[The JVM Architecture Explained](https://dzone.com/articles/jvm-architecture-explained)

Every Java developer knows that bytecode will be executed by **JRE** (Java Runtime Environment). But many doesn't know the fact that **JRE** is the implementation of **Java Virtual Machine** (JVM), which analyzes the bytecode, interprets the code, and executes it. It is very important as a developer that we should know the Architecture of the JVM, as it enables us to write code more efficiently. In this article, we will learn more deeply about the **JVM architecture** in Java and the **different components** of the JVM.

## What is the JVM?

A **Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (Write Once Run Anywhere)**,which runs on a **VM**. The**compiler** compiles the Java file into a Java **.class** file, then that **.class** file is input into the JVM, which Loads and executes the class file. Below is a diagram of the Architecture of the JVM.



## How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. **Class Loader Subsystem**
2. **Runtime Data Area**
3. **Execution Engine**

### 1. Class Loader Subsystem

Java's [**dynamic class loading**](http://www.javainterviewpoint.com/use-of-class-forname-in-java/) functionality is handled by the class loader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at **runtime**, not **compile time.**

#### 1.1 Loading

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

1. **Boot Strap [ClassLoader](http://www.javainterviewpoint.com/)** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.**Highest priority will be given to this loader.

Bootstrap ClassLoader doesn’t have any parent ClassLoaders. It is also called as the **Primodial ClassLoader**

1. **Extension ClassLoader** – Responsible for loading classes which are inside **ext** folder **(jre\lib).**

It loads files from ***jre/lib/ext*** directory or any other directory pointed by the system property ***java.ext.dirs***.

1. **Application ClassLoader** –Responsible for loading **Application Level Classpath**, path mentioned Environment Variable etc.

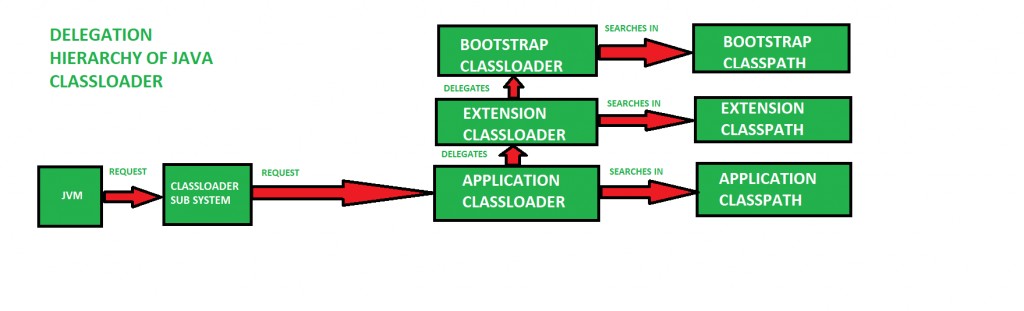
it loads the Application type classes found in the environment variable ***CLASSPATH, -classpath or -cp command line option***. The Application ClassLoader is a child class of Extension ClassLoader.

The above **Class Loaders** will follow **Delegation Hierarchy Algorithm**while loading the class files.

1. **Delegation Model**: The Java Virtual Machine and the Java ClassLoader use an algorithm called the **Delegation Hierarchy Algorithm** to Load the classes into the Java file.

The ClassLoader works based on a set of operations given by the delegation model. They are:

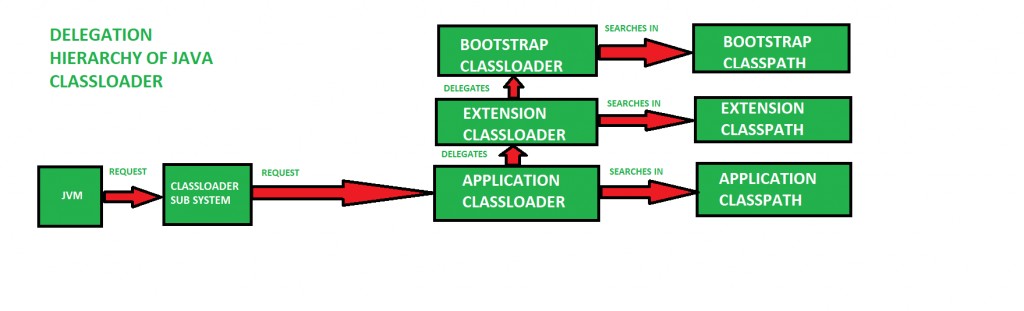
* + ClassLoader always follows the **Delegation Hierarchy Principle**.
  + Whenever JVM comes across a class, it checks whether that class is already loaded or not.
  + If the Class is already loaded in the method area then the JVM proceeds with execution.
  + If the class is not present in the method area then the JVM asks the Java ClassLoader Sub-System to load that particular class, then ClassLoader sub-system hands over the control to **Application ClassLoader**.
  + Application ClassLoader then delegates the request to Extension ClassLoader and the **Extension ClassLoader** in turn delegates the request to **Bootstrap ClassLoader**.
  + Bootstrap ClassLoader will search in the Bootstrap classpath(JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.
  + Extension ClassLoader searches for the class in the Extension Classpath(JDK/JRE/LIB/EXT). If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.
  + Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a **ClassNotFoundException** exception is generated.

[](https://media.geeksforgeeks.org/wp-content/uploads/20190417144207/java_classloader.png)

1. **Delegation Model**: The Java Virtual Machine and the Java ClassLoader use an algorithm called the **Delegation Hierarchy Algorithm** to Load the classes into the Java file.

The ClassLoader works based on a set of operations given by the delegation model. They are:

* + ClassLoader always follows the **Delegation Hierarchy Principle**.
  + Whenever JVM comes across a class, it checks whether that class is already loaded or not.
  + If the Class is already loaded in the method area then the JVM proceeds with execution.
  + If the class is not present in the method area then the JVM asks the Java ClassLoader Sub-System to load that particular class, then ClassLoader sub-system hands over the control to **Application ClassLoader**.
  + Application ClassLoader then delegates the request to Extension ClassLoader and the **Extension ClassLoader** in turn delegates the request to **Bootstrap ClassLoader**.
  + Bootstrap ClassLoader will search in the Bootstrap classpath(JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.
  + Extension ClassLoader searches for the class in the Extension Classpath(JDK/JRE/LIB/EXT). If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.
  + Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a **ClassNotFoundException** exception is generated.

[](https://media.geeksforgeeks.org/wp-content/uploads/20190417144207/java_classloader.png)

1. **Visibility Principle**: The **Visibility Principle** states that a class loaded by a parent ClassLoader is visible to the child ClassLoaders but a class loaded by a child ClassLoader is not visible to the parent ClassLoaders. Suppose a class GEEKS.class has been loaded by the Extension ClassLoader, then that class is only visible to the Extension ClassLoader and Application ClassLoader but not to the Bootstrap ClassLoader. If that class is again tried to load using Bootstrap ClassLoader it gives an exception ***java.lang.ClassNotFoundException***.
2. **Uniqueness Property**: The **Uniquesness Property** ensures that the classes are unique and there is no repetition of classes. This also ensures that the classes loaded by parent classloaders are not loaded by the child classloaders. If the parent class loader isn’t able to find the class, only then the current instance would attempt to do so itself.

#### 1.2 Linking

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the **verification error.**
2. **Prepare** – For all static variables memory will be allocated and assigned with **default values.**
3. **Resolve** – All **symbolic memory references** are replaced with the **original references** from **Method Area**.

#### 1.3 Initialization

This is the final phase of Class Loading, here all [**static variables**](http://www.javainterviewpoint.com/use-of-static-keyword-in-java/)will be assigned with the original values, and the [**static block**](http://www.javainterviewpoint.com/java-static-import/) will be executed.

### 2. Runtime Data Area

The Runtime Data Area is divided into 5 major components:

1. **Method Area** – All the **class level data** will be stored here, including **static variables**. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the **Objects** and their corresponding**instance variables** and **arrays** will be stored here. There is also one Heap Area per JVM. Since the **Method** and **Heap areas** share memory for multiple threads, the data stored is not thread**safe.**
3. **Stack Area** – For every thread, a separate **runtime stack** will be created. For every **method call**, one entry will be made in the stack memory which is called as **Stack Frame**. All **local variables** will be created in the stack memory. The stack area is thread safe since it is not a shared resource. The Stack Frame is divided into three subentities:
   1. **Local Variable Array** – Related to the method how many **local variables** are involved and the corresponding values will be stored here.
   2. **Operand stack** – If any intermediate operation is required to perform, **operand stack** acts as runtime workspace to perform the operation.
   3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate**PC Registers,** to hold the address of **current executing instruction** once the instruction is executed the PC register will be **updated** with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

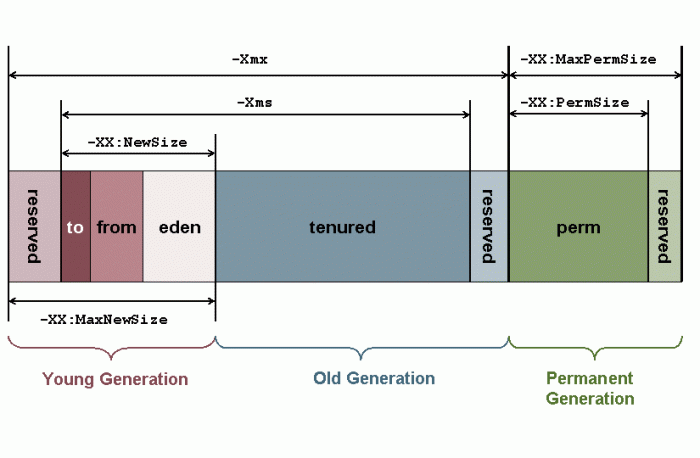
### 3. Execution Engine

The bytecode which is assigned to the **Runtime Data Area** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
   1. **Intermediate Code generator** – Produces intermediate code
   2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
   3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
   4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling **"System.gc()"**, but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**Java Native Interface (JNI)**: **JNI** will be interacting with the **Native Method Libraries** and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries**:It is a collection of the Native Libraries which is required for the Execution Engine.



### D:\Users\mohammes\Documents\Downloads\JUtH_20121024_RuntimeDataAreas_1_MemoryModel.pngTracking the leaks

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapDumps

A very useful tool that helps in heap analysis is jmap that comes with jdk 6. Two really useful commands using jmap are:

jmap -permstat <pid>

This command shows the classloaders in the perm gen space and their status if they are live or dead. This is very useful to check if the classloaders have been garbage collected.

jmap -dump:format=b,file=heap.bin <pid>

This dumps the heap into the heap.bin file and can then be examined to find the reason for the classloader not being garbage collected. We used the tool called [visualvm](http://visualvm.java.net/download.html" \o "Visual VM) another very useful tool. It helps view the dumped heap and can show the nearest GC root that is holding an object in the heap to prevent it from garbage collecting

So the Java classes are stored in the permanent generation. What all does that entail? Besides the basic fields of a Java class there are

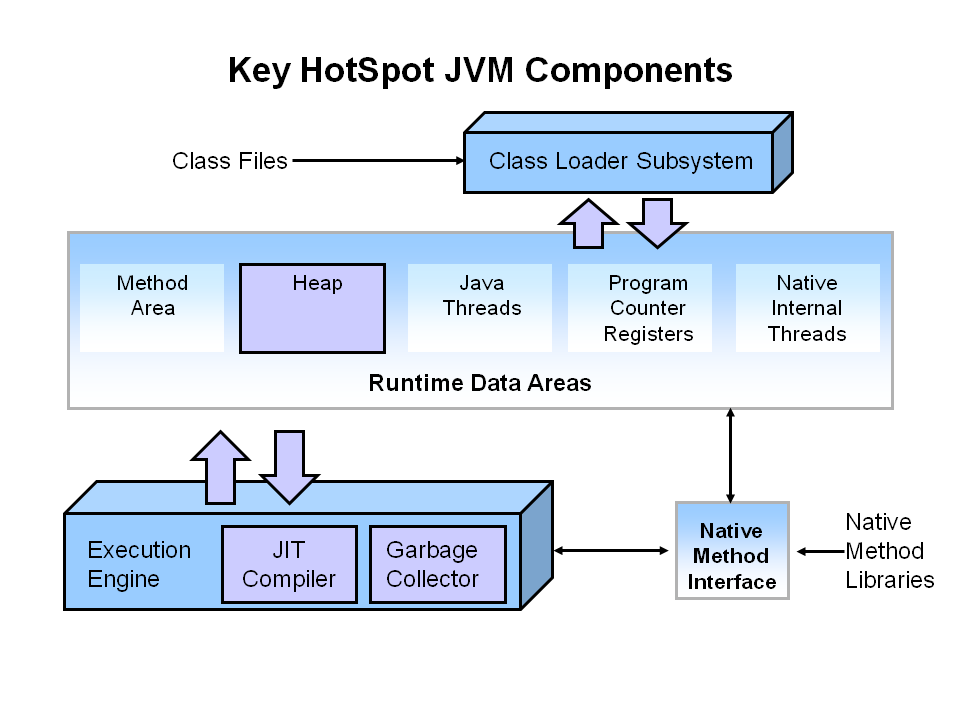
* Methods of a class (including the bytecodes)
* Names of the classes (in the form of an object that points to a string also in the permanent generation)
* Constant pool information (data read from the class file, see chapter 4 of the JVM specification for all the details).
* Object arrays and type arrays associated with a class (e.g., an object array containing references to methods).
* Internal objects created by the JVM (java/lang/Object or java/lang/exception for instance)
* Information used for optimization by the compilers (JITs)

From Oracle webite(http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html)

#### Hotspot Architecture

The HotSpot JVM possesses an architecture that supports a strong foundation of features and capabilities and supports the ability to realize high performance and massive scalability. For example, the HotSpot JVM JIT compilers generate dynamic optimizations. In other words, they make optimization decisions while the Java application is running and generate high-performing native machine instructions targeted for the underlying system architecture. In addition, through the maturing evolution and continuous engineering of its runtime environment and multithreaded garbage collector, the HotSpot JVM yields high scalability on even the largest available computer systems.

The main components of the JVM include the classloader, the runtime data areas, and the execution engine.



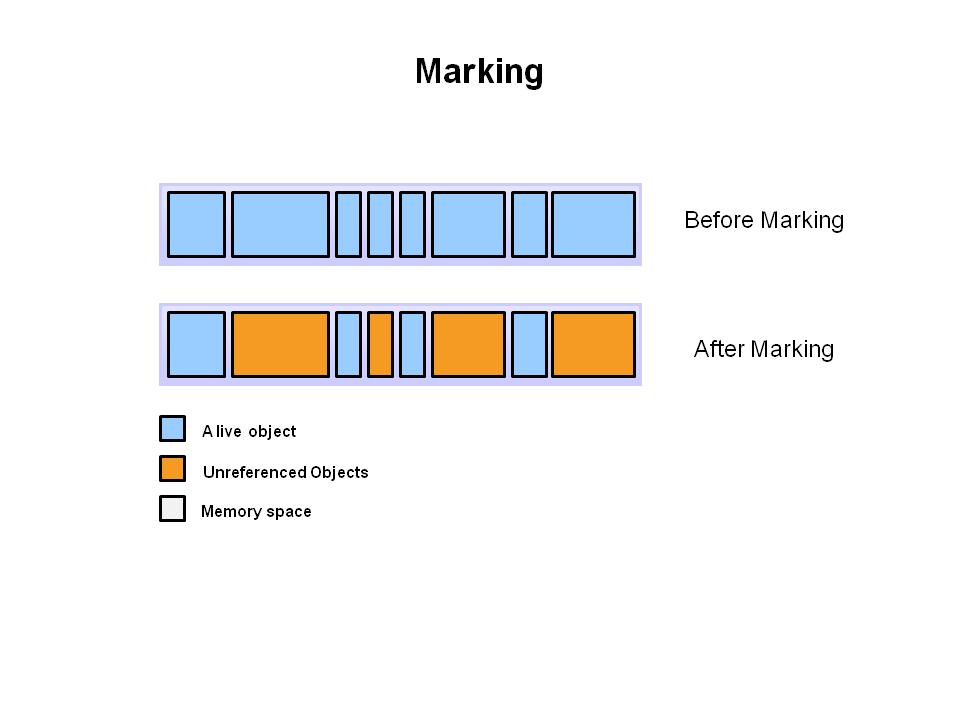
### What is Automatic Garbage Collection?

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

In a programming language like C, allocating and deallocating memory is a manual process. In Java, process of deallocating memory is handled automatically by the garbage collector. The basic process can be described as follows.

#### Step 1: Marking

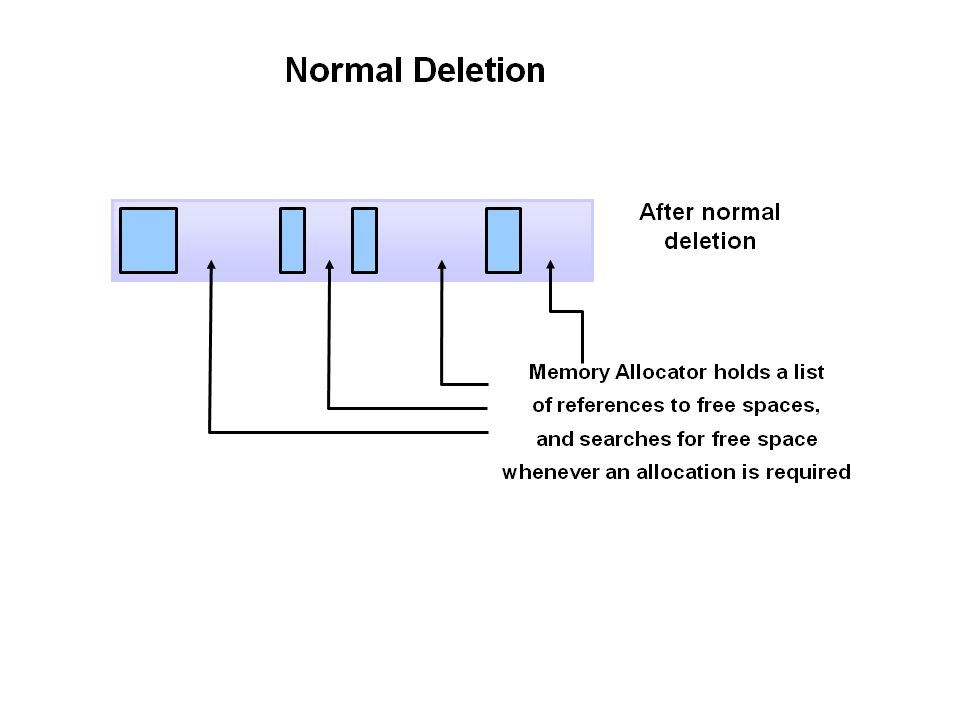
The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.



Referenced objects are shown in blue. Unreferenced objects are shown in gold. All objects are scanned in the marking phase to make this determination. This can be a very time consuming process if all objects in a system must be scanned.

#### Step 2: Normal Deletion

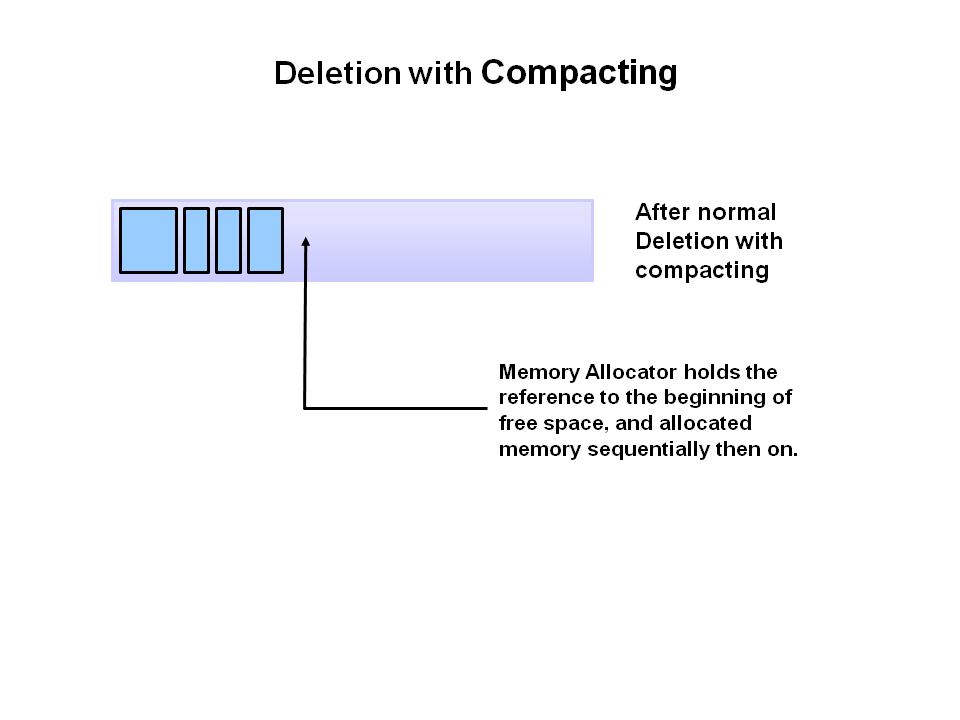
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.



The memory allocator holds references to blocks of free space where new object can be allocated.

#### Step 2a: Deletion with Compacting

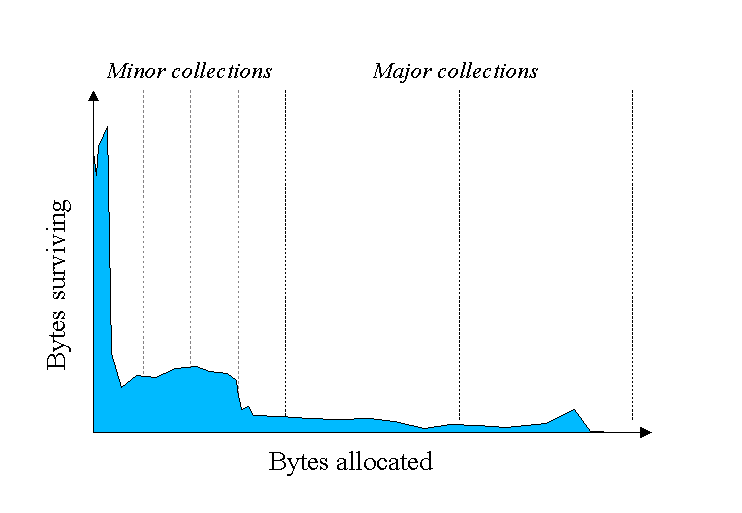
To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.



#### Why Generational Garbage Collection?

**As stated earlier, having to mark and compact all the objects in a JVM is inefficient**. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short lived.

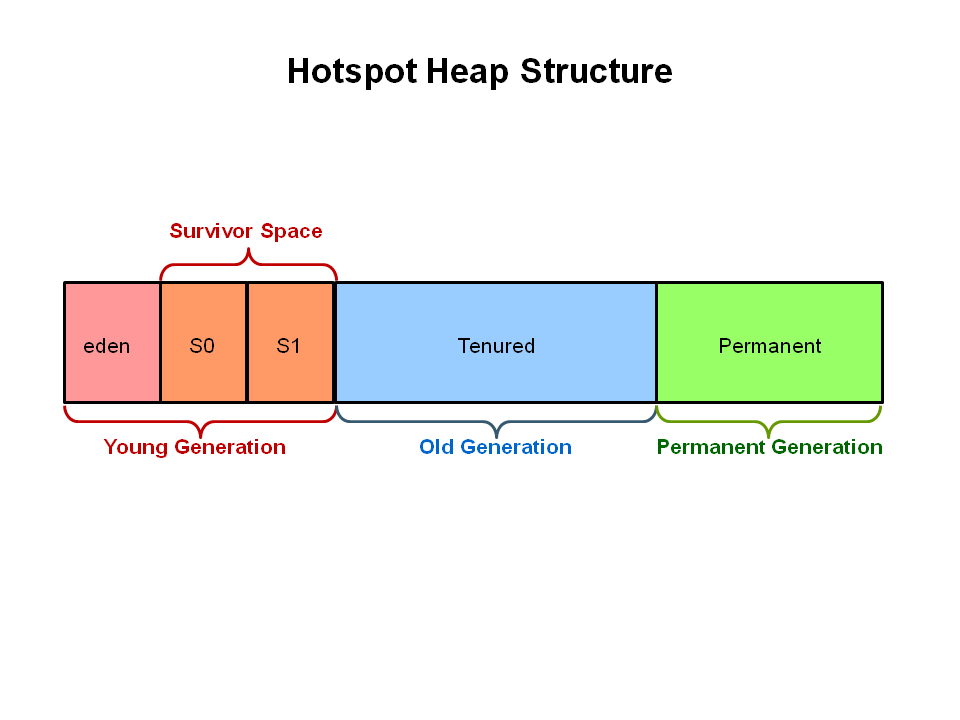
Here is an example of such data. The Y axis shows the number of bytes allocated and the X access shows the number of bytes allocated over time.



As you can see, fewer and fewer objects remain allocated over time. In fact most objects have a very short life as shown by the higher values on the left side of the graph.

#### JVM Generations

The information learned from the object allocation behavior can be used to enhance the performance of the JVM. Therefore, the heap is broken up into smaller parts or generations. The heap parts are: Young Generation, Old or Tenured Generation, and Permanent Generation



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a **minor garbage collection**. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

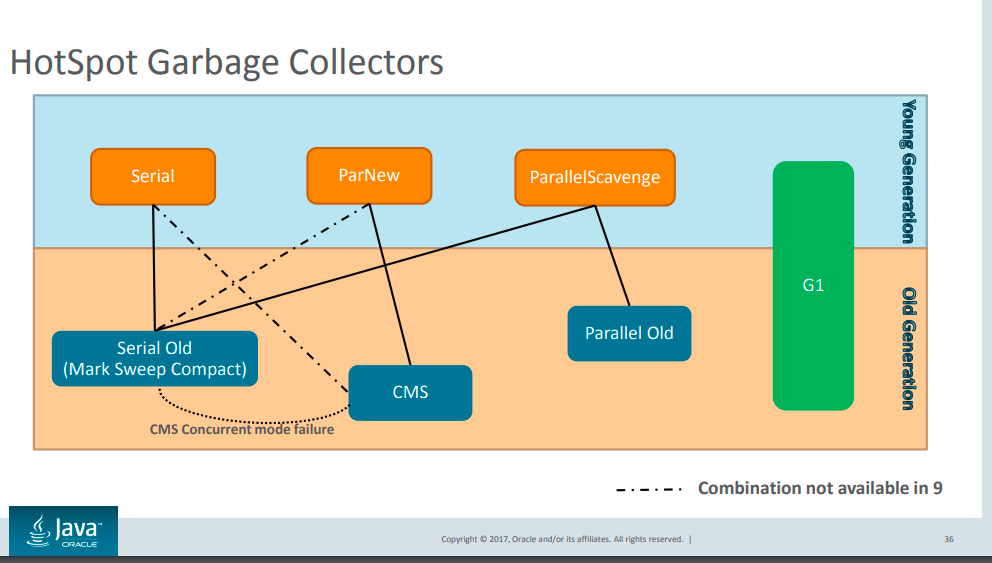
**Stop the World Event** - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

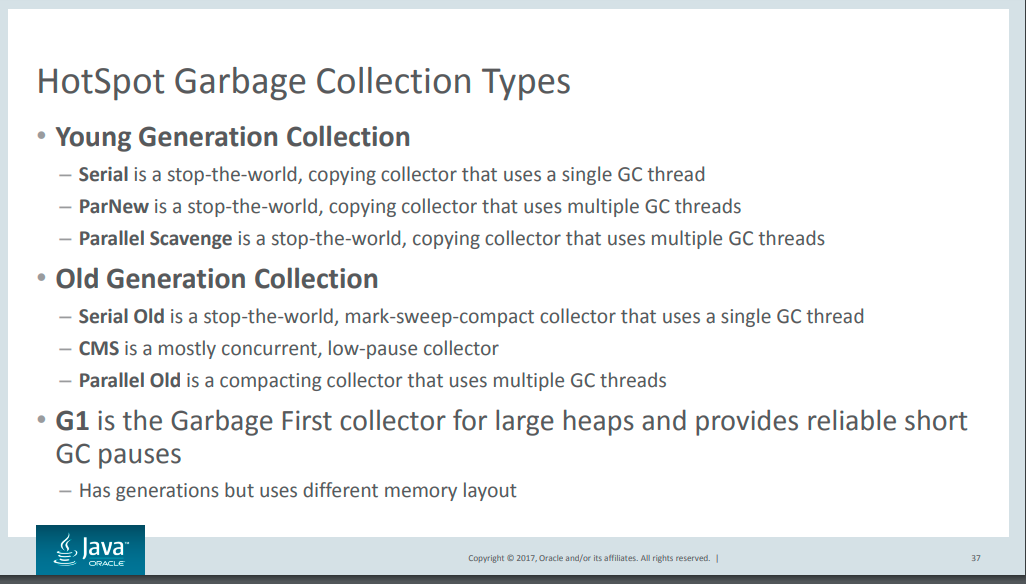
The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.

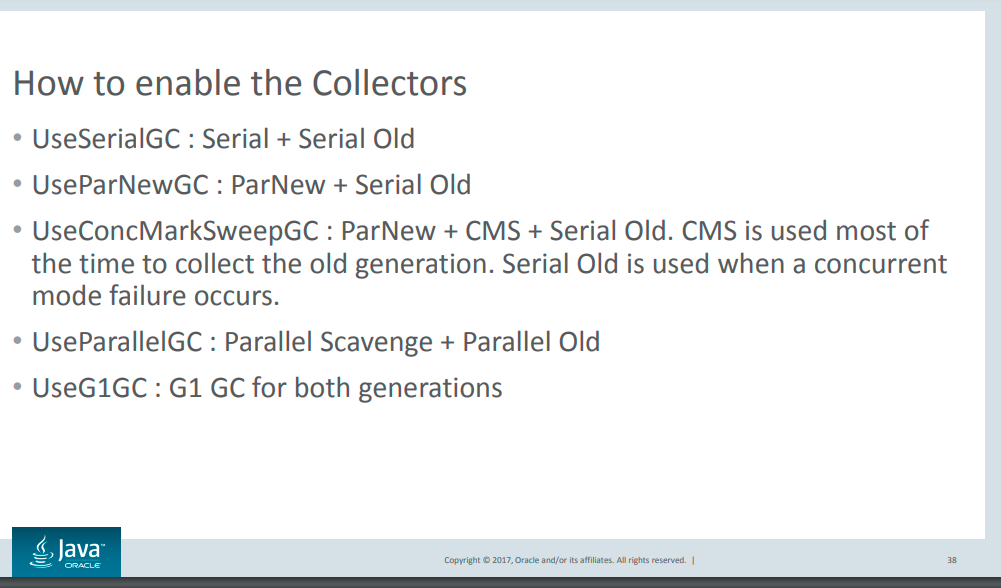
Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

### The Permanent generation contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.







**Table 1: JVM Options to Be Checked for GC Tuning.**

| **CLASSIFICATION** | **OPTION** | **DESCRIPTION** |
| --- | --- | --- |
| Heap area size | -Xms | Heap area size when starting JVM |
|  | -Xmx | Maximum heap area size |
| New area size | -XX:NewRatio | Ratio of New area and Old area |
|  | -XX:NewSize | New area size |
|  | -XX:SurvivorRatio | Ratio of Eden area and Survivor area |

**Table 2: Available Options by GC Type.**

| **CLASSIFICATION** | **OPTION** | **REMARKS** |
| --- | --- | --- |
| Serial GC | -XX:+UseSerialGC |  |
| Parallel GC | -XX:+UseParallelGC -XX:ParallelGCThreads=value |  |
| Parallel Compacting GC | -XX:+UseParallelOldGC |  |
| CMS GC | -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=value -XX:+UseCMSInitiatingOccupancyOnly |  |
| G1 | -XX:+UnlockExperimentalVMOptions -XX:+UseG1GC | In JDK 6, these two options must be used together. |

Java classes are stored in the permanent generation. What all does that entail? Besides the basic fields of a Java class there are

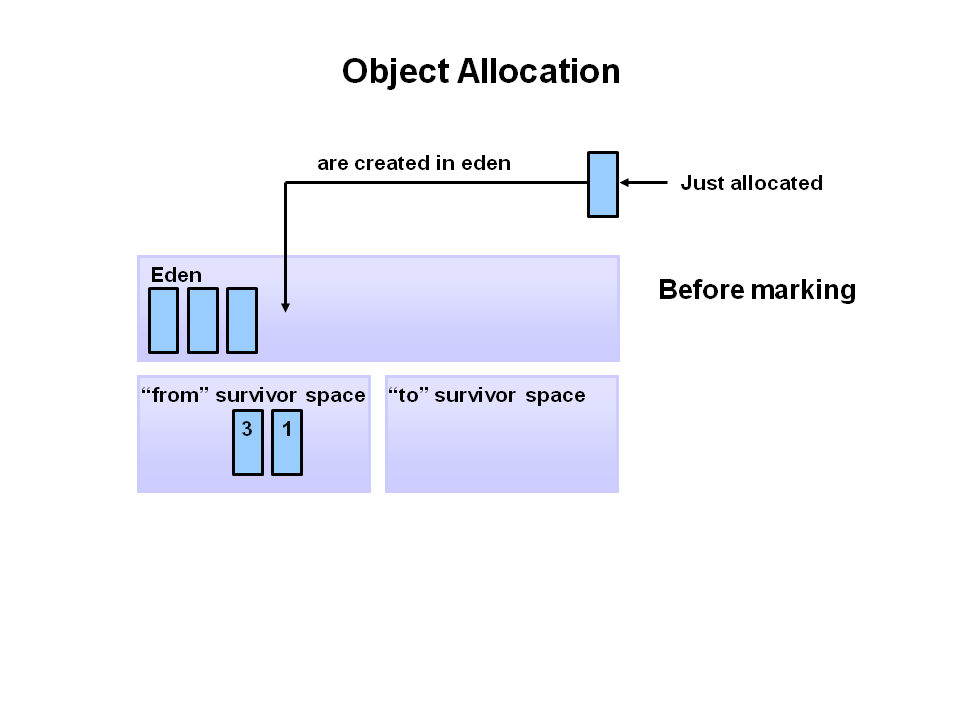
* Methods of a class (including the bytecodes)
* Names of the classes (in the form of an object that points to a string also in the permanent generation)
* Constant pool information (data read from the class file, see chapter 4 of the JVM specification for all the details).
* Object arrays and type arrays associated with a class (e.g., an object array containing references to methods).
* Internal objects created by the JVM (java/lang/Object or java/lang/exception for instance)
* Information used for optimization by the compilers (JITs)

(2)[Tuning GC with JVM 5 - Section 3 Generations](http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html)

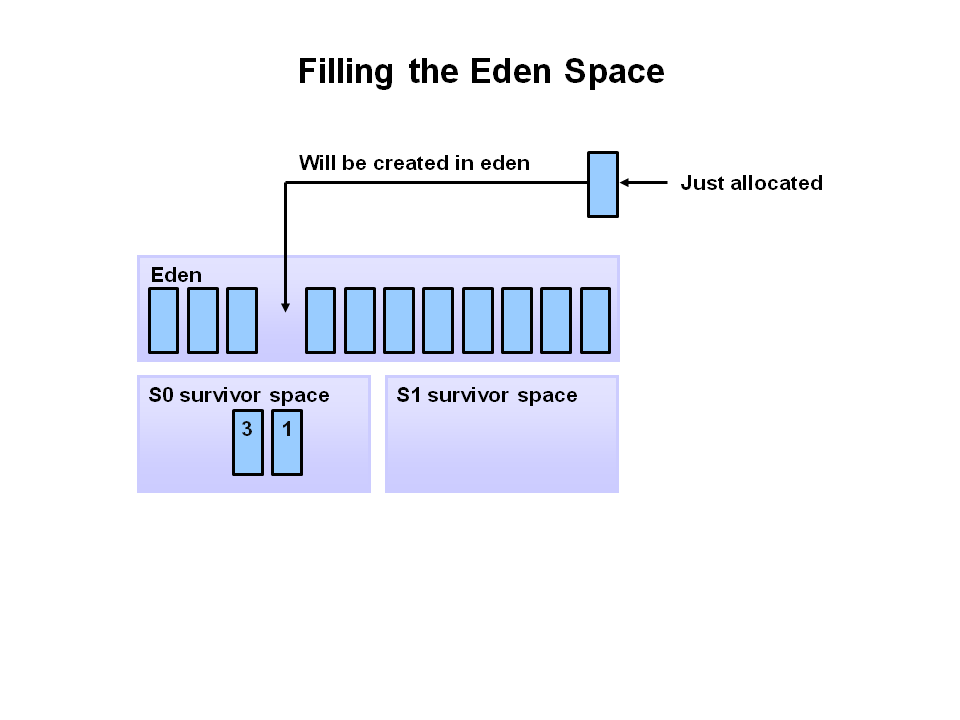
## The Generational Garbage Collection Process

Now that you understand why the heap is separted into different generations, it is time to look at how exactly these spaces interact. The pictures that follow walks through the object allocation and aging process in the JVM.

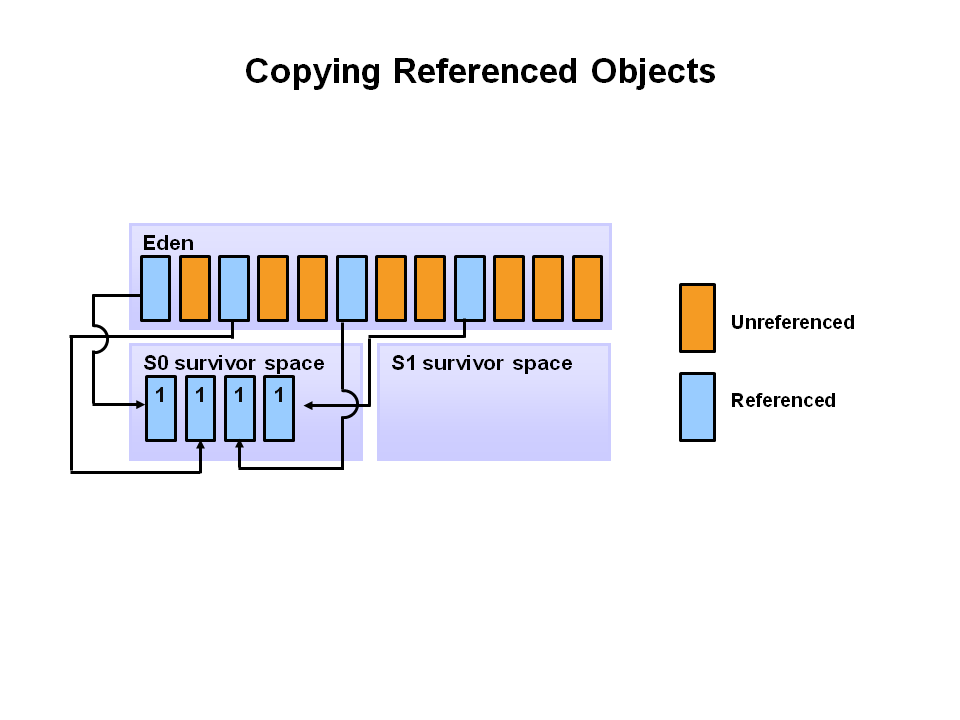
First, any new objects are allocated to the eden space. Both survivor spaces start out empty.



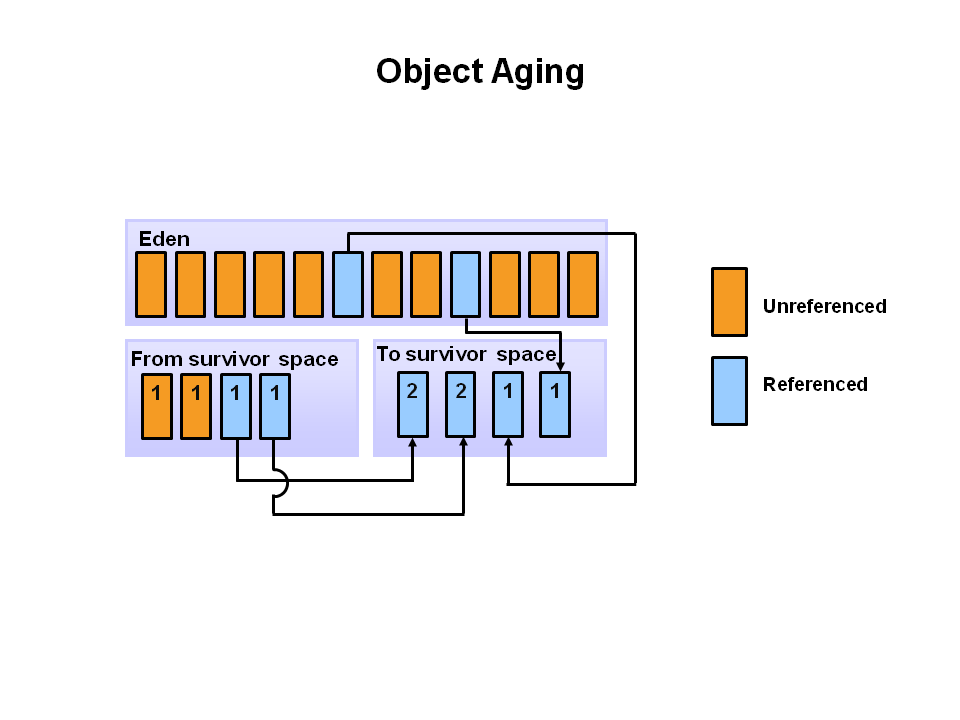
When the eden space fills up, a minor garbage collection is triggered.



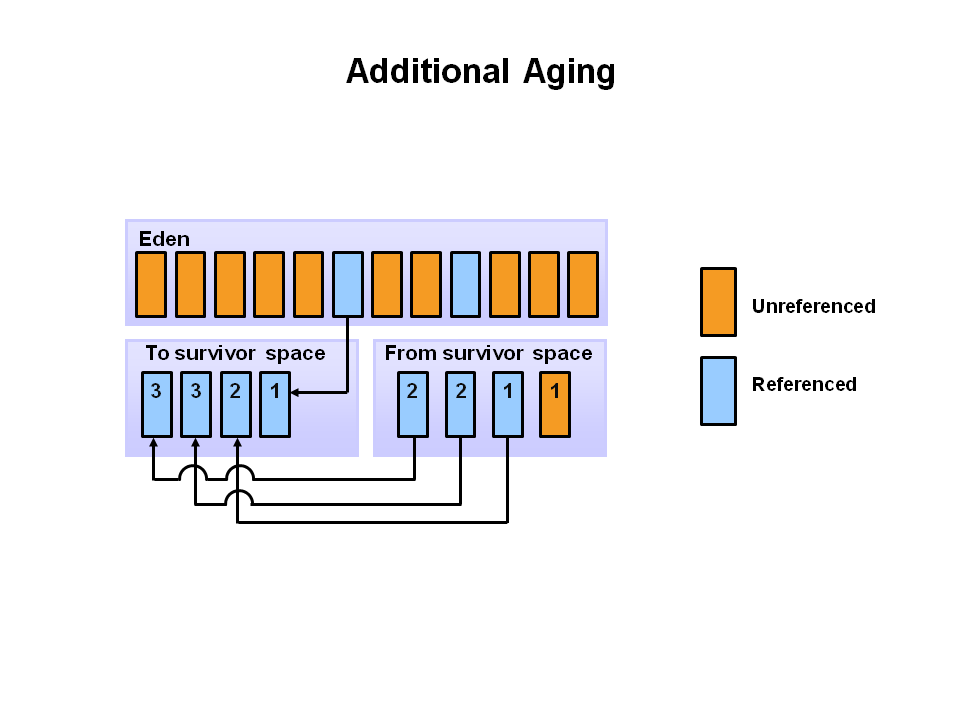
Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.



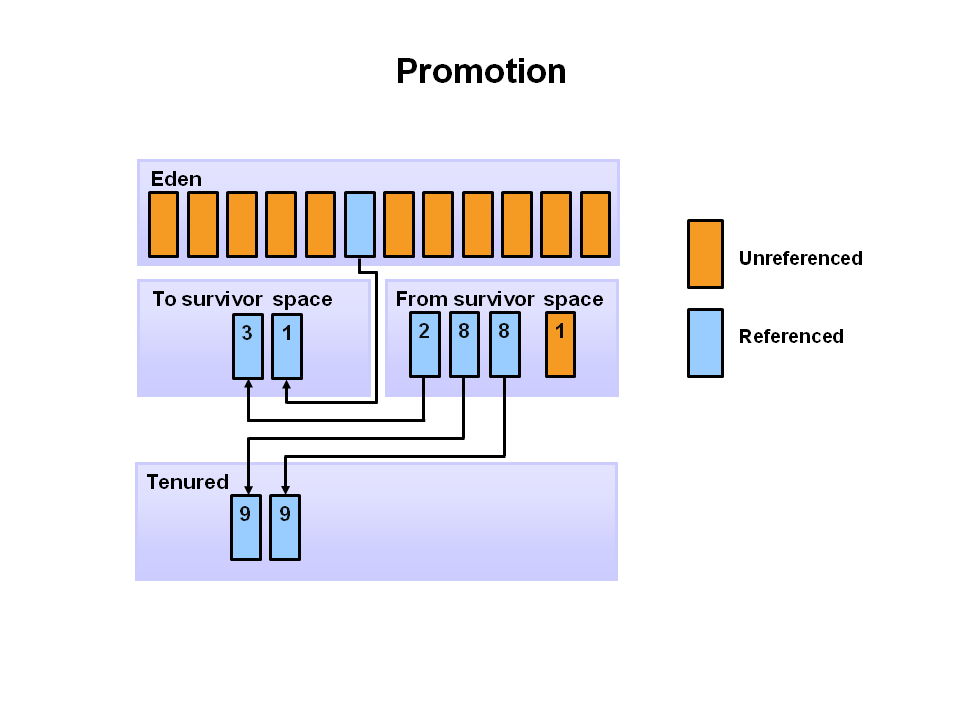
At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.



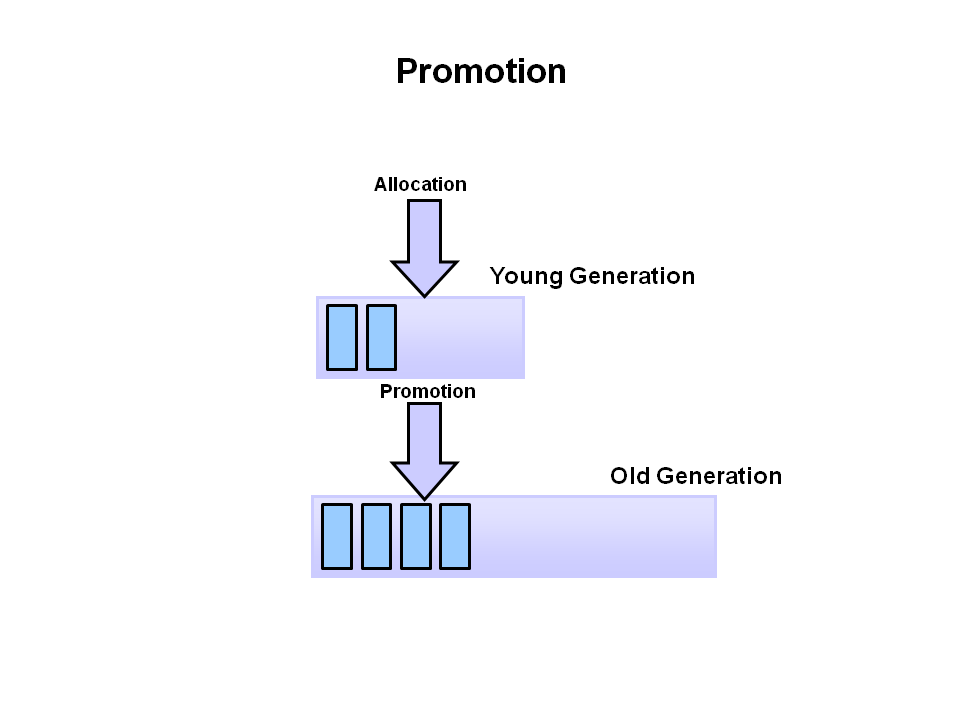
At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.



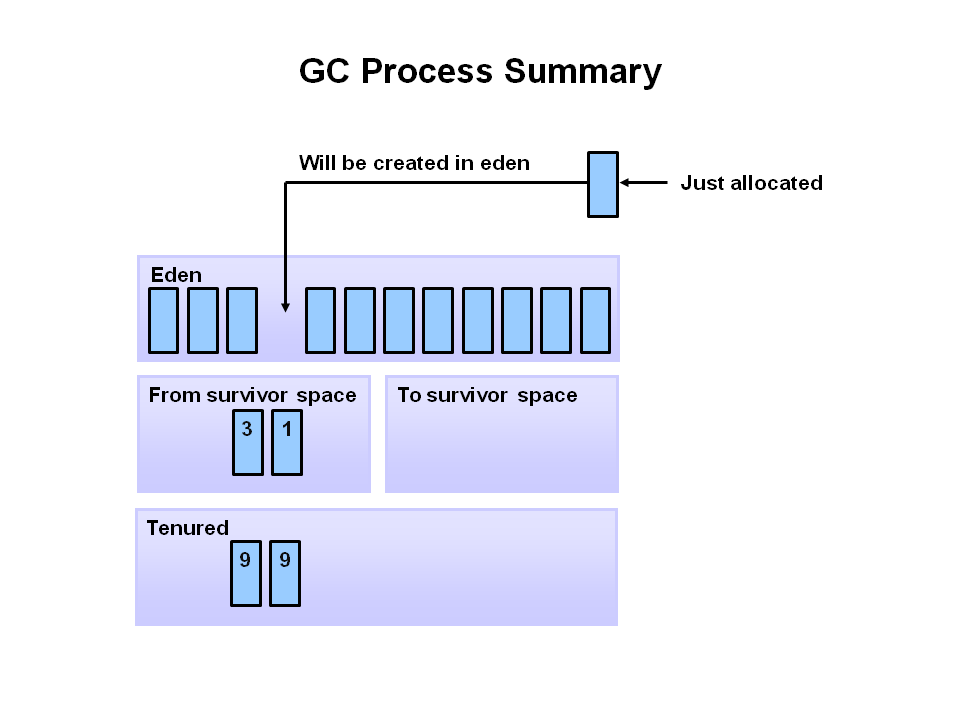
This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.



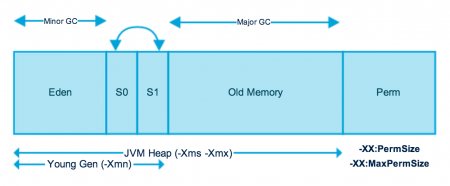
As minor GCs continue to occure objects will continue to be promoted to the old generation space.



So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.



####################################################################################



At first the good news: [Java SE 6.0](http://www.oracle.com/technetwork/java/javase/tech/index.html) is doing a wonderful job in picking the right options for a system based on the available system resources. The entire tuning of the various memory pools and picking of the right garbage collectors becomes significant less important compared to older JVM versions.

There is a lot of literature on the web. A good starting point are the [SDN performance pages](http://java.sun.com/performance/reference/whitepapers/tuning.html) .

Assuming that you don't have the time to read everything in detail. Here's a 30 minute animated

overview using [visualgc](http://java.sun.com/performance/jvmstat/) from the freely available jvmstat package. [jconsole](http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html) (part of SE6.0) will shown as well.

To keep a 10 year garbage collection history short:

The Sun Hotspot VM consist of a number of memory pools which need to be large enough for a given application:

* Young generation (named as well Eden space): New objects will be allocated in the memory pool. The assumption is that most object get dereferenced and become unreachable soon after their creation. Objects not being dereferenced will be copied by the new generation garbage collector into the survivor spaces. They may get copied n some special cases directly into the old generation pool.
* Survivor spaces: These two small spaces keep the surviving objects of a young generation garbage collection. Surviving objects will be copied for a (small) number of times from one survivor into the other. This allows to harvest our more dereferenced objects.
* Old generation: The largest memory pool which should keep the long living objects. Objects are getting copied into this pool once they leave the survivor spaces.
* Permament generation: This fairly unknown pool keeps the information of all the classes. It doesn't need any attention for most applications. It may need to be adapted for some applications with many classes. It may need some attentention as well if the application permanently loads and unloads classes.

The entire tuning of these pools is based on some view principles:

* The new generation should be large enough to keep all temporary objects until they are getting dereferenced.
* The old generation should host all permanent objects
* Young generation garbage collections (GC) are unavoidable. Their run time typically grows with the number of objects.
* The young generation GC is typically cheap compared to the old generation GC
* Old generation GC should be avoided. They're needed to clean up objects which aren't permanent, yet some always slip through the young generation GC.

The typical strategy for users who can't rely on the the auto adaptive sizing options is the following:

* Increase the young generation until the runtime of the GC becomes unacceptable or all temporary objects can be cleaned up in the new generation.
* Increase the old generation until around at least 50% remain free after a GC or the runtime of the GC run times becomes unacceptable.

The best way to get a feeling for these decisions is to get a quantitative understanding of the application by attaching visualgc or a jconsole. The other way to monitor an application is adding the option -verbose:gc to a VM which will force the VM to prompt all garbage consoles on console.

The examples used here is Specjbb, an application which is growing it's data set in three steps every minute.

The examples are being run on a two processor Sun Blade 2000. There a number of things to look for on the screen:

* a graphical CPU monitor (sdtprocess) is running in the left upper corner. Blue means the first CPU is getting used. The red range indicates that the second CPU is getting used. Missing pixels means that CPU cycles aren't getting used. Single threaded garbage collections for the VM to use one processor for the GC leaving the other processor unused. The goal is to keep the CPUs of the system busy as much as possible.
* Visualgc will show the sizes of all pools and how the fill up over time
* The console output from -verbose:gc will show from time to time “Full GC”. This is a last resort for the VM to clean up objects. They should be avoided whenever possible. The parallel new (generation) GC and the mostly concurrent GC (CMS) will only switch back to full GC when they are in trouble.

## Memory Options:

XX:MaxNewSize=size : Maximum size of new generation (in bytes). Since 1.4, MaxNewSize is computed as a function of NewRatio.

XX:NewSize=2m : Default size of new generation (in bytes)

-XX:SurvivorRatio=8: Ratio of eden/survivor space size

-XX:MaxPermSize=64m : Size of the Permanent Generation.

-Xms180m -Xmx180m : Oracle recommends setting the minimum heap size (**-Xms)** equal to the maximum heap size (**-Xmx)** to minimize garbage collections

-XX:SurvivorRatio : The SurvivorRatio parameter controls the size of the two survivor spaces. For example, -XX:SurvivorRatio=6 sets the ratio between each survivor space and eden to be 1:6, each survivor space will be one eighth of the young generation. The default for Solaris is 32. If survivor spaces are too small, copying collection overflows directly into the old generation. If survivor spaces are too large, they will be empty. At each GC, the JVM determines the number of times an object can be copied before it is tenured, called the tenure threshold. This threshold is chosen to keep the survivor space half full.

Use the option -XX:+PrintTenuringDistribution to show the threshold and ages of the objects in the new generation. It is useful for observing the lifetime distribution of an application.

So the -XX:SurvivorRatio is ignored unless you use -XX-UseAdaptiveSizePolicy

## The Young Generation

After total available memory, the second most influential factor affecting garbage collection performance is the proportion of the heap dedicated to the young generation. The bigger the young generation, the less often minor collections occur. However, for a bounded heap size, a larger young generation implies a smaller tenured generation, which will increase the frequency of major collections. The optimal choice depends on the lifetime distribution of the objects allocated by the application.

By default, the young generation size is controlled by the parameter NewRatio. For example, setting -XX:**NewRatio**=3 means that the ratio between the young and tenured generation is 1:3. In other words, the combined size of the eden and survivor spaces will be one-fourth of the total heap size.

The parameters NewSize and MaxNewSize bound the young generation size from below and above. Setting these to the same value fixes the young generation, just as setting -Xms and -Xmx to the same value fixes the total heap size. This is useful for tuning the young generation at a finer granularity than the integral multiples allowed by NewRatio.

### Survivor Space Sizing

You can use the parameter SurvivorRatio can be used to tune the size of the survivor spaces, but this is often not important for performance. For example, -XX:SurvivorRatio=6 sets the ratio between eden and a survivor space to 1:6. In other words, each survivor space will be one-sixth the size of eden, and thus one-eighth the size of the young generation (not one-seventh, because there are two survivor spaces).

If survivor spaces are too small, copying collection overflows directly into the tenured generation. If survivor spaces are too large, they will be uselessly empty. At each garbage collection, the virtual machine chooses a threshold number, which is the number times an object can be copied before it is tenured. This threshold is chosen to keep the survivors half full. The command line option -XX:+PrintTenuringDistribution (not available on all garbage collectors) can be used to show this threshold and the ages of objects in the new generation. It is also useful for observing the lifetime distribution of an application.

+++++++++++++++++++\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_++++++++++++++============================

**Example 1: An undersized VM with not enough memory**

The first example shows a sequence where a fixed 60MB new generation and a fixed 120MB old generation are by far to small to host the application. Many objects are spilling over from the young generation. The old generation will fill up quickly. The system will spend most of the time with inefficient garbage collections. A visual GC pattern like below is an application which is in deep trouble. The application is getting interrupted very often by the garbage collector and the garbage collector is not able to free to free enough memory. The GC is a kind of busy all the time without being able to release enough memory.

* The example: [A 3 minute flash video pulled from the screen](http://www.scalingbits.com/system/files/lauf1.html).
* The options used are: -XX:MaxNewSize=60m -XX:NewSize=60M -Xms180m -Xmx180m -XX:SurvivorRatio=10

**Example 2: A VM with an increased old generation and a to small young generation**

The second example shows a VM which has an increased old generation of 320MB. The permanent full GCs are now gone. The system isn't trashing anymore. The new generation is still not able to keep the temporary objects. Many temporary objects are spiling over into the old generation. The filling degree of the old generation goes up. The system is now able to use the two processors for most of the time. Full GCs can be recognized by the sharp decrease in old generation filling level. The application is blocked through the full GCs. This means hat all in flight transactions for user are paused. The response times will be very bad while a full GC os gettin processed.

* The example: [A 3 minute flash video pulled from the screen](http://www.scalingbits.com/system/files/lauf2.html).
* The options used are: -XX:MaxNewSize=60m -XX:NewSize=60M -Xms320m -Xmx320m

**Example 3: A VM setup with parallel new generation and mostly concurrent garbage collections**

The third example is now enabling a parallel new generation GC which is able to use both processors at a time and the mostly concurrent GC for the old generation. This is the typical combination which will work well with interactive systems. The parallel new GC will shorten the interruptions to the application by using all processors. The mostly concurrent GC will use a thread in the back ground to do the job. The second processor is still available to the application. It remains however a sort single threaded period. Visualgc is documenting this algorithm by an old heap which is slowly decreasing while the application still progresses. The examples however shows what happens if the old generation becomes overcrowded towards the end of the run. The mostly concurrent GC leaves a fragmented memory space. It can' compact the heap. It has to fall back to a compacting full GC. Monitor this event in your application. You can't avoid it completley. It should be a rare event (a few times a day).

* The example: [A 3 minute flash video pulled from the screen](http://www.scalingbits.com/system/files/lauf3.html).
* The options used are: -XX:MaxNewSize=60m -XX:NewSize=60M -Xms320m -Xmx320m -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled

**Example 4: A VM sized as before yet with an increased young generation**

The fourth example fixes the problems of the over spilling objects from the young generation by increasing to 120MB. The video shows how this strategy works for the first third of the run. The object creation rate increases through the second and the third phase. The old generation starts filling up again. Old generation GCs are kicking in in the later phases.

This increased load is a typical issue of a real application. The new generation needs to be adapted with increasing load. he reason is:

* The new generation pool can keep a certain number of objects
* Objects have their average live time

Creating more objects per time (increasing load, more users etc.) means that there is less space. Less space means the objects is copied in the (expensive to clean up) old generation pool earlier. The old GC has overproportional work to do since there are dereferenced objects which didn't show up before and the load increased in parallel.

* The example: [A 3 minute flash video pulled from the screen](http://www.scalingbits.com/system/files/lauf4.html).
* The options used are: -XX:MaxNewSize=120m -XX:NewSize=120m -Xms320m -Xmx320m -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled

**Example 5: A well sized VM with increased threshold levels for survivor space with a short jconsole demo**

The last example is now fixing this problem as well. One option would have been to increase the young generation furthermore. A different solution is to increase the maximum allowed filling degree of the survivor spaces. Setting this value to 90% warrants for this application that all temporary objects get unreferenced within 4 young generation garbage collections. The things to watch out in the video below are:

The lowest window shows the survivor histograms. The surviving objects in the second and third generation become visible in this demo.

* Old generation Gcs: There are no more Gcs after the system stabilized.
* The example: [A 5 minute flash video pulled from the screen](http://www.scalingbits.com/system/files/lauf5.html).
* The options used are: -XX:MaxNewSize=120m -XX:NewSize=120m -Xms320m -Xmx320m -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:TargetSurvivorRatio=90

From other website :

### Permanent Generation

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

### Method Area

Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

### Memory Pool

Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.

### Runtime Constant Pool

Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is the part of method area.

### Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. You should read [Difference between Stack and Heap Memory](http://www.journaldev.com/4098/java-heap-memory-vs-stack-memory-difference).

### Java Heap Memory Switches

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

|  |  |
| --- | --- |
| VM Switch | VM Switch Description |
| -Xms | For setting the initial heap size when JVM starts |
| -Xmx | For setting the maximum heap size. |
| -Xmn | For setting the size of the Young Generation, rest of the space goes for Old Generation. |
| -XX:PermGen | For setting the initial size of the Permanent Generation memory |
| -XX:MaxPermGen | For setting the maximum size of Perm Gen |
| -XX:SurvivorRatio | For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8. |
| -XX:NewRatio | For providing ratio of old/new generation sizes. The default value is 2. |

Most of the times, above options are sufficient, but if you want to check out other options too then please check [JVM Options Official Page](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html).

### Java Garbage Collection

Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space to be allocated to objects created in the future processing. One of the best feature of java programming language is the **automatic garbage collection**, unlike other programming languages such as C where memory allocation and deallocation is a manual process.

**Garbage Collector** is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program. All these unreferenced objects are deleted and space is reclaimed for allocation to other objects.

One of the basic way of garbage collection involves three steps:

1. **Marking**: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.
2. **Normal Deletion**: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.
3. **Deletion with Compacting**: For better performance, after deleting unused objects, all the survived objects can be moved to be together. This will increase the performance of allocation of memory to newer objects.

**There are two problems with simple mark and delete approach.**

1. **First one is that it’s not efficient because most of the newly created objects will become unused**
2. **Secondly objects that are in-use for multiple garbage collection cycle are most likely to be in-use for future cycles too.**

**The above shortcomings with the simple approach is the reason that Java Garbage Collection is Generational and we have Young Generation and Old Generation spaces in the heap memory. I have already explained above how objects are scanned and moved from one generational space to another based on the Minor GC and Major GC.**

### Java Garbage Collection Types

There are five types of garbage collection types that we can use in our applications. We just need to use JVM switch to enable the garbage collection strategy for the application. Let’s look at each of them one by one.

1. **Serial GC (-XX:+UseSerialGC)**: Serial GC uses the simple **mark-sweep-compact** approach for young and old generations garbage collection i.e Minor and Major GC.

Serial GC is useful in client-machines such as our simple stand alone applications and machines with smaller CPU. It is good for small applications with low memory footprint.

1. **Parallel GC (-XX:+UseParallelGC)**: Parallel GC is same as Serial GC except that is spawns N threads for young generation garbage collection where N is the number of CPU cores in the system. We can control the number of threads using -XX:ParallelGCThreads=n JVM option.

Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses single thread for Old Generation garbage collection.

1. **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
2. **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for Old generation. CMS collector tries to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

CMS collector on young generation uses the same algorithm as that of the parallel collector. This garbage collector is suitable for responsive applications where we can’t afford longer pause times. We can limit the number of threads in CMS collector using -XX:ParallelCMSThreads=n JVM option.

1. **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and it’s long term goal is to replace the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

Gar\bage First Collector doesn’t work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”. You can find more details about it at [Garbage-First Collector Oracle Documentation](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html).

Hence it is better to use:  
1.cms collector when  
-you have high number of cpus  
-your application demands short pauses  
-you have more memory  
  
2.parallel collector when  
-you have less number of cpus  
-your application demands throughput and can withstand recurring long pauses  
-you have less memory

If you have a single processor, single thread machine then you should use the serial collector (default for some configurations, can be enabled explicitly for with -XX:+UseSerialGC).

For multiprocessor machines where your workload is basically CPU bound, use the parallel collector. This is enabled by default if you use the -server flag, or you can enable it explicitly with -XX:+UseParallelGC.

If you'd rather keep the GC pauses shorter at the expense of using more total CPU time for GC, and you have more than one CPU, you can use the concurrent collector (-XX:+UseConcMarkSweepGC). Note that the concurrent collector tends to require more RAM allocated to the JVM than the serial or parallel collectors for a given workload because some memory fragmentation can occur.

### Java Garbage Collection Monitoring

We can use Java command line as well as UI tools for monitoring garbage collection activities of an application. For my example, I am using one of the demo application provided by Java SE downloads.

If you want to use the same application, go to [Java SE Downloads](http://www.oracle.com/technetwork/java/javase/downloads/index.html) page and download **JDK 7 and JavaFX Demos and Samples**. The sample application I am using is **Java2Demo.jar** and it’s present injdk1.7.0\_55/demo/jfc/Java2D directory. However this is an optional step and you can run the GC monitoring commands for any java application.

Command used by me to start the demo application is:

|  |  |
| --- | --- |
| 1 | pankaj@Pankaj:~/Downloads/jdk1.7.0\_55/demo/jfc/Java2D$ java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar Java2Demo.jar |

#### jsat

We can use jstat command line tool to monitor the JVM memory and garbage collection activities. It ships with standard JDK, so you don’t need to do anything else to get it.

For executing jstat you need to know the process id of the application, you can get it easily using ps -eaf | grep java command.

|  |  |
| --- | --- |
| 1  2  3 | pankaj@Pankaj:~$ ps -eaf | grep Java2Demo.jar    501 9582  11579   0  9:48PM ttys000    0:21.66 /usr/bin/java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseG1GC -jar Java2Demo.jar    501 14073 14045   0  9:48PM ttys002    0:00.00 grep Java2Demo.jar |

So the process id for my java application is 9582. Now we can run **jstat** command as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | pankaj@Pankaj:~$ jstat -gc 9582 1000   S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC    FGCT     GCT  1024.0 1024.0  0.0    0.0    8192.0   7933.3   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654  1024.0 1024.0  0.0    0.0    8192.0   8026.5   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654  1024.0 1024.0  0.0    0.0    8192.0   8030.0   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654  1024.0 1024.0  0.0    0.0    8192.0   8122.2   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654  1024.0 1024.0  0.0    0.0    8192.0   8171.2   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654  1024.0 1024.0  48.7   0.0    8192.0   106.7    42108.0    23401.3   20480.0 19990.9    158    0.275  40      1.381    1.656  1024.0 1024.0  48.7   0.0    8192.0   145.8    42108.0    23401.3   20480.0 19990.9    158    0.275  40      1.381    1.656 |

The last argument for jstat is the time interval between each output, so it will print memory and garbage collection data every 1 second.

Let’s go through each of the columns one by one.

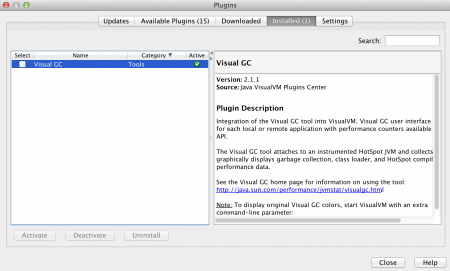
* **S0C and S1C**: This column shows the current size of the Survivor0 and Survivor1 areas in KB.
* **S0U and S1U**: This column shows the current usage of the Survivor0 and Survivor1 areas in KB. Notice that one of the survivor areas are empty all the time.
* **EC and EU**: These columns show the current size and usage of Eden space in KB. Note that EU size is increasing and as soon as it crosses the EC, Minor GC is called and EU size is decreased.
* **OC and OU**: These columns show the current size and current usage of Old generation in KB.
* **PC and PU**: These columns show the current size and current usage of Perm Gen in KB.
* **YGC and YGCT**: YGC column displays the number of GC event occurred in young generation. YGCT column displays the accumulated time for GC operations for Young generation. Notice that both of them are increased in the same row where EU value is dropped because of minor GC.
* **FGC and FGCT**: FGC column displays the number of Full GC event occurred. FGCT column displays the accumulated time for Full GC operations. Notice that Full GC time is too high when compared to young generation GC timings.
* **GCT**: This column displays the total accumulated time for GC operations. Notice that it’s sum of YGCT and FGCT column values.

The advantage of **jstat** is that it can be executed in remote servers too where we don’t have GUI. Notice that sum of S0C, S1C and EC is 10m as specified through -Xmn10m JVM option.

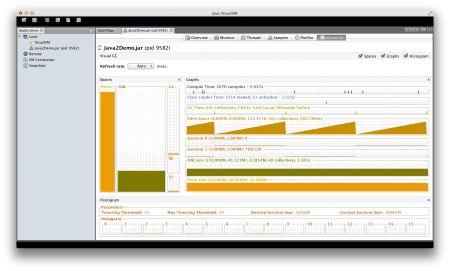
#### Java VisualVM with Visual GC

If you want to see memory and GC operations in GUI, then you can use jvisualvm tool. Java VisualVM is also part of JDK, so you don’t need to download it separately.

Just run jvisualvm command in the terminal to launch the Java VisualVM application. Once launched, you need to install **Visual GC** plugin from Tools -< Plugins option, as shown in below image.

[](http://www.journaldev.com/wp-content/uploads/2014/05/VisualVM-Visual-GC-Plugin.png)

After installing **Visual GC**, just open the application from the left side column and head over to **Visual GC**section. You will get an image of JVM memory and garbage collection details as shown in below image.

[](http://www.journaldev.com/wp-content/uploads/2014/05/Serial-GC-VisualGC.png)

### Java Garbage Collection Tuning

**Java Garbage Collection Tuning** should be the last option you should use for increasing the throughput of your application and only when you see drop in performance because of longer GC timings causing application timeout.

If you see java.lang.OutOfMemoryError: PermGen space errors in logs, then try to monitor and increase the Perm Gen memory space using -XX:PermGen and -XX:MaxPermGen JVM options. You might also try using -XX:+CMSClassUnloadingEnabled and check how it’s performing with CMS Garbage collector.

If you are see a lot of Full GC operations, then you should try increasing Old generation memory space.

Overall garbage collection tuning takes a lot of effort and time and there is no hard and fast rule for that. You would need to try different options and compare them to find out the best one suitable for your application.

That’s all for Java Memory Model and Garbage Collection, I hope it helps you in understanding JVM memory and garbage collection process.

From Oracle office website

The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads. Normally the concurrent low pause collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap.

**Note:** CMS collector on young generation uses the same algorithm as that of the parallel collector.

#### CMS Collection Phases

The CMS collector performs the following phases on the old generation of the heap:

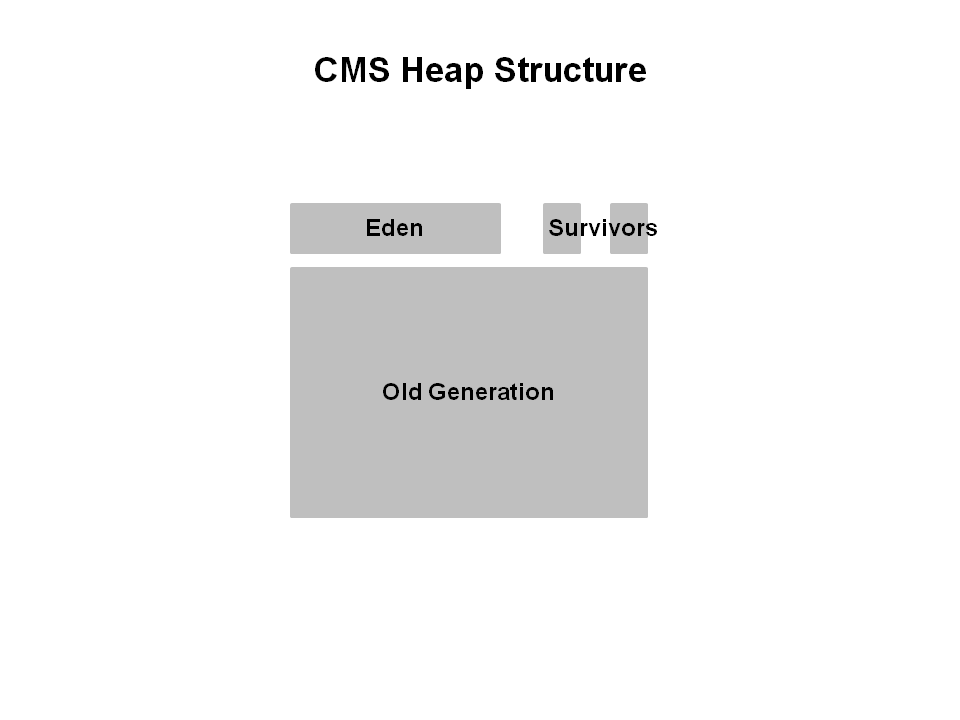
| **Phase** | **Description** |
| --- | --- |
| (1) Initial Mark *(Stop the World Event)* | Objects in old generation are “marked” as reachable including those objects which may be reachable from young generation. Pause times are typically short in duration relative to minor collection pause times. |
| (2) Concurrent Marking | Traverse the tenured generation object graph for reachable objects concurrently while Java application threads are executing. Starts scanning from marked objects and transitively marks all objects reachable from the roots. The mutators are executing during the concurrent phases 2, 3, and 5 and any objects allocated in the CMS generation during these phases (including promoted objects) are immediately marked as live. |
| (3) Remark *(Stop the World Event)* | Finds objects that were missed by the concurrent mark phase due to updates by Java application threads to objects after the concurrent collector had finished tracing that object. |
| (4) Concurrent Sweep | Collects the objects identified as unreachable during marking phases. The collection of a dead object adds the space for the object to a free list for later allocation. Coalescing of dead objects may occur at this point. Note that live objects are not moved. |
| (5) Resetting | Prepare for next concurrent collection by clearing data structures. |

#### Reviewing Garbage Collection Steps

Next, let's review CMS Collector operations step by step.

**Heap Structure for CMS Collector**

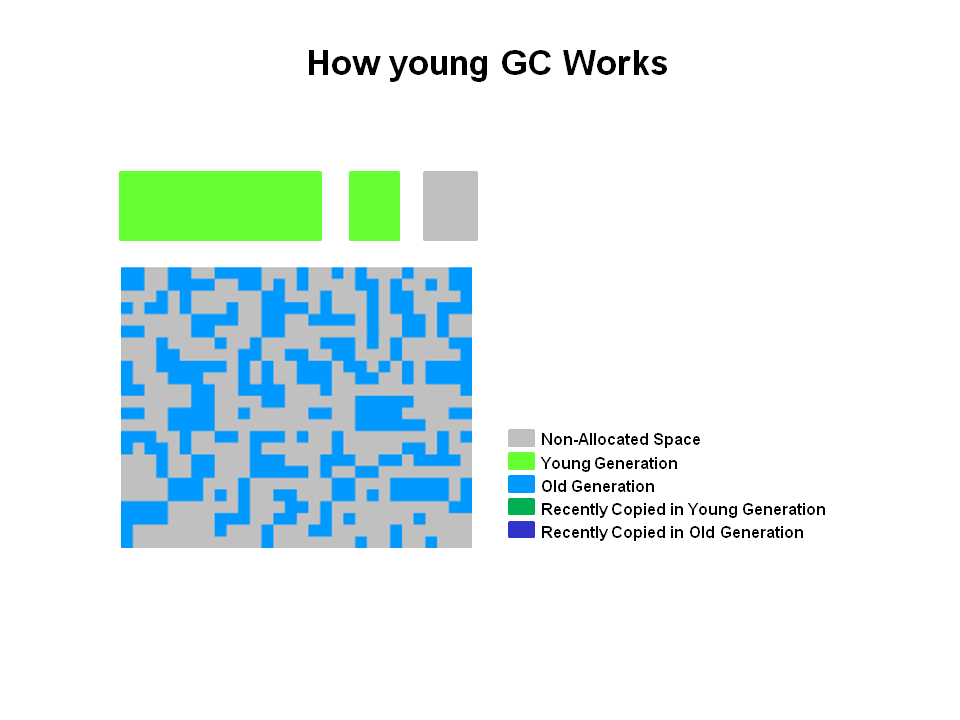
The heap is split into three spaces.



Young generation is split into Eden and two survivor spaces. Old generation is one contiguous space. Object collection is done in place. No compaction is done unless there is a full GC.

**How Young GC works in CMS**

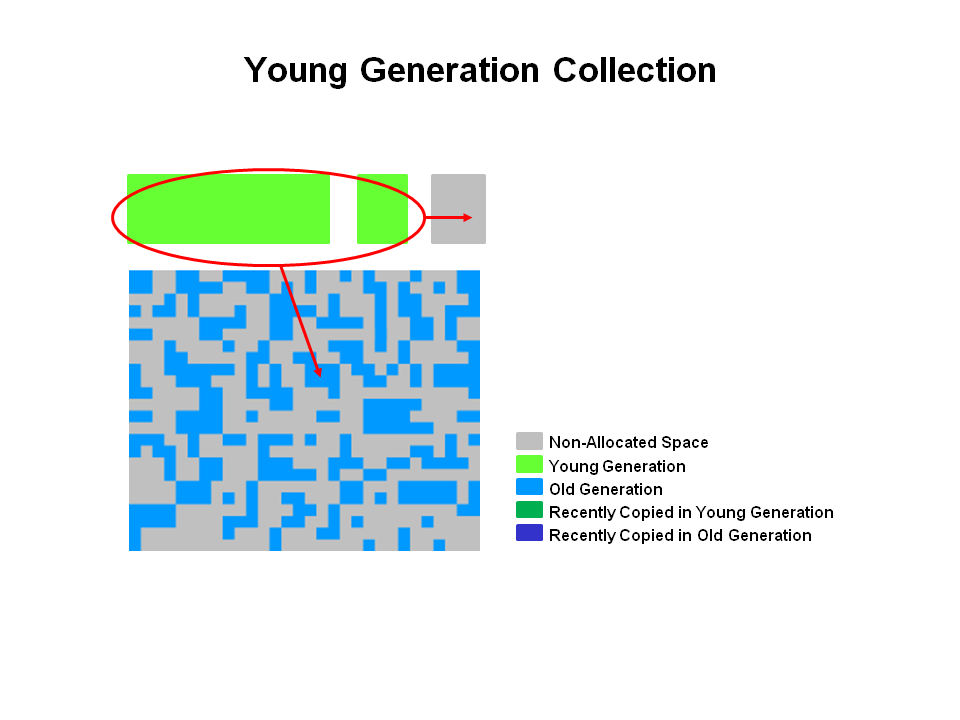
The young generation is colored light green and the old generation in blue. This is what the CMS might look like if your application has been running for a while. Objects are scattered around the old generation area.



With CMS, old generation objects are deallocated in place. They are not moved around. The space is not compacted unless there is a full GC.

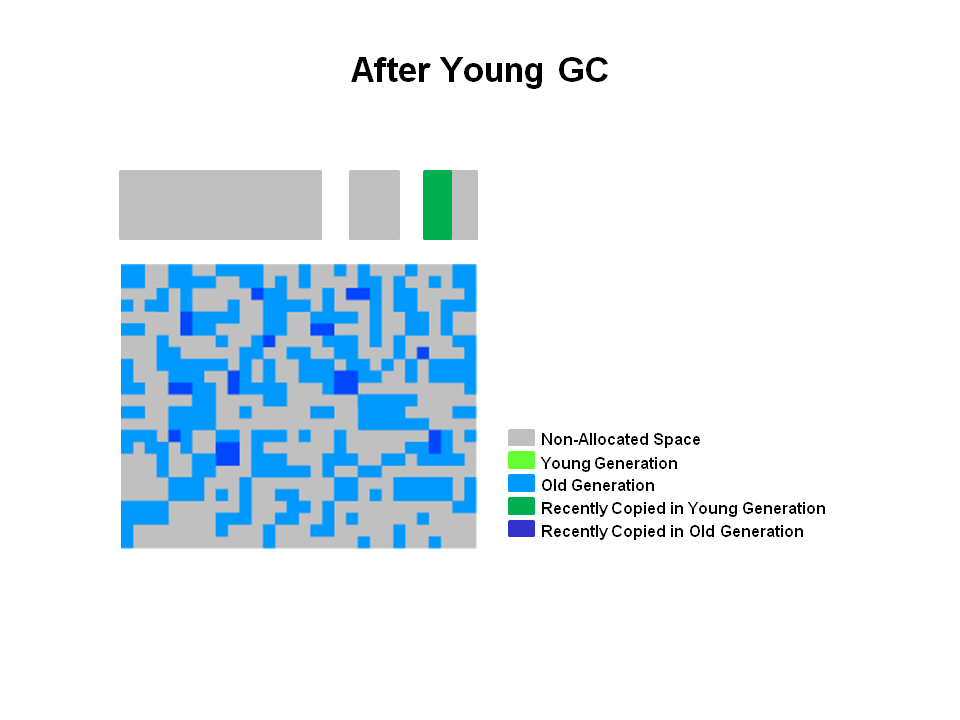
**Young Generation Collection**

Live objects are copied from the Eden space and survivor space to the other survivor space. Any older objects that have reached their aging threshold are promoted to old generation.



**After Young GC**

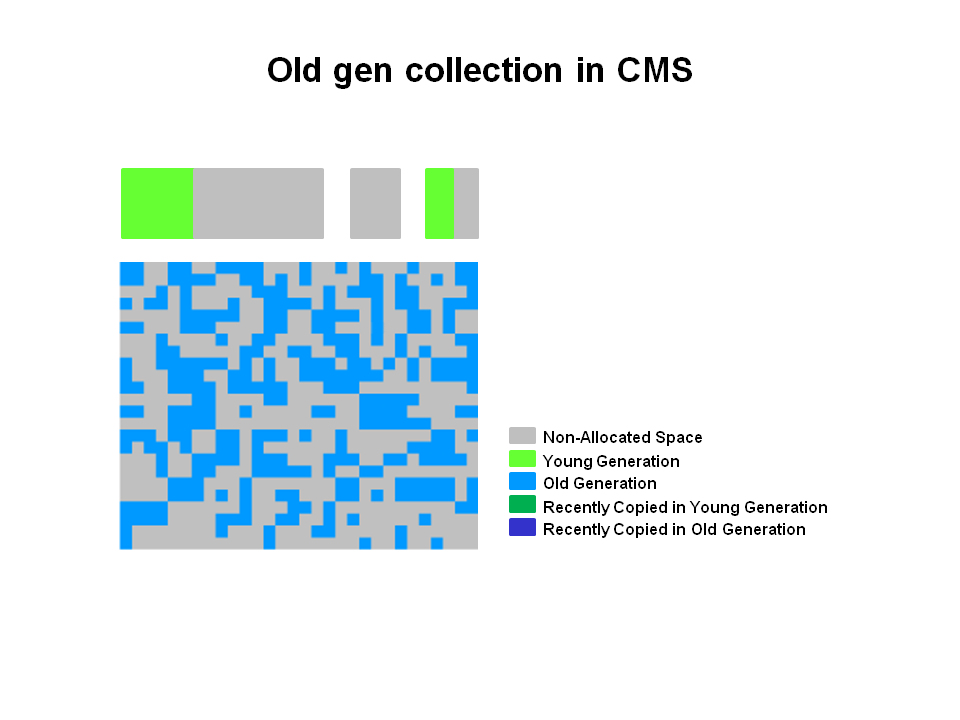
After a young GC, the Eden space is cleared and one of the survivor spaces is cleared.



Newly promoted objects are shown in dark blue on the diagram. The green objects are surviving young generation objects that have not yet been promoted to old generation.

**Old Generation Collection with CMS**

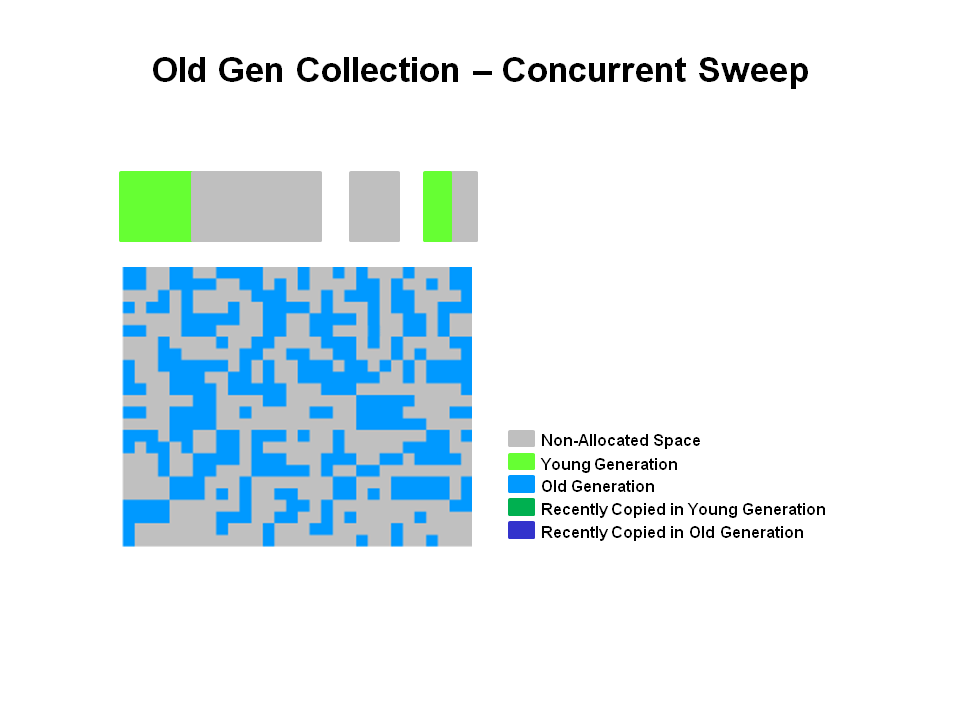
Two stop the world events take place: initial mark and remark. When the old generation reaches a certain occupancy rate, the CMS is kicked off.



(1) Initial mark is a short pause phase where live (reachable) objects are marked. (2) Concurrent marking finds live objects while the application continues to execute. Finally, in the (3) remark phase, objects are found that were missed during (2) concurrent marking in the previous phase.

**Old Generation Collection - Concurrent Sweep**

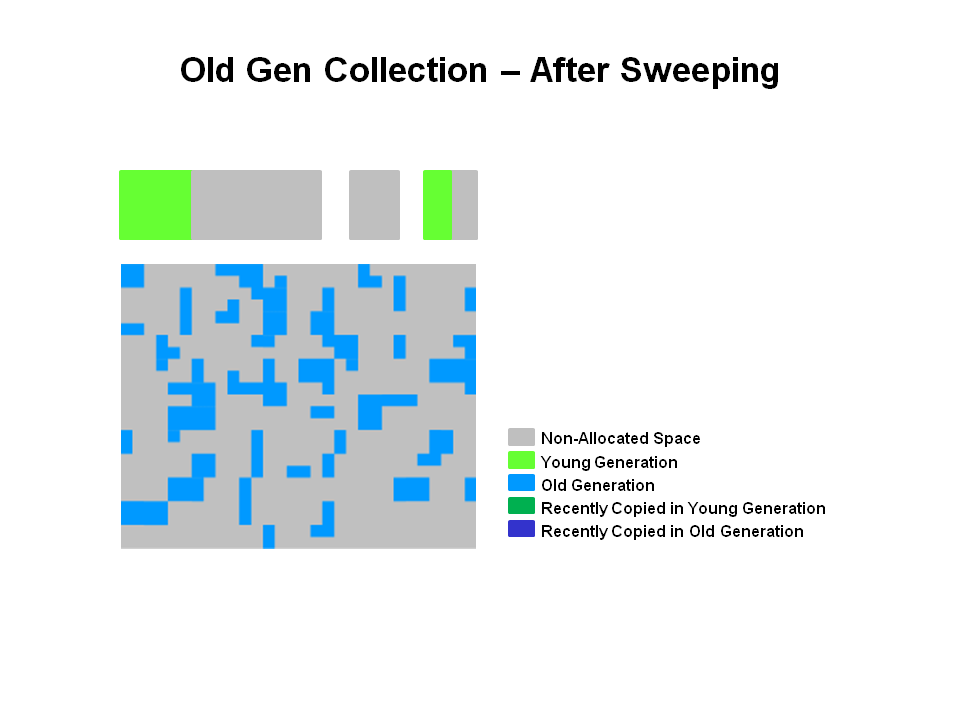
Objects that were not marked in the previous phase are deallocated in place. There is no compaction.



**Note:** Unmarked objects == Dead Objects

**Old Generation Collection - After Sweeping**

After the (4) Sweeping phase, you can see that a lot of memory has been freed up. You will also notice that no compaction has been done.



Finally, the CMS collector will move through the (5) resetting phase and wait for the next time the GC threshold is reached.

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*&&&&&&&&&&&$$$$$$$$$$$

# The Garbage-First Garbage Collector

## Introduction

The Garbage-First (G1) garbage collector is fully supported in Oracle JDK 7 update 4 and later releases. The G1 collector is a server-style garbage collector, targeted for multi-processor machines with large memories. It meets garbage collection (GC) pause time goals with high probability, while achieving high throughput. Whole-heap operations, such as global marking, are performed concurrently with the application threads. This prevents interruptions proportional to heap or live-data size.

## Technical description

The G1 collector achieves high performance and pause time goals through several techniques.

The heap is partitioned into a set of equal-sized heap regions, each a contiguous range of virtual memory. G1 performs a concurrent global marking phase to determine the liveness of objects throughout the heap. After the mark phase completes, G1 knows which regions are mostly empty. It collects in these regions first, which usually yields a large amount of free space. This is why this method of garbage collection is called Garbage-First. As the name suggests, G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, that is, garbage. G1 uses a pause prediction model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target.

The regions identified by G1 as ripe for reclamation are garbage collected using evacuation. G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory. This evacuation is performed in parallel on multi-processors, to decrease pause times and increase throughput. Thus, with each garbage collection, G1 continuously works to reduce fragmentation, working within the user defined pause times. This is beyond the capability of both the previous methods. CMS (Concurrent Mark Sweep ) garbage collection does not do compaction. ParallelOld garbage collection performs only whole-heap compaction, which results in considerable pause times.

It is important to note that G1 is not a real-time collector. It meets the set pause time target with high probability but not absolute certainty. Based on data from previous collections, G1 does an estimate of how many regions can be collected within the user specified target time. Thus, the collector has a reasonably accurate model of the cost of collecting the regions, and it uses this model to determine which and how many regions to collect while staying within the pause time target.  
  
For more further information about using and configuring G1 please see the [command line options](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html).

## Recommended Use Cases for G1

The first focus of G1 is to provide a solution for users running applications that require large heaps with limited GC latency. This means heap sizes of around 6GB or larger, and stable and predictable pause time below 0.5 seconds.

Applications running today with either the CMS or the ParallelOld garbage collector would benefit switching to G1 if the application has one or more of the following traits.

* More than 50% of the Java heap is occupied with live data.
* The rate of object allocation rate or promotion varies significantly.
* Undesired long garbage collection or compaction pauses (longer than 0.5 to 1 second)

## Future

G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS). Comparing G1 with CMS, there are differences that make G1 a better solution. One difference is that G1 is a compacting collector. G1 compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions. This considerably simplifies parts of the collector, and mostly eliminates potential fragmentation issues. Also, G1 offers more predictable garbage collection pauses than the CMS collector, and allows users to specify desired pause targets.

### The G1 Garbage Collector

The Garbage-First (G1) collector is a server-style garbage collector, targeted for multi-processor machines with large memories. It meets garbage collection (GC) pause time goals with a high probability, while achieving high throughput. *The G1 garbage collector is fully supported in Oracle JDK 7 update 4 and later releases*. The G1 collector is designed for applications that:

* + Can operate concurrently with applications threads like the CMS collector.
  + Compact free space without lengthy GC induced pause times.
  + Need more predictable GC pause durations.
  + Do not want to sacrifice a lot of throughput performance.
  + Do not require a much larger Java heap.

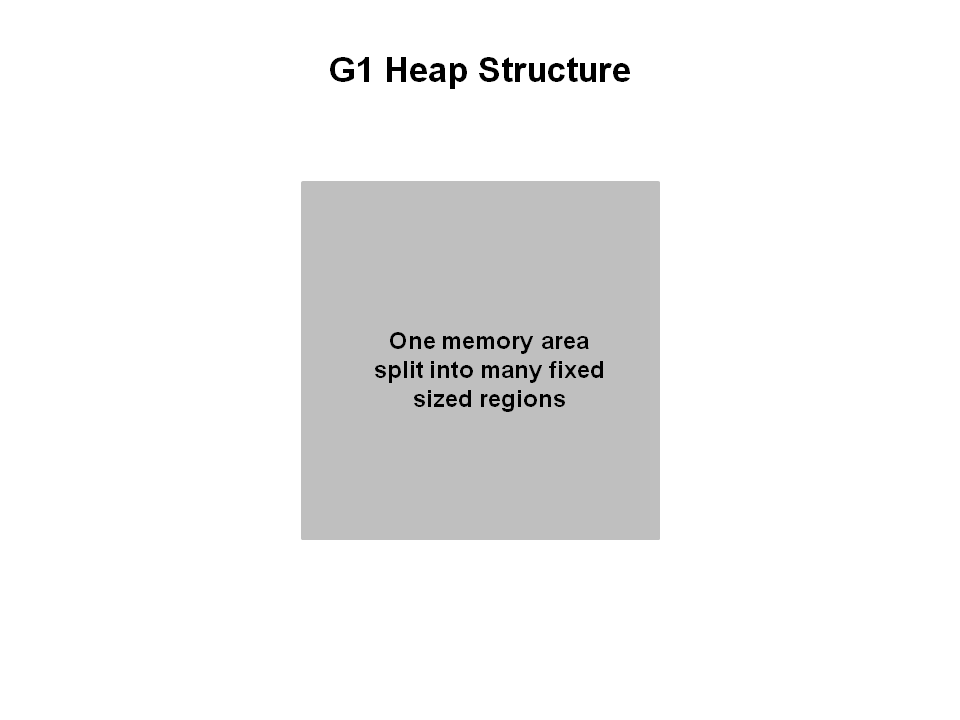
G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS). Comparing G1 with CMS, there are differences that make G1 a better solution. One difference is that G1 is a compacting collector. G1 compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions. This considerably simplifies parts of the collector, and mostly eliminates potential fragmentation issues. Also, G1 offers more predictable garbage collection pauses than the CMS collector, and allows users to specify desired pause targets.

### The G1 Garbage Collector Step by Step

The G1 collector takes a different approach to allocating the heap. The pictures that follow review the G1 system step by step.

**G1 Heap Structure**

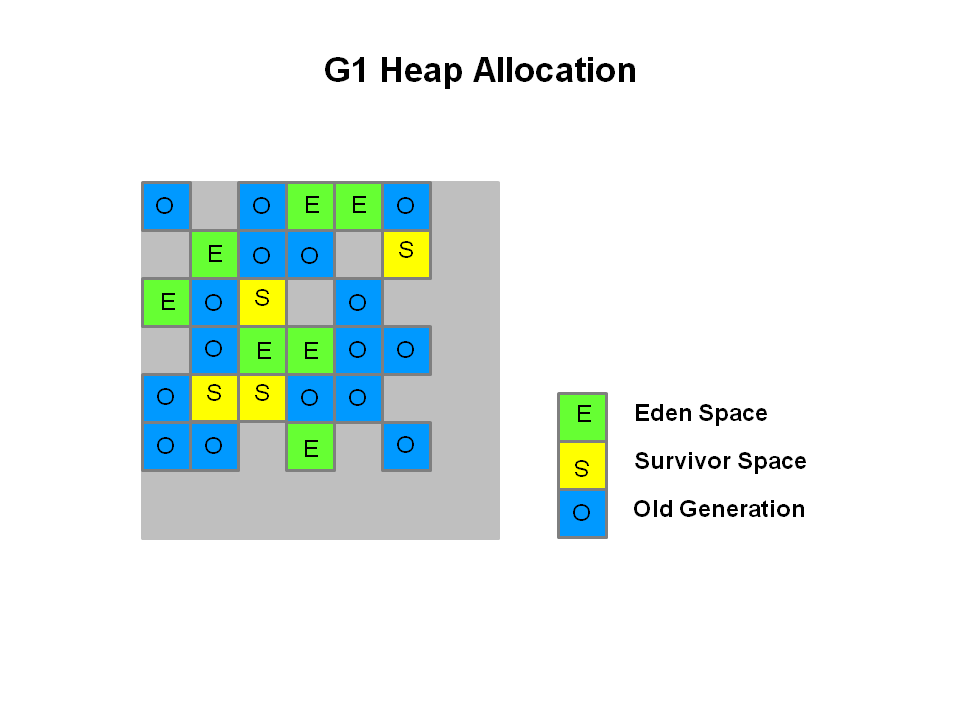
The heap is one memory area split into many fixed sized regions.



Region size is chosen by the JVM at startup. The JVM generally targets around 2000 regions varying in size from 1 to 32Mb.

**G1 Heap Allocation**

In reality, these regions are mapped into logical representations of Eden, Survivor, and old generation spaces.



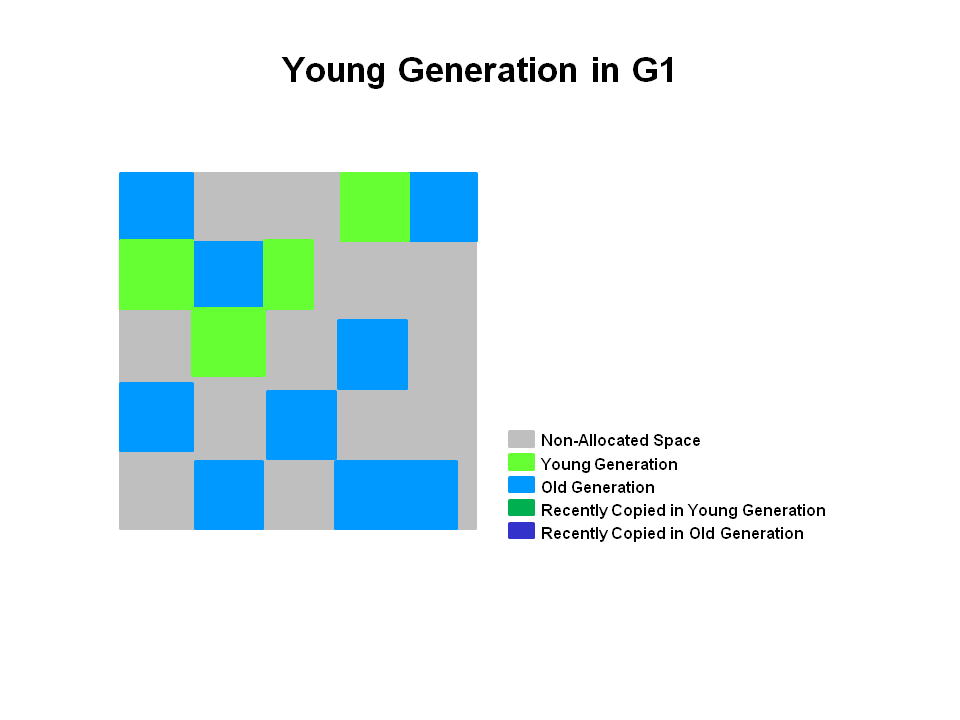
The colors in the picture shows which region is associated with which role. Live objects are evacuated (i.e., copied or moved) from one region to another. Regions are designed to be collected in parallel with or without stopping all other application threads.

As shown regions can be allocated into Eden, survivor, and old generation regions. In addition, there is a fourth type of object known as Humongous regions. These regions are designed to hold objects that are 50% the size of a standard region or larger. They are stored as a set of contiguous regions. Finally the last type of regions would be the unused areas of the heap.

**Note:** At the time of this writing, collecting humongous objects has not been optimized. Therefore, you should avoid creating objects of this size.

**Young Generation in G1**

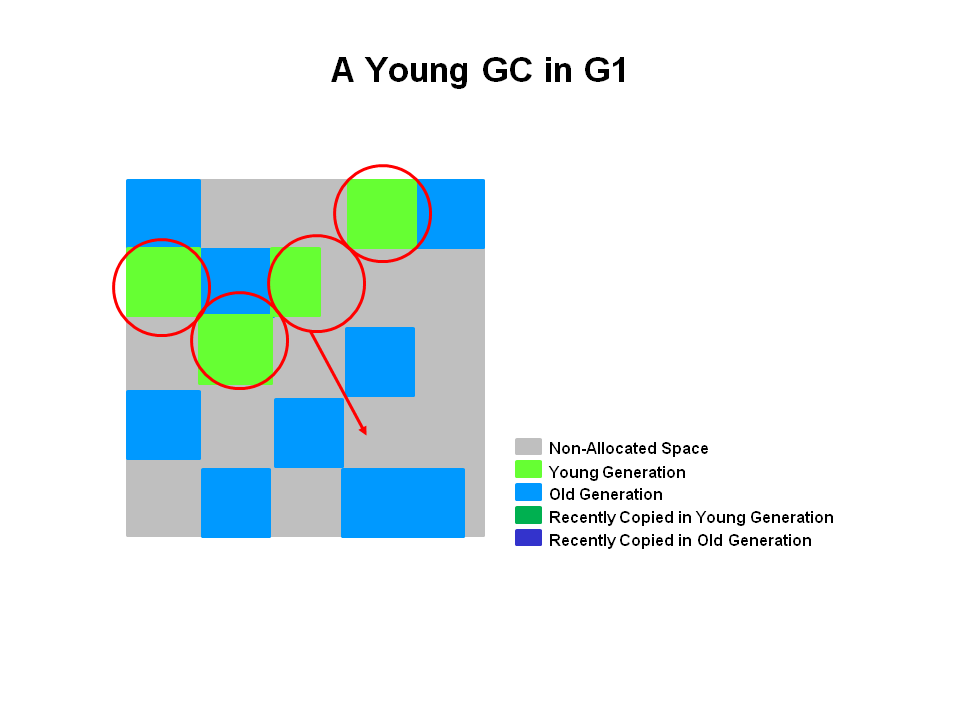
The heap is split into approximately 2000 regions. Minimum size is 1Mb and maximum size is 32Mb. Blue regions hold old generation objects and green regions hold young generation objects.



Note that the regions are not required to be contiguous like the older garbage collectors.

**A Young GC in G1**

Live objects are evacuated (i.e., copied or moved) to one or more survivor regions. If the aging threshold is met, some of the objects are promoted to old generation regions.

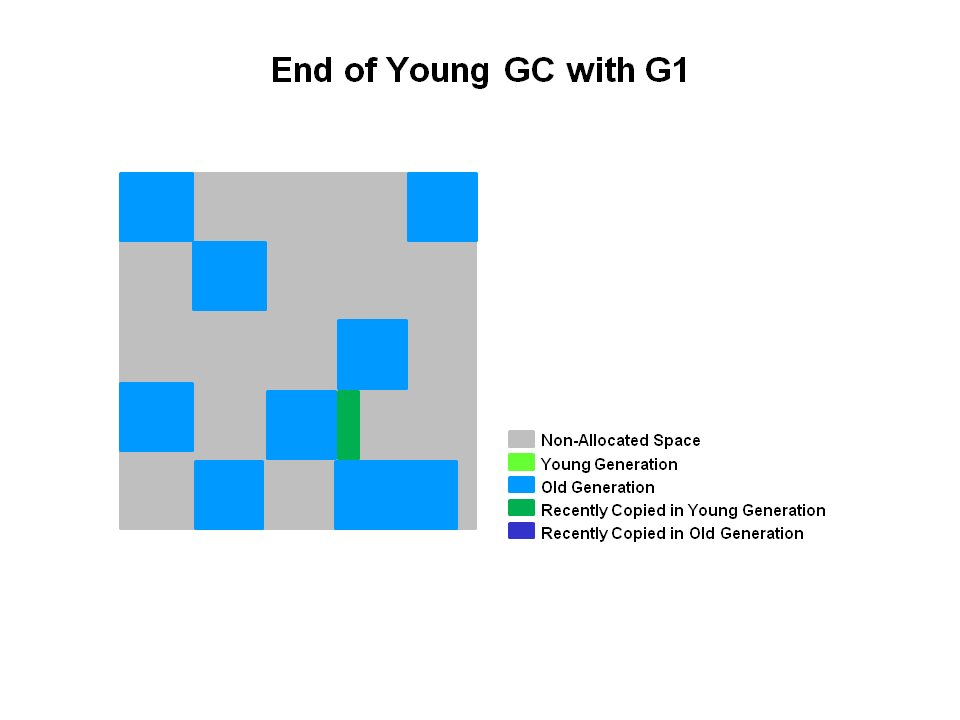


This is a stop the world (STW) pause. Eden size and survivor size is calculated for the next young GC. Accounting information is kept to help calculate the size. Things like the pause time goal are taken into consideration.

This approach makes it very easy to resize regions, making them bigger or smaller as needed.

**End of a Young GC with G1**

Live objects have been evacuated to survivor regions or to old generation regions.



Recently promoted objects are shown in dark blue. Survivor regions in green.

In summary, the following can be said about the young generation in G1:

* + The heap is a single memory space split into regions.
  + Young generation memory is composed of a set of non-contiguous regions. This makes it easy to resize when needed.
  + Young generation garbage collections, or young GCs, are stop the world events. All application threads are stopped for the operation.
  + The young GC is done in parallel using multiple threads.
  + Live objects are copied to new survivor or old generation regions.

### Old Generation Collection with G1

Like the CMS collector, the G1 collector is designed to be a low pause collector for old generation objects. The following table describes the G1 collection phases on old generation.

#### G1 Collection Phases - Concurrent Marking Cycle Phases

The G1 collector performs the following phases on the old generation of the heap. Note that some phases are part of a young generation collection.

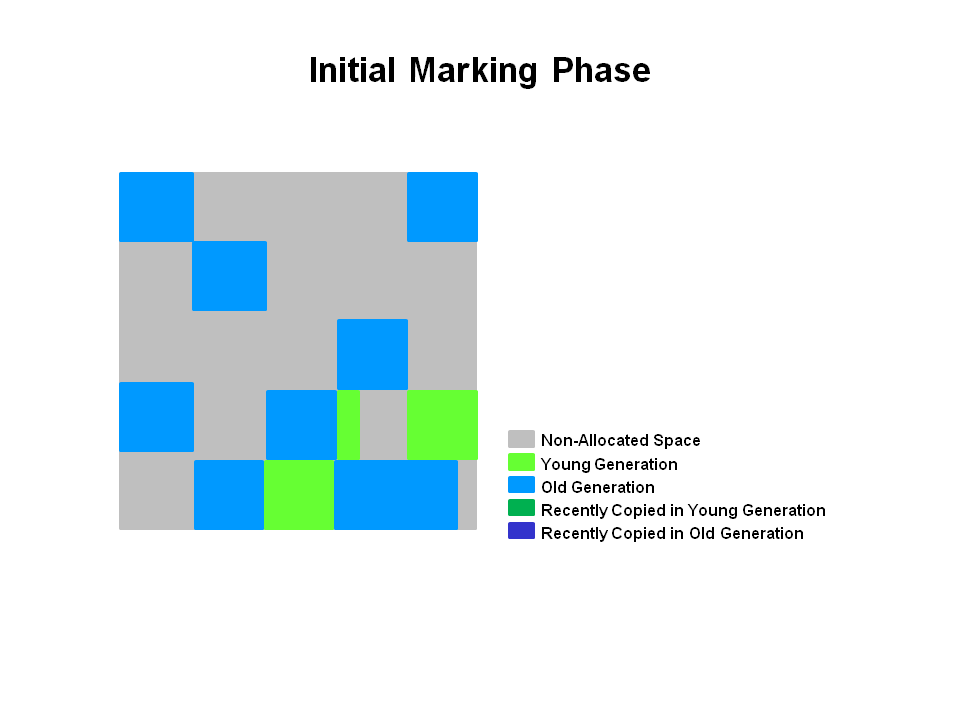
| **Phase** | **Description** |
| --- | --- |
| (1) Initial Mark *(Stop the World Event)* | This is a stop the world event. With G1, it is piggybacked on a normal young GC. Mark survivor regions (root regions) which may have references to objects in old generation. |
| (2) Root Region Scanning | Scan survivor regions for references into the old generation. This happens while the application continues to run. The phase must be completed before a young GC can occur. |
| (3) Concurrent Marking | Find live objects over the entire heap. This happens while the application is running. This phase can be interrupted by young generation garbage collections. |
| (4) Remark *(Stop the World Event)* | Completes the marking of live object in the heap. Uses an algorithm called snapshot-at-the-beginning (SATB) which is much faster than what was used in the CMS collector. |
| (5) Cleanup *(Stop the World Event and Concurrent)* | * + Performs accounting on live objects and completely free regions. (Stop the world)   + Scrubs the Remembered Sets. (Stop the world)   + Reset the empty regions and return them to the free list. (Concurrent) |
| (\*) Copying *(Stop the World Event)* | These are the stop the world pauses to evacuate or copy live objects to new unused regions. This can be done with young generation regions which are logged as [GC pause (young)]. Or both young and old generation regions which are logged as [GC Pause (mixed)]. |

#### G1 Old Generation Collection Step by Step

With the phases defined, let's look at how they interact with the old generation in the G1 collector.

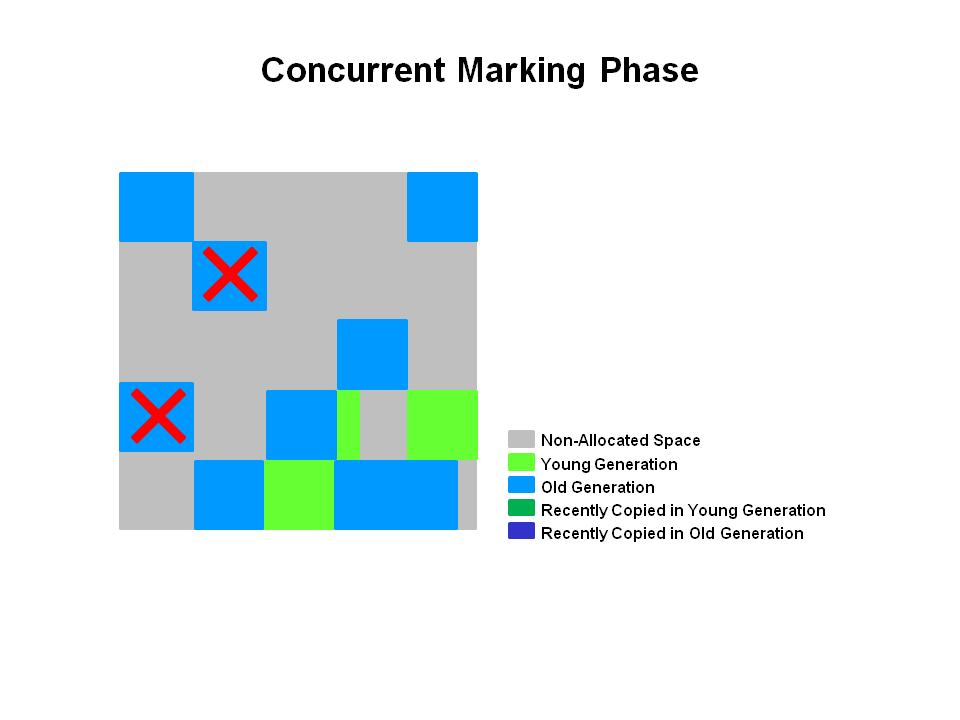
**Initial Marking Phase**

Initial marking of live object is piggybacked on a young generation garbage collection. In the logs this is noted as GC pause (young)(inital-mark).



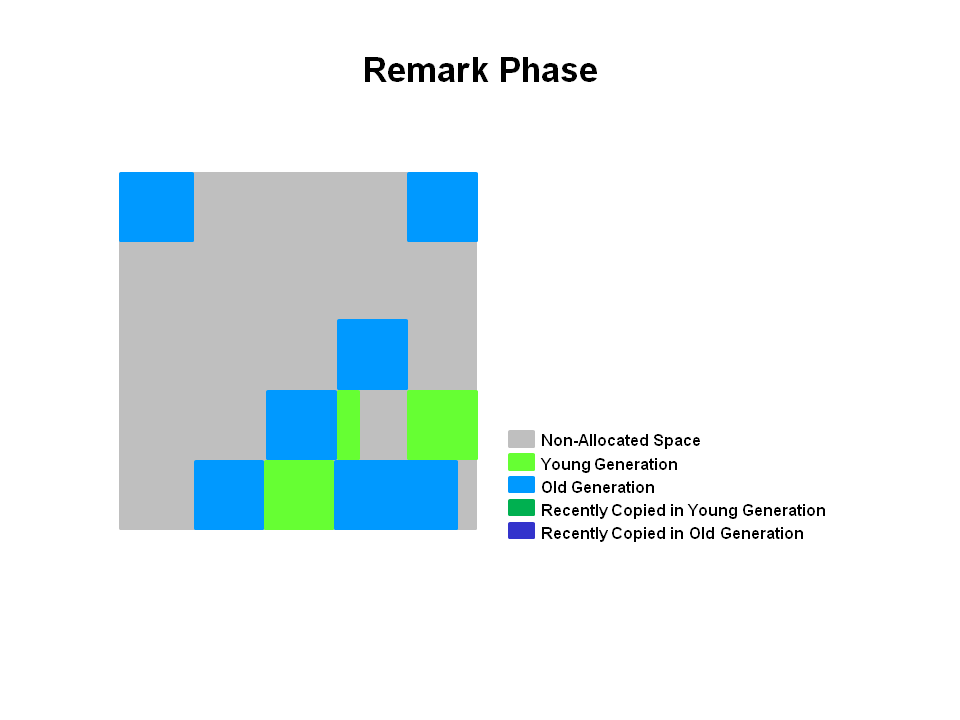
**Concurrent Marking Phase**

If empty regions are found (as denoted by the "X"), they are removed immediately in the Remark phase. Also, "accounting" information that determines liveness is calculated.



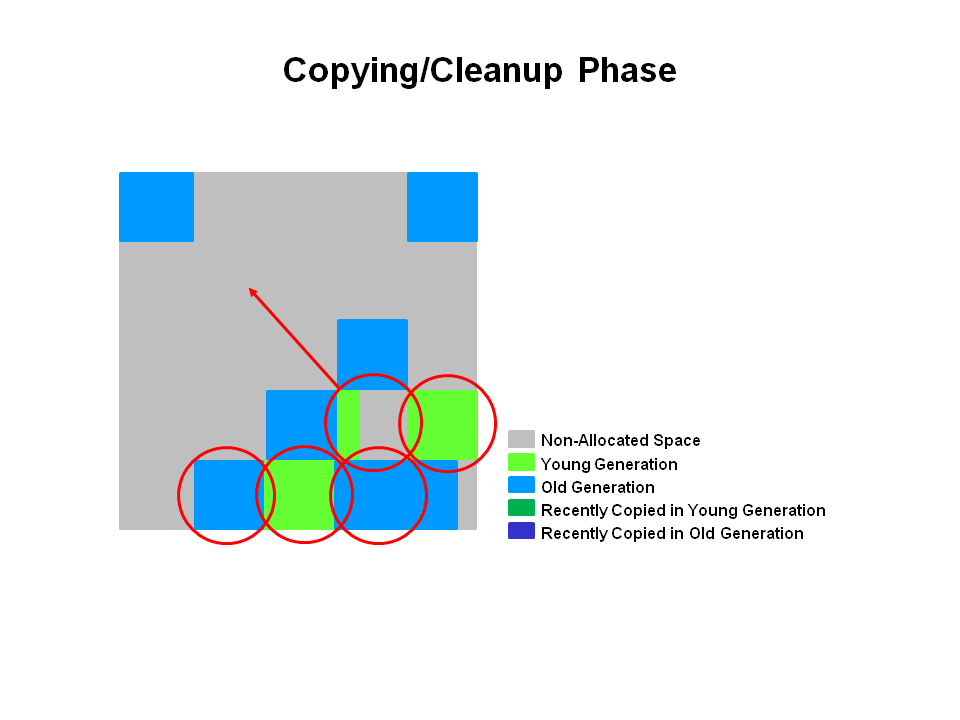
**Remark Phase**

Empty regions are removed and reclaimed. Region liveness is now calculated for all regions.



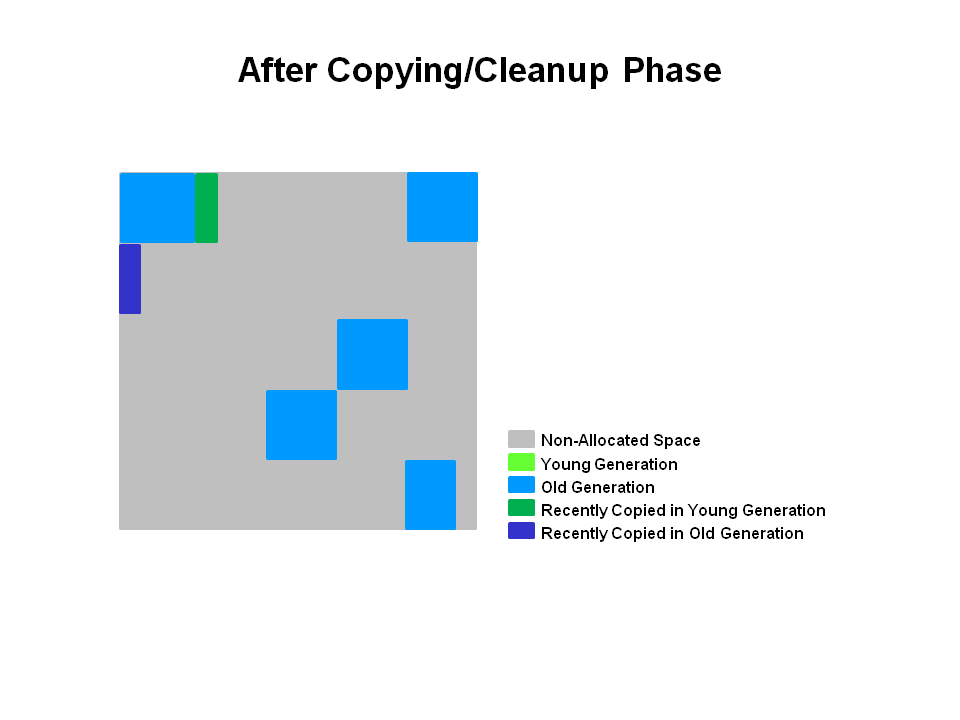
**Copying/Cleanup Phase**

G1 selects the regions with the lowest "liveness", those regions which can be collected the fastest. Then those regions are collected at the same time as a young GC. This is denoted in the logs as [GC pause (mixed)]. So both young and old generations are collected at the same time.



**After Copying/Cleanup Phase**

The regions selected have been collected and compacted into the dark blue region and the dark green region shown in the diagram.



#### Summary of Old Generation GC

In summary, there are a few key points we can make about the G1 garbage collection on the old generation.

* + Concurrent Marking Phase
    - Liveness information is calculated concurrently while the application is running.
    - This liveness information identifies which regions will be best to reclaim during an evacuation pause.
    - There is no sweeping phase like in CMS.
  + Remark Phase
    - Uses the Snapshot-at-the-Beginning (SATB) algorithm which is much faster then what was used with CMS.
    - Completely empty regions are reclaimed.
  + Copying/Cleanup Phase
    - Young generation and old generation are reclaimed at the same time.
    - Old generation regions are selected based on their liveness.

######################################################################

### Our Collectors

#### By jonthecollector on [Feb 01, 2008](https://blogs.oracle.com/jonthecollector/entry/our_collectors)

I drew this diagram on a white board for some customers recently. They seemed to like it (or were just being very polite) so I thought I redraw it for your amusement.

Each blue box represents a collector that is used to collect a generation. The young generation is collected by the blue boxes in the yellow region and the tenured generation is collected by the blue boxes in the gray region.

 "Serial" is a stop-the-world, copying collector which uses a single GC thread.

 "ParNew" is a stop-the-world, copying collector which uses multiple GC threads. It differs from "Parallel Scavenge" in that it has enhancements that make it usable with CMS. For example, "ParNew" does the synchronization needed so that it can run during the concurrent phases of CMS.

 "Parallel Scavenge" is a stop-the-world, copying collector which uses multiple GC threads.

 "Serial Old" is a stop-the-world, mark-sweep-compact collector that uses a single GC thread.

 "CMS" is a mostly concurrent, low-pause collector.

 "Parallel Old" is a compacting collector that uses multiple GC threads.

Using the -XX flags for our collectors for jdk6,

 UseSerialGC is "Serial" + "Serial Old"

 UseParNewGC is "ParNew" + "Serial Old"

 UseConcMarkSweepGC is "ParNew" + "CMS" + "Serial Old". "CMS" is used most of the time to collect the tenured generation. "Serial Old" is used when a concurrent mode failure occurs.

 UseParallelGC is "Parallel Scavenge" + "Serial Old"

 UseParallelOldGC is "Parallel Scavenge" + "Parallel Old"

FAQ

1) UseParNew and UseParallelGC both collect the young generation using multiple GC threads. Which is faster?

There's no one correct answer for this questions. Mostly they perform equally well, but I've seen one do better than the other in different situations. If you want to use GC ergonomics, it is only supported by UseParallelGC (and UseParallelOldGC) so that's what you'll have to use.

2) Why doesn't "ParNew" and "Parallel Old" work together?

"ParNew" is written in a style where each generation being collected offers certain interfaces for its collection. For example, "ParNew" (and "Serial") implements space\_iterate() which will apply an operation to every object in the young generation. When collecting the tenured generation with either "CMS" or "Serial Old", the GC can use space\_iterate() to do some work on the objects in the young generation. This makes the mix-and-match of collectors work but adds some burden to the maintenance of the collectors and to the addition of new collectors. And the burden seems to be quadratic in the number of collectors. Alternatively, "Parallel Scavenge" (at least with its initial implementation before "Parallel Old") always knew how the tenured generation was being collected and could call directly into the code in the "Serial Old" collector. "Parallel Old" is not written in the "ParNew" style so matching it with "ParNew" doesn't just happen without significant work. By the way, we would like to match "Parallel Scavenge" only with "Parallel Old" eventually and clean up any of the ad hoc code needed for "Parallel Scavenge" to work with both.

Please don't think too much about the examples I used above. They are admittedly contrived and not worth your time.

3) How do I use "CMS" with "Serial"?

-XX:+UseConcMarkSweepGC -XX:-UseParNewGC. Don't use -XX:+UseConcMarkSweepGC and -XX:+UseSerialGC. Although that's seems like a logical combination, it will result in a message saying something about conflicting collector combinations and the JVM won't start. Sorry about that. Our bad.

4) Is the blue box with the "?" a typo?

That box represents the new garbage collector that we're currently developing called Garbage First or G1 for short. G1 will provide

 More predictable GC pauses

 Better GC ergonomics

 Low pauses without fragmentation

 Parallelism and concurrency in collections

 Better heap utilization

G1 straddles the young generation - tenured generation boundary because it is a generational collector only in the logical sense. G1 divides the heap into regions and during a GC can collect a subset of the regions. It is logically generational because it dynamically selects a set of regions to act as a young generation which will then be collected at the next GC (as the young generation would be).

The user can specify a goal for the pauses and G1 will do an estimate (based on past collections) of how many regions can be collected in that time (the pause goal). That set of regions is called a collection set and G1 will collect it during the next GC.

G1 can choose the regions with the most garbage to collect first (Garbage First, get it?) so gets the biggest bang for the collection buck.

G1 compacts so fragmentation is much less a problem. Why is it a problem at all? There can be internal fragmentation due to partially filled regions.

The heap is not statically divided into a young generation and a tenured generation so the problem of an imbalance in their sizes is not there.

Along with a pause time goal the user can specify a goal on the fraction of time that can be spent on GC during some period (e.g., during the next 100 seconds don't spend more than 10 seconds collecting). For such goals (10 seconds of GC in a 100 second period) G1 can choose a collection set that it expects it can collect in 10 seconds and schedules the collection 90 seconds (or more) from the previous collection. You can see how an evil user could specify 0 collection time in the next century so again, this is just a goal, not a promise.

If G1 works out as we expect, it will become our low-pause collector in place of "ParNew" + "CMS". And if you're about to ask when will it be ready, please don't be offended by my dead silence. It's the highest priority project for our team, but it is software development so there are the usual unknowns. It will be out by JDK7. The sooner the better as far as we're concerned.

**Updated February 4.**Yes, I can edit an already posted blog. Here's a reference to the G1 paper if you have ACM portal access.

<http://portal.acm.org/citation.cfm?id=1029879>