 12:54 am*

Joe, big props! Awesome tutorial and in general your blog Rocks ! :) Keep it up! You make it sound, so easy! But thumbs up, as well for the major technical areas covered in such simple and yet thoroughly explained wording!

Java Coder says: *16/10/2014 at 12:03 am*

Am little confused with “Example Program for GC Scope” could you please explain in detail.

Java Garbage Collection Monitoring and Analysis says: *19/10/2014 at 5:07 pm*

[…] its time to monitor the garbage collection process. Start your Java application and it will be auto detected and shown in the Java VisualVM […]

Prashant Singh says: *04/12/2014 at 12:02 pm*

Hi Joe,
I have one confusion that which type of object are marked under GC cycle i.e. referenced or derefenced, I think only referenced object are marked so they can moved to next Area(S0,S1 or old Gen).

Steve says: *27/06/2015 at 11:30 am*

The best tutorial for Java garbage collection on Net!

Richie says: *27/06/2015 at 1:33 pm*

I though Garbage collection is a complex topic. You explained it in a simple and easy to understand manner. Kudos!

Thanks, keep writing more tutorials.

Rashi says: *07/07/2015 at 12:43 pm*

Awesome explanation dude!!! Thanks :)

Comments are closed for this "How Java Garbage Collection Works?" tutorial.

↑ Go to Top Site Map

JAVA ANDROID DESIGN PATTERNS SPRING

WEB SERVICES SERVLET