**What is Garbage Collection in Java**

Memory management is one of the most crucial works for any programming language. Considering an application that reads a set of data and than write it to a database, for this it will use some intermediary storage place where it will store the data before putting it on to the database. Now, if that intermediary memory place is not cleared of the previous data it might result into its exhaustion which will bring our application down. To prevent such a case from occurring there can be two ways, one, clear the memory manually like in C++ or C, else, employ a background process of **garbage collection** to do this work automatically.

Now, whenever a software program runs it eats away some memory, like heap where the objects are created. **Garbage collection** deals with it and its main purpose are to delete all the objects that are either not in use or out of reach. So, all we do through a Java code is that we create objects as per our need and then destroy them when they are no longer in use. This garbage collector is under the control of JVM. Though JVM can be commanded to run the Garbage collector but then there seems no guarantee that actually the Garbage collector will be set out to work. So, the Garbage collector is normally run at the times when the JVM falls short of memory.

**Garbage Collector at work**

It is said that garbage collection uses “mark and sweep” algorithm and some other details however, the first thing we must know is that, when does one object become eligible for the garbage collection….or shall we say, is a garbage to the JVM. When we run the main() thread (one thread which will always be there along with other threads) and this thread can either be alive that is running or dead that is the execution is complete but it may also be the case that while main() thread is silent some other threads are working on it. And all the life cycle of the thread is maintained on its stack trace. And when garbage collector finds this that the particular object is no more accessible to any threads i.e. it is unreachable or no references exist to the object, such an object becomes vulnerable to be caught by the garbage collector.

Now, to make the object unreachable it is clear that the references to that object should cease to exist and this can be done by removing the reference to the object either by making the references null or by reassigning a new object to the existing reference for the current object.

Let’s take some examples to understand how it can be done.

**A) What if the “Reference is Null”?**

Now, all we are looking forward to do here is to nullify the references to an object and once there are no reachable references the object looses its importance and is no longer needed thus becomes eligible for Garbage Collection.

Let’s have a look at the example.

[code language=”java”]  
public class RefrenceGarbageDemo{

public static void main(String [] args) {  
StringBuffer greet = new StringBuffer("hello");  
System.out.println(greet);  
greet = null;  
}  
}  
[/code]

Here, in the above given example code we can see that before the reference to the StringBuffer object “hello” is removed or nulled i.e. greet is set to null there can be seen on the console the print “hello” but it more refers to “hello” after greet = null;

**B) Try “Reassigning a Reference Variable”**

Another way in effect to remove the reference to a particular object and making it eligible for Garbage Collection is to set the reference variable which currently refers to it to a new object. Thus, re-assigning the reference variable. Let’s have a look at the following code –

[code language=”java”]  
class RefrenceGarbageDemo{

public static void main(String [] args) {  
StringBuffer greet = new StringBuffer("hello");  
StringBuffer greetNu = new StringBuffer("goodbye");  
System.out.println(s1);  
greet = greetNu;  
}  
}  
[/code]

**C) How about “Isolating a Reference”…**

This time we are not removing the references but just isolating them. Consider a case where a reference to an object is also used to refer to another reference to the same object. Now imagine that two such instances exist and that they refer to each other. If all other references to these two objects are removed, then even though each object still has a valid reference, there will be no way for any live thread to access either object. When the garbage collector runs, it will remove such objects.

[code language=”java”]  
public class Referen {  
Referen r;

public static void main(String args[]) {  
Referen r2 = new Referen ();  
Referen r3= new Referen ();  
Referen r4= new Referen ();  
r2.r = r3;  
r3.r = r4;  
r4.r = r2;  
r2 = null;  
r3 = null;  
r4 = null;  
}  
}  
[/code]

However, we must always keep this in mind that the garbage collection is all not all in our control and how JVM works for it and selects the algorithm is all left to JVM’s discretion.

**I have covered Basics of Java Garbage collection above, which is more than sufficient for a java beginner. However  if you want to read more and more about it then continue below!!!**

Understanding Java Garbage Collection Concept is the most important step towards developing any good performance oriented and reliable java application. This document briefs the working of Garbage Collection and different algorithms employed by JVM to garbage collect.

“Garbage Collection” as the name suggests is used to remove the objects that are no longer needed by the Java application. The Java Garbage Collector will identify those objects and remove those from the memory thereby freeing the memory for the new objects.

**Defragmentation:**

The Java Objects are stored in the Heap. The Garbage Collector does another important work of *Defragmenting* (similar to our Windows Disk Defragmenter) the Heap spaces. The Objects collected by the Garbage Collector doesn’t reside in contiguous memory location, they will be collected from random location in the Heap. So fragmenting (collating into contiguous memory location) those random spaces becomes an important task to be performed by the Garbage Collector after freeing the Heap.

**Advantages of Garbage Collector**

1. It makes the developer’s life simple as the thorniest task of maintaining the memory is taken care by the JVM more effectively and efficiently.
2. It also ensures the integrity and security of the application is maintained as the developer is not involved in Memory Management which might accidentally crash the JVM due to improper memory handling.

**Sun JVM Heap Configuration**

The configuration of the heap is different for different vendor implementations. Let us consider the Sun JVM Heap configuration.

The Heap is divided into regions referred as Generations (Refer below Figure):

1. Young Generation – further divided into Eden, Survivor Space 1 and Survivor Space 2.
2. Tenured Generation.
3. Permanent Generation.

Most Java Objects created are initially allocated in Eden. When the Eden space is full the objects are moved to the Survivor Space 1 and when the Survivor Space 1 is full then the all the live objects are moved to Survivor Space 2. The objects that live for longer time or not Garbage collected in Young Generation space are moved to Tenured Generation.

The third generation is Permanent Generation which holds objects describing classes, methods, Constant pool Information, Object arrays associated with the classes, JVM internal objects and other information needed for optimization by the JIT compilers.

Garbage collection occurs in each generation when it fills up thereby freeing the spaces for new objects. The collection of unused objects in the any of the three regions inside Young Generations is termed as *Minor Collection*and from Tenured or Old Generation is termed as *Major Collection.*GenerallyMajor collections last much longer than Minor collections because a significantly larger number of objects are involved during Major Collections.

**Garbage Collection Types**

There are four different types of Collectors available,

* Serial Collector.
* Parallel Collector.
* Parallel Compacting Collector.
* Concurrent Mark-Sweep (CMS) Collector.

In the next section, let us briefly see the working of above collectors.

***1) Serial Collector***

The collections using Serial Collector are done in ‘Stop the World’ fashion. That is, the execution of application is paused while the collection of garbage takes place.

***Collection of Young Generation using Serial Collector***

* The live objects in Eden Space are copied to the empty survivor spaces.
* The live objects which do not fit completely in the empty survivor spaces are directly moved to Old Generation.
* The live objects in the Survivor space 1 that are still relatively Young are also copied to other Survivor Space, while the Old Objects are copied to the Old generation.
* Any objects that are remained in the Eden or Survivor Space 1 after the live objects are copied are considered as not live and will be garbage collected.
* After the collection, the Eden and Survivor Space 1 are empty and the Survivor Space 2 is with live objects.
* Now the Survivor Spaces will swap their roles.

***Collection of Old Generation using Serial Collector***

* Mark Sweep Compact collection algorithm is used to collect the Objects from Old Generation.
* Marking Phase: The collector identifies the live objects and marks them in the mark phase.
* Sweeping Phase: Then in the sweep phase the collector sweeps over the generations identifying the garbage.
* Compaction Phase: Once the garbage collected, then Sliding Compaction will be performed. This slides the live objects to the beginning of the Old generation and free spaces will be contiguous in the other end.

***When to use Serial Collector***

The serial collector is the collector of choice for applications that are run on client style machines and that do not have a requirement for low pause times. On current hardware configurations, the serial collector can efficiently manage a lot of simple applications with short pauses of less than half a second for full collections.

***How to use Serial Collector***

In the Java 5 release, the serial collector is automatically chosen as the default garbage collector on machines that are client style machines. On other machines, the serial collector can be explicitly requested by using the -XX:+UseSerialGC command line option.

***2) Parallel Collector***

The Parallel Collector, (AKA throughput collector), was developed in order to take advantage of available CPUs rather than leaving them idle while garbage collection work.

***Collection of Young Generation using Parallel Collector***

The parallel collector uses a parallel version of serial collector’s young generation collection algorithm. Parallel Collector still works in stop the world fashion, but since the collection is parallel, using many CPUs, garbage collection overhead decreased and hence application throughput is increased.

***Collection of Old Generation using Parallel Collector***

Old generation garbage collection for the parallel collector is done using the serial mark sweep compact collection algorithm used by the serial collector.

***When to use Parallel Collector***

Applications that can benefit from the parallel collector are those that run on machines with more than one CPU and do not have pause time constraints. Infrequent, but long, old generation collections can occur. Scenarios for which the parallel collector can be used are batch processing applications, billing process, and etc.

***How to use Parallel Collector***

In the Java 5 release, the parallel collector is automatically chosen as the default garbage collector on server-class machines.  The parallel collector can be explicitly requested by using the -XX:+UseParallelGC command line option.

***3) Parallel Compacting Collector***

The Parallel Compacting collector is more efficient than the Parallel Collector as it uses new algorithm to garbage collect the Old generations. The new algorithm enhances the Sweep Phase in Parallel Collector by introducing a new phase (Summary Phase), which effectively collects the Old Generation when compared to Parallel Collector collection of Old Generation.

***Collection of Young Generation using Parallel Compacting Collector (PCC)***

The algorithm for PCC of Young Generation is same as the collection of Young Generation using Parallel Collector.

***Collection of Old Generation using Parallel Compacting Collector (PCC)***

The old and permanent generations are collected in a stop the world fashion, in parallel fashion with sliding compaction. There are three phases involved.

* Marking Phase: The generations are logically divided into fixed-sized regions. The live objects are divided among garbage collection threads and they are marked parallel. When the objects are marked live the data of its region is updated with size and location of that object.
* Summary Phase: This phase operates on region rather than on objects. Some portion of the generation will be densely packed due to previous compactions. This phase will examine the density of the phases starting from left most area and stop at a point where the space could be recovered. Left side of that point is considered as dense prefix and the right side of that point will be considered as the free space where the compaction can happen and dead space will be eliminated. This phase just calculates the first byte of the live object for each of the compacted regions. Currently this phase happens in serial manner without any hiccups in performance.
* Compaction Phase: The Garbage Collection Threads will use the Summary Phase data to identify the regions and threads will copy the data into these regions.

***When to use Parallel Compacting Collector***

Parallel Compacting Collector is more efficient and beneficial than the Parallel Collector discussed in the previous section which can be used on applications that run on machines with multiple CPU’s.

***How to use Parallel Compacting Collector***

Parallel Compacting Collector can be used by specifying the command line option XX:+UseParallelOldGC.

***4) Concurrent Mark Sweep Collector***

Old generation collections, though infrequent, can compel long pauses, when large heaps are involved. To address this issue, the HotSpot JVM (introduced as an add-on in Java 1.2 and made default from Java 1.3) includes a collector called the *concurrent mark-sweep (CMS) collector*, also known as the *low-latency collector*.

***Collection of Young Generation using Concurrent Mark Sweep Collector***

This is same as the Parallel Collector.

***Collection of Old Generation using Concurrent Mark Sweep Collector***

The collections are done concurrently with the execution of the application.

* The collection cycle of CMS collector starts with a short pause called Initial Mark where the live objects are identified and then marked during the concurrent marking phase.
* Since this marking of live objects happens while the application is running there might be possibility of live objects being not marked. For this reason another short pause will happen called *remark* which will finalize the marking.
* At the end of *remark* phase, *concurrent sweep* phase will reclaim all the garbage.
* This collector doesn’t perform the compaction operation. It just frees the space but not collate it. This saves time but the simple pointer to next free location cannot be maintained as in other collectors instead free lists will be created linking the unallocated regions and these free lists will be searched when the new objects are allocated.
* To perorate, the CMS when compared to parallel collector, old generation pauses are decreased drastically, young generations are slightly longer and extra heap sizes are needed.

***When to use CMS Collector***

CMS can be used based on the below needs of the application,

* Shorter garbage collection pauses,
* Application which can share processor resources as the CMS runs concurrently,
* Application which can have large Old generations.

***How to use CMS Collector***

CMS collector can be used, by specifying the command line option XX:+UseConcMarkSweepGC

**Introduction of Garbage First Garbage Collector**

The Garbage-First Garbage Collector (aka G1 GC) is a new GC that is being introduced in the Java HotSpot VM in JDK 7. An initial version of G1 has been released in Java SE 6 Update 14. G1 is the replacement for Concurrent Mark-Sweep GC.

**Working of G1 GC**

* The heap is split into fixed size regions and the separation between the two generations is basically logical. So some regions are considered to be young, some old.
* Reclamation of space is done through copying. G1 selects a set of regions, pick the surviving object from those regions and copy them to another set of regions, instead of the combination of copying and in-place de-allocation that CMS does.
* *Compaction*: G1 GC is a compacting collector. Compacting is the process of moving the live object towards one end of the heap thereby the other end becomes contiguous free area of memory. G1 compacts efficiently to avoid the free lists for allocation used by CMS and eradicates potential fragmentation issues.
* *Low Pauses and avoid Full GC*: There will always be areas of contiguous free space ready for allocation, allowing G1 to have consistent low pauses over time. As G1 scans throughout the heap to identify the live objects, it knows the regions that are mostly empty and concentrate on those regions thereby making lot of free space available for allocation and try avoiding the Full GC.

**Main difference between G1 GC and CMS collectors**

* *Compaction*: G1 GC is a compacting collector. Compacting is the process of moving the live object towards one end of the heap thereby the other end becomes contiguous free area of memory. G1 compacts efficiently to avoid the free lists for allocation used by CMS and eradicates potential fragmentation issues.
* *Low Pauses and avoid Full GC*: There will always be areas of contiguous free space ready for allocation, allowing G1 to have consistent low pauses over time. As G1 scans throughout the heap to identify the live objects, it knows the regions that are mostly empty and concentrate on those regions thereby making lot of free space available for allocation and try avoiding the Full GC.

**Garbage Collection Performance**

The three main characteristics that affect garbage collection performance are:

* Pause: The amount of time that the JVM pauses while performing garbage collection.
* Throughput: The ratio of garbage collection time to application time.
* Footprint: The Heap Size

The goal is to minimize the pause and maximize the throughput. There is a tradeoff in defining the sizes of each region, *“the larger the region, the less frequently it needs to be collected, but the longer each collection will take”*.

The Maximum Pause Time Goal can be specified using the below command line option

***-XX:MaxGCPauseMillis=<Value>****Where <Value>  user defined*

The throughput goal can be specified using the below command line option***XX:GCTimeRatio=<Value>***, which sets the ratio of garbage collection time to application time to ***1 / (1 + < Value >).***

For example, -XX:GCTimeRatio=19 sets a goal of 1/20 or 5% of the total time in garbage collection. The default value is 99, resulting in a goal of 1% of the time in garbage collection.

Maximum heap footprint (i.e. heap Size) is specified using the option ***-Xmx<N>***. The heap size depends upon the system’s hardware. Application’s memory footprint should not exceed the available physical memory.

**Sizing Heap spaces**

Listed below are some of the JVM options to size the heap

* -Xmx<size> : max heap size (young generation + old generation)
* -Xms<size> : initial heap size (young generation + old generation)
* -Xmn<size> : young generation size
* Applications that stress on performance tend to set -Xms and -Xmx to the same value. When -Xms != -Xmx, heap growth or shrinking requires a Full GC.
* -XX:PermSize=<size> : permanent generation initial size
* -XX:MaxPermSize=<size> : permanent generation max size
* Applications with emphasis on performance almost always set -XX:PermSize and -XX:MaxPermSize to the same value. Growing or shrinking the permanent generation requires a Full GC too.

In general, a particular generation sizing also plays an important part. For example, a very large young generation may maximize throughput, but at the expense of footprint, promptness and pause times. Small Young generation minimizes the Young Generation pauses at the expense of throughput. The sizing of one generation does not affect the collection frequency and pause times for another generation.

There can be no one straightforward way to size the generations. The developer has to choose the best way based on how application uses memory and other requirements.

**OutOfMemory Error**

The most common issue any java developers will encounter is java.lang.OutOfMemoryError. An OutOfMemoryError does not necessarily involve a memory leak. The issue might simply be a configuration issue, for example if the specified heap size (or the default size if not specified) is insufficient for the application.

To diagnose this error, the complete exception message has to be analyzed. Some common additional information’s are,

* ***Java Heap size****:* This indicates that object could not be allocated in the heap. The possible reasons are,
  + Maximum or default heap size specified is not sufficient for the application.
  + Objects that are no longer needed cannot be garbage collected as the applications are still having the reference unintentionally.
  + Excessive use of finalizers in the application.
* ***PermGen space****:*This indicates that the permanent generation is full. If an application loads a large number of classes, then the permanent generation may need to be increased. Command line option –XX: MaxPermSize=n, where n specifies the size can be provided.
* ***Requested array size exceeds VM limit****:*The application attempted to allocate an array that is larger than the heap size. For example, if an application tries to allocate an array of 128MB but the maximum heap size is 64MB, then this error will be thrown.

Use of HAT tool, jconsole etc can be used to identify the above issues. The proper runtime JVM configuration will also eradicate the OutOfMemoryError Issues.