[**Understanding Java Garbage Collection**](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)

What are the benefits of knowing how garbage collection (GC) works in [Java](http://www.cubrid.org/blog/tags/Java/)? Satisfying the intellectual curiosity as a software engineer would be a valid cause, but also, understanding how GC works can help you write much better Java applications.

This is a very personal and subjective opinion of mine, but I believe that a person well versed in GC tends to be a better Java developer. If you are interested in the GC process, that means you have experience in developing applications of certain size. If you have thought carefully about choosing the right GC algorithm, that means you completely understand the features of the application you have developed. Of course, this may not be common standards for a good developer. However, few would object when I say that understanding GC is a requirement for being a great Java developer.

This is the first of a series of "[*Become a Java GC Expert*](http://www.cubrid.org/blog/tags/Garbage%20Collection/)" articles. I will cover the *GC introduction* this time, and in the next article, I will talk about analyzing GC status and GC tuning examples from [NHN](http://www.cubrid.org/blog/tags/NHN/).

The purpose of this article is to introduce GC to you in an easy way. I hope this article proves to be very helpful. Actually, my colleagues have already published [a few great articles on Java Internals](http://www.cubrid.org/blog/tags/Java/) which became quite popular on Twitter. You may refer to them as well.

Returning back to Garbage Collection, there is a term that you should know before learning about GC. The term is "**stop-the-world**." Stop-the-world will occur no matter which GC algorithm you choose. *Stop-the-world*means that the [JVM](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/) is stopping the application from running to execute a GC. When stop-the-world occurs, every thread except for the threads needed for the GC will stop their tasks. The interrupted tasks will resume only after the GC task has completed. GC tuning often means reducing this stop-the-world time.

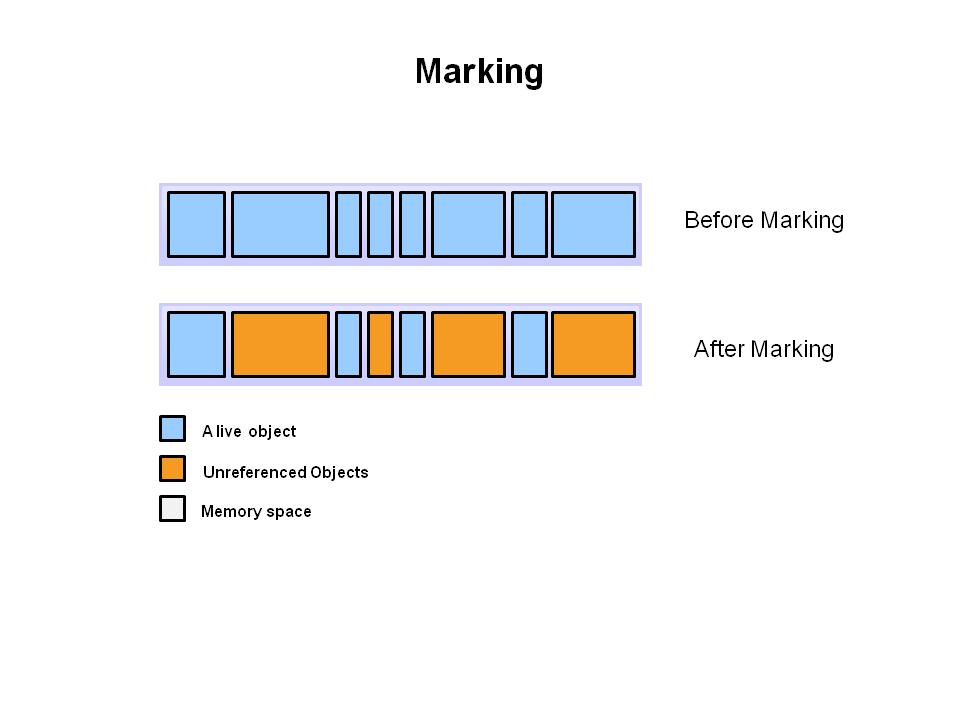
### What is Automatic Garbage Collection?

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

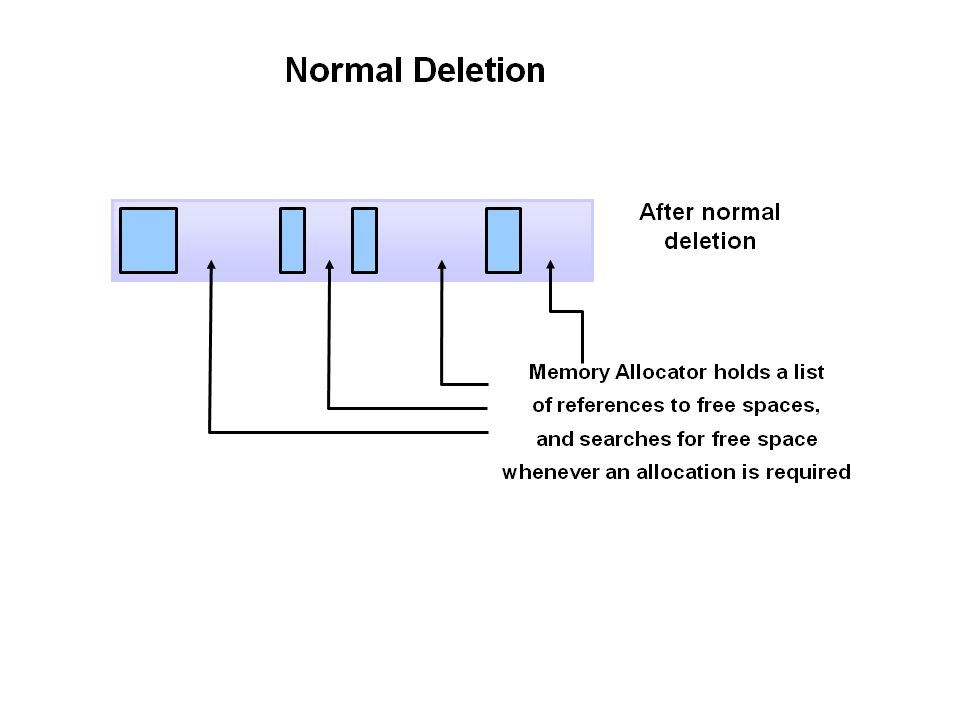
The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.



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

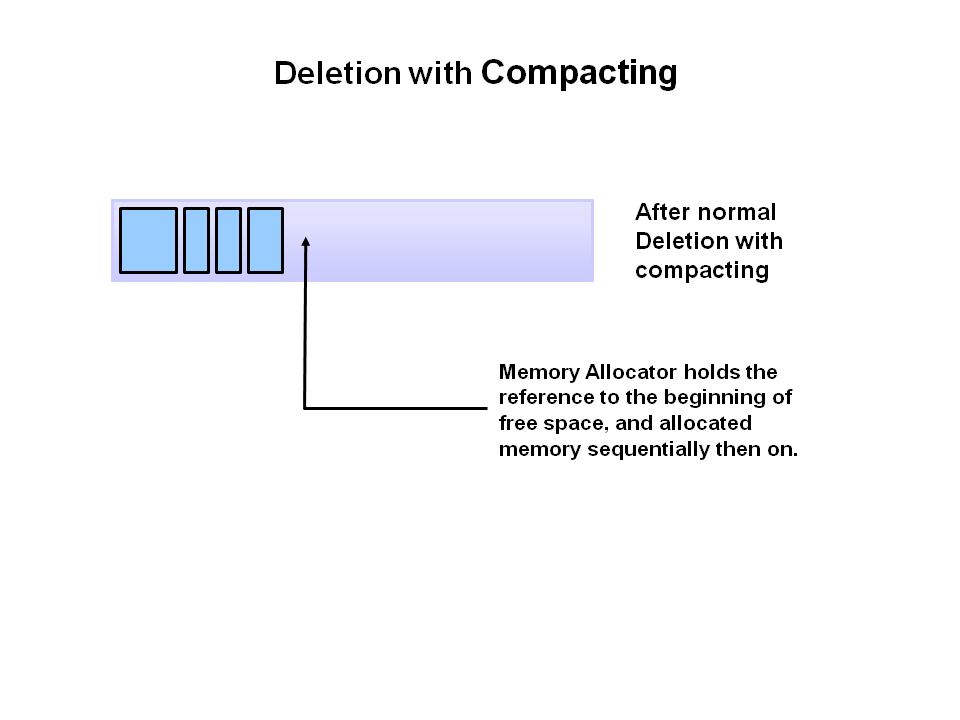
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.



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

#### Step 2a: Deletion with Compacting

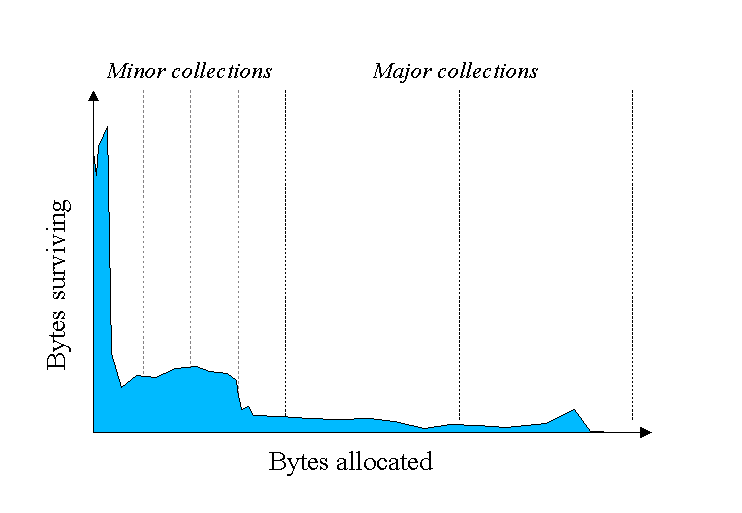
To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.



#### 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.

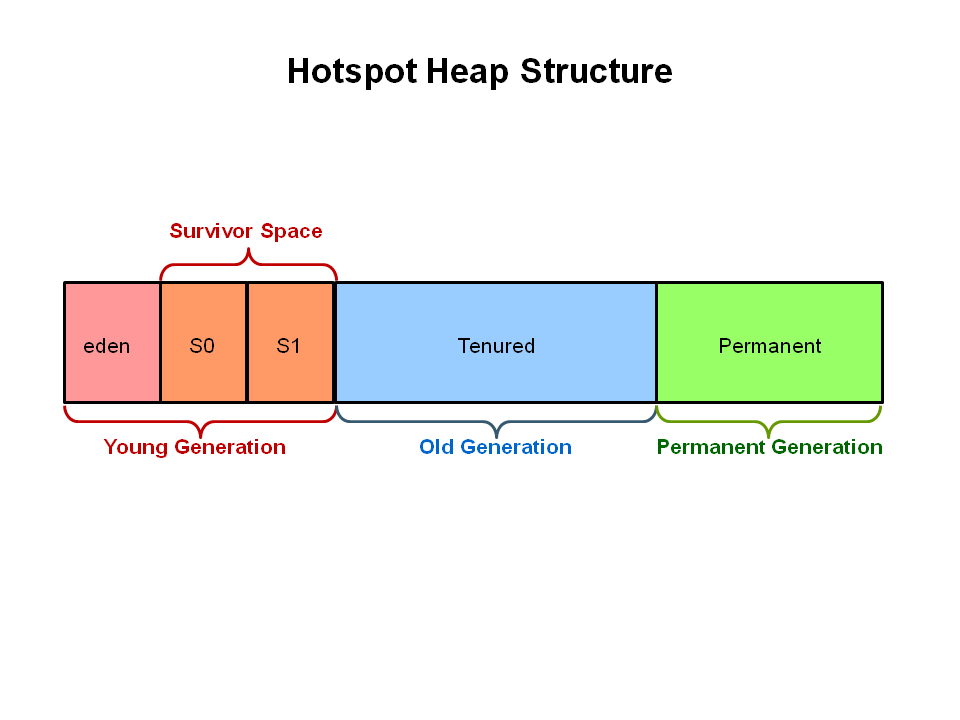
Here is an example of such data. The Y axis shows the number of bytes allocated and the X access shows the number of bytes allocated over time.



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.

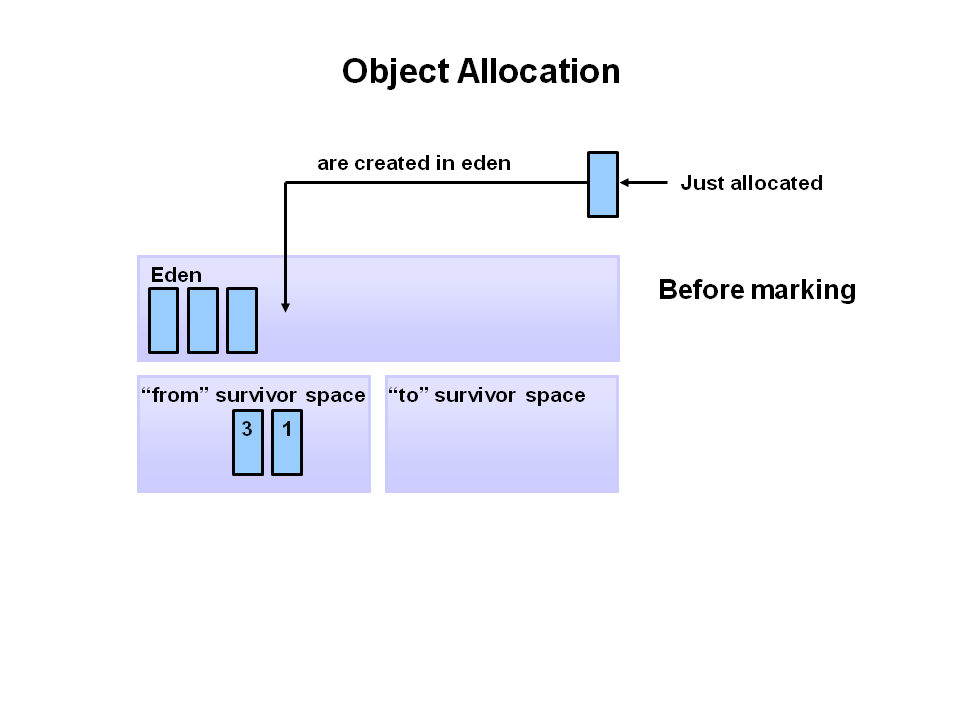
The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

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.

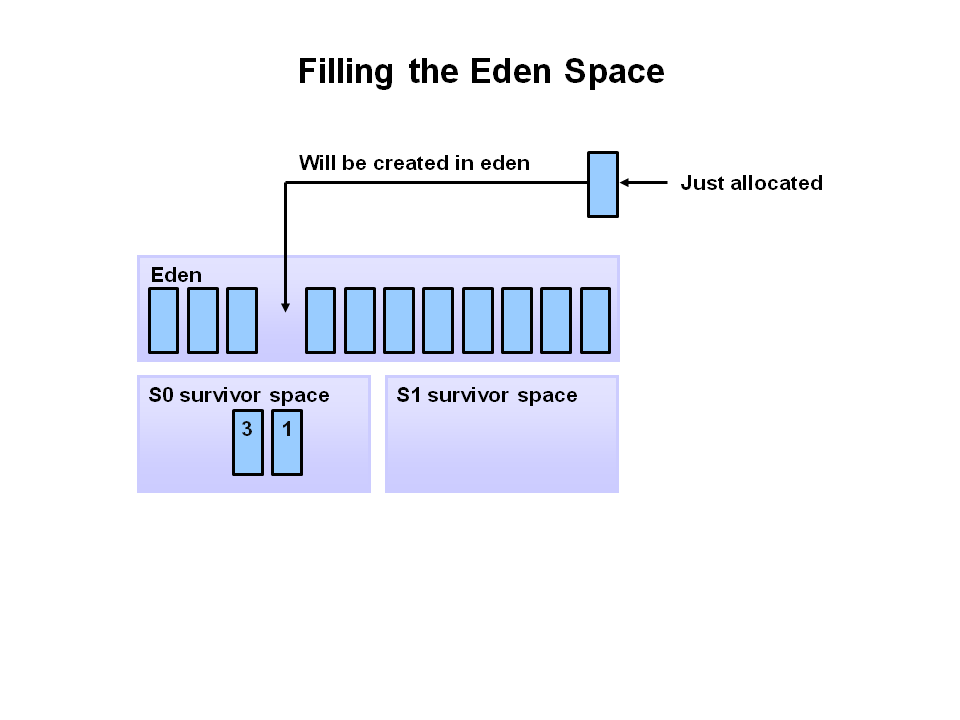
**The Generational Garbage Collection Process**

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.

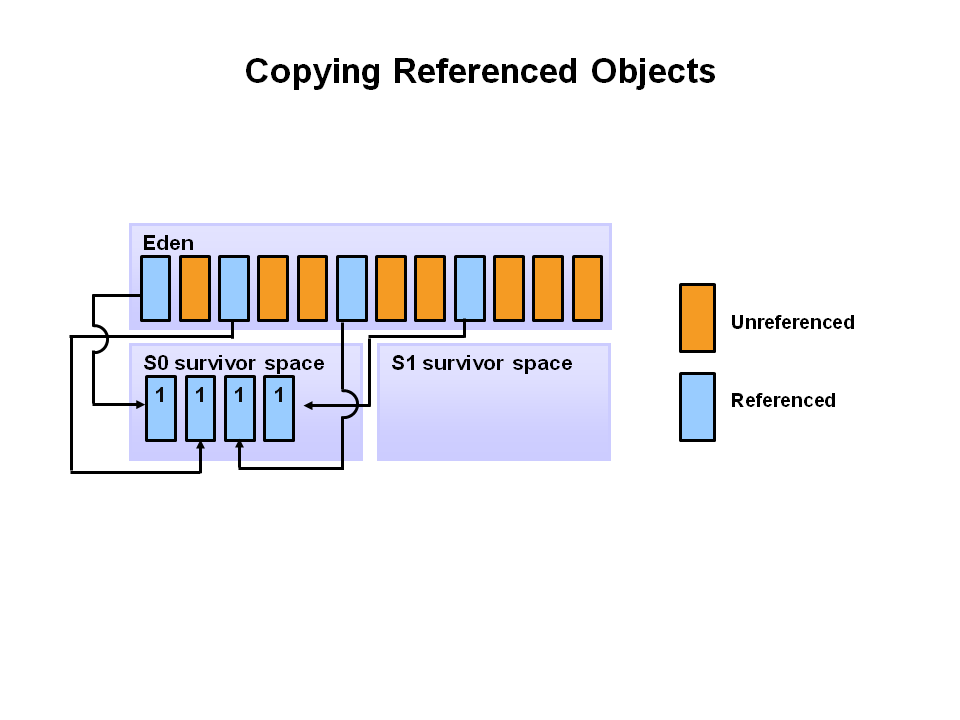
First, any new objects are allocated to the eden space. Both survivor spaces start out empty.



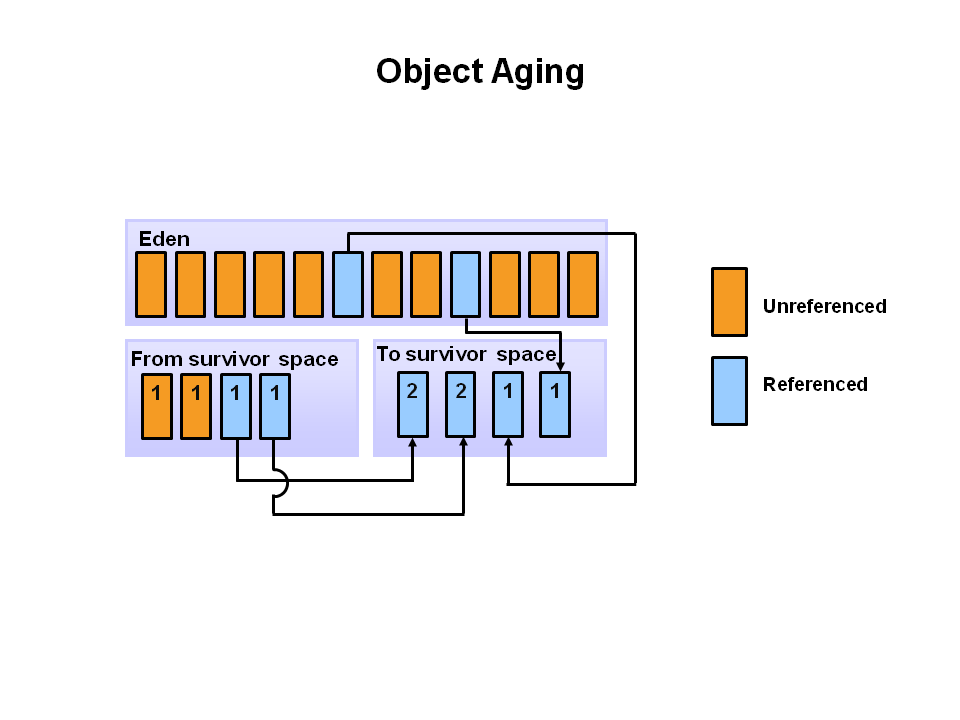
When the eden space fills up, a minor garbage collection is triggered.



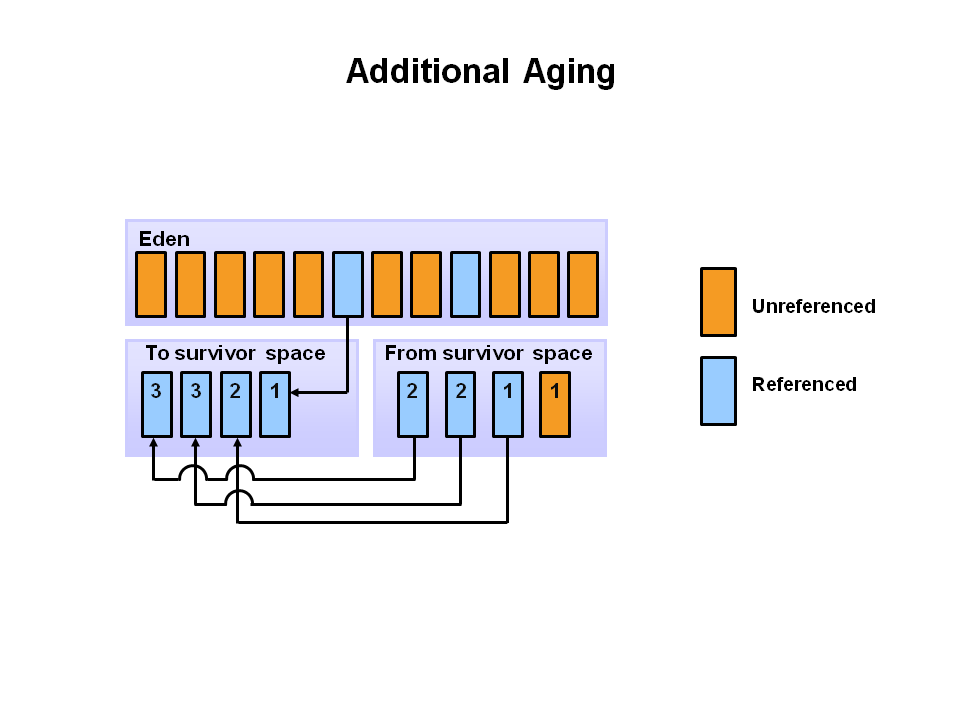
Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.



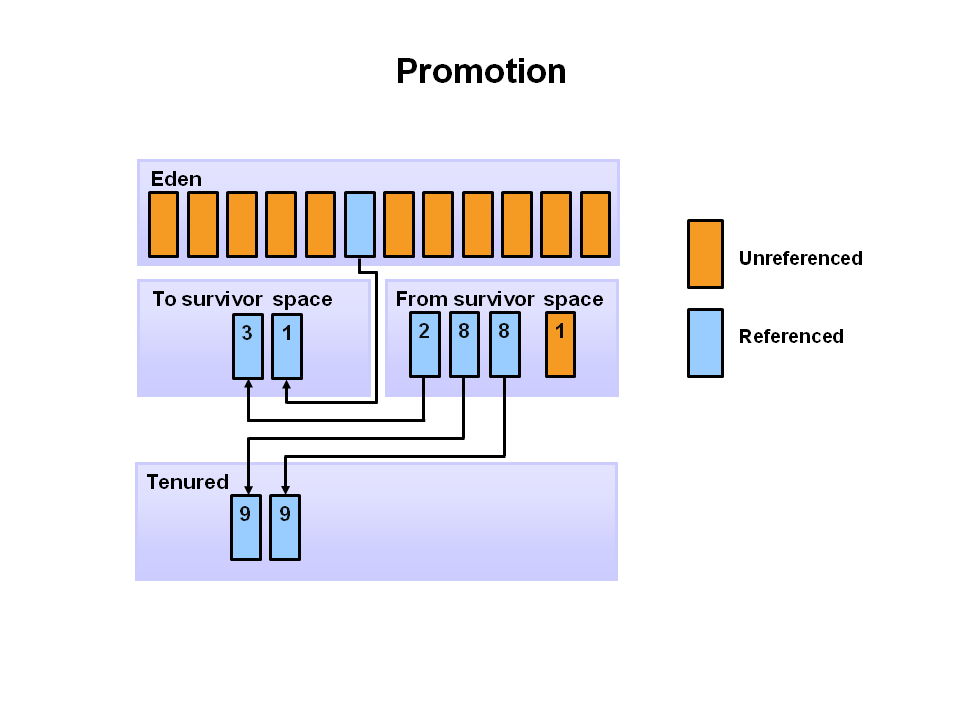
At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) 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.



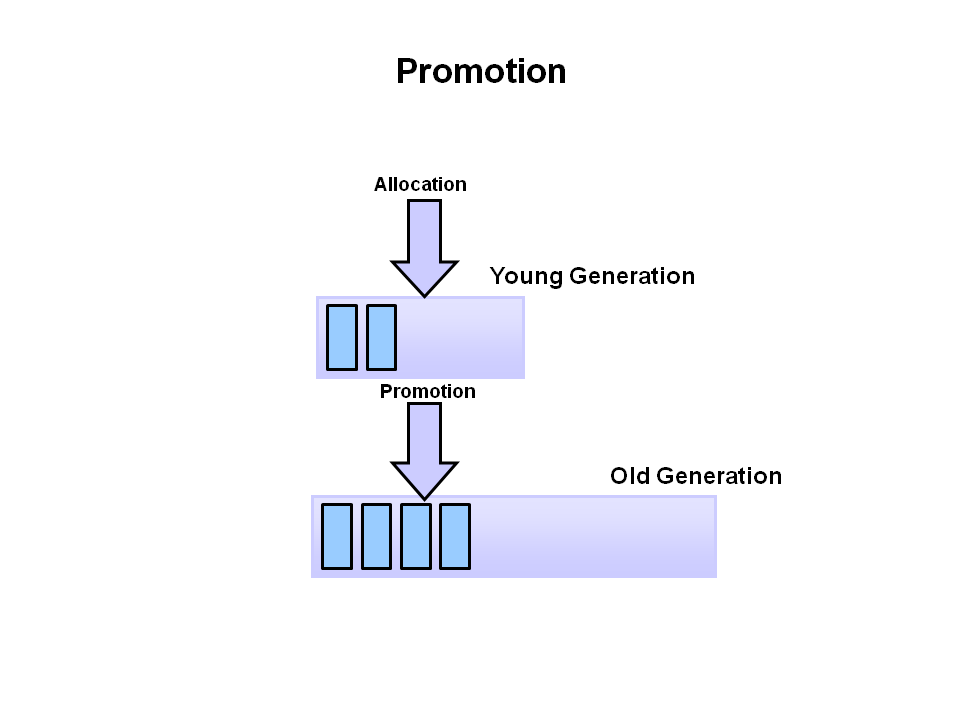
At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.



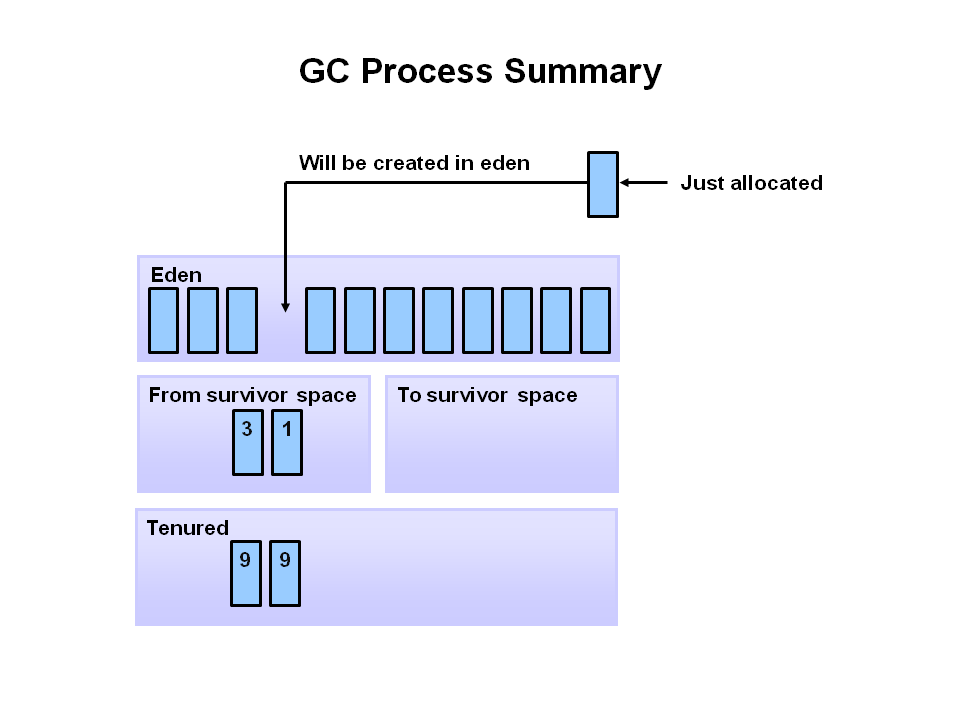
This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.



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



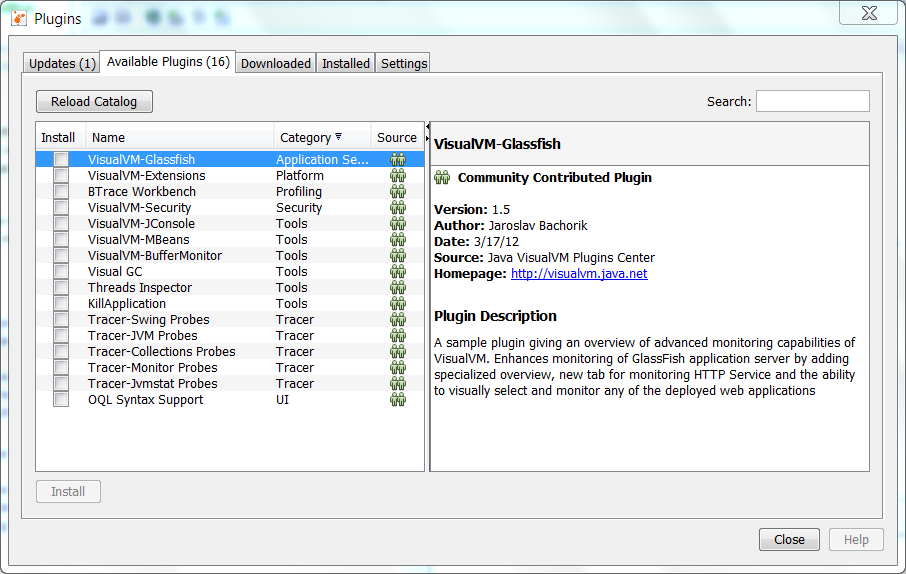
So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.



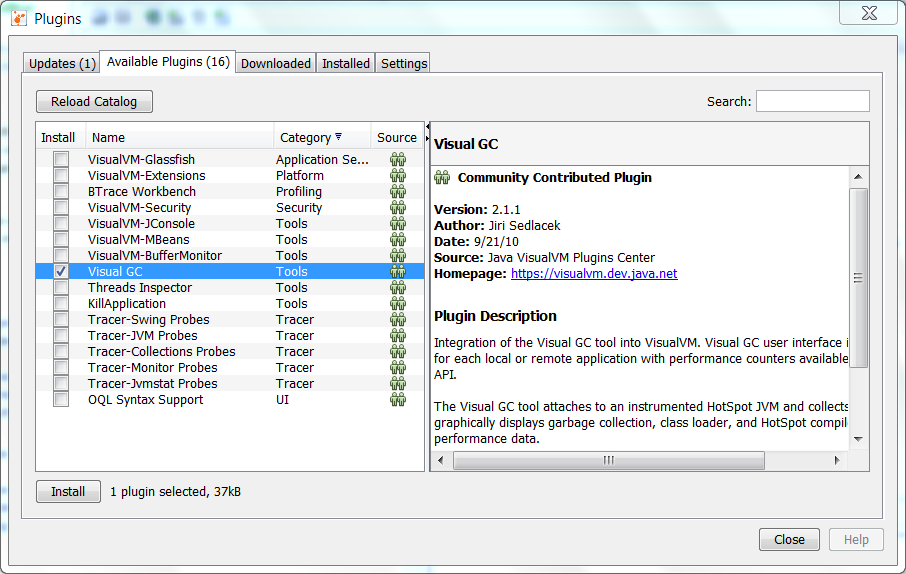
#### Install Visual GC

The Visual GC plugin for Visual VM provides a graphical representation of the garbage collectors activity in the JVM. It is free and can be added to your installation by following these steps.

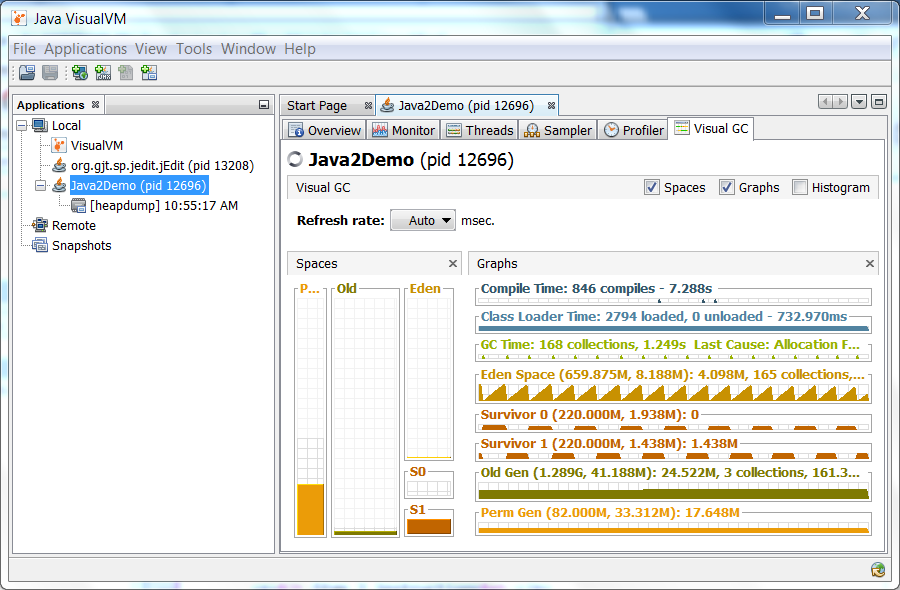
* 1. To install Visual GC, from the menu select **Tools --> Plugins**. You should get the following dialog.



Select **Visual GC** and then select the **Install** button.



The Visual GC tab shows all the activity going on in the Garbage Collector. You can watch as the Eden space is filled up and data is moved between the two survivor spaces S1 and S0. In addition, graphs and stats of related to the garbage collector are shown at the right.



**GC Types**

The old generation basically performs a GC when the data is full. The execution procedure varies by the GC type, so it would be easier to understand if you know different types of GC.

According to JDK 7, there are 5 GC types.

1. Serial GC
2. Parallel GC
3. Parallel Old GC (Parallel Compacting GC)
4. Concurrent Mark & Sweep GC  (or "CMS")
5. Garbage First (G1) GC

Among these, the **serial GC must not be used on an operating server**. This GC type was created when there was only one CPU core on desktop computers. Using this serial GC will drop the application performance significantly.

Now let's learn about each GC type.

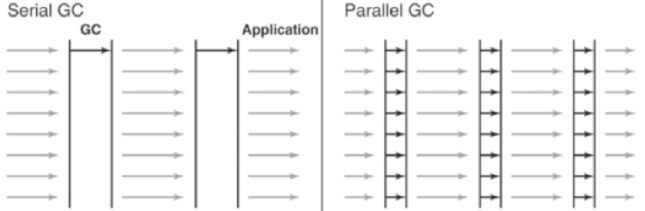
**Serial GC (-XX:+UseSerialGC)**

The GC in the young generation uses the type we explained in the previous paragraph. The GC in the old generation uses an algorithm called "**mark-sweep-compact**."

1. The first step of this algorithm is to mark the surviving objects in the old generation.
2. Then, it checks the heap from the front and leaves only the surviving ones behind (sweep).
3. In the last step, it fills up the heap from the front with the objects so that the objects are piled up consecutively, and divides the heap into two parts: one with objects and one without objects (compact).

The serial GC is suitable for a small memory and a small number of CPU cores.

**Parallel GC (-XX:+UseParallelGC)**



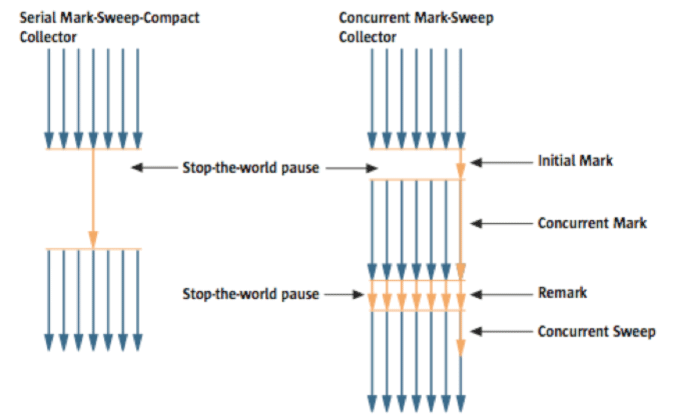
**Figure 4: Difference between the Serial GC and Parallel GC.**

From the picture, you can easily see the difference between the serial GC and parallel GC. While the serial GC uses only one thread to process a GC, the parallel GC uses several threads to process a GC, and therefore, faster. This GC is useful when there is enough memory and a large number of cores. It is also called the "**throughput GC**."

**Parallel Old GC(-XX:+UseParallelOldGC)**

Parallel Old GC was supported since JDK 5 update. Compared to the parallel GC, the only difference is the GC algorithm for the old generation. It goes through three steps: *mark – summary – compaction*. The summary step identifies the surviving objects separately for the areas that the GC have previously performed, and thus different from the sweep step of the mark-sweep-compact algorithm. It goes through a little more complicated steps.

**CMS GC (-XX:+UseConcMarkSweepGC)**



**Figure 5: Serial GC & CMS GC.**

As you can see from the picture, the Concurrent Mark-Sweep GC is much more complicated than any other GC types that I have explained so far. The early *initial mark* step is simple. The surviving objects among the objects the closest to the classloader are searched. So, the pausing time is very short. In the *concurrent mark* step, the objects referenced by the surviving objects that have just been confirmed are tracked and checked. The difference of this step is that it proceeds while other threads are processed at the same time. In the *remark*step, the objects that were newly added or stopped being referenced in the concurrent mark step are checked. Lastly, in the *concurrent sweep* step, the garbage collection procedure takes place. The garbage collection is carried out while other threads are still being processed. Since this GC type is performed in this manner, the pausing time for GC is very short. The CMS GC is also called the low latency GC, and is **used when the response time from all applications is crucial**.

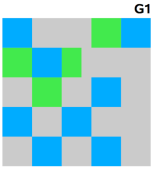
While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

* It uses more memory and CPU than other GC types.
* The compaction step is not provided by default.

You need to carefully review before using this type. Also, if the compaction task needs to be carried out because of the many memory fragments, the stop-the-world time can be longer than any other GC types. You need to check how often and how long the compaction task is carried out.

**G1 GC**

Finally, let's learn about the garbage first (G1) GC.



**Figure 6: Layout of G1 GC.**

If you want to understand G1 GC, forget everything you know about the young generation and the old generation. As you can see in the picture, one object is allocated to each grid, and then a GC is executed. Then, once one area is full, the objects are allocated to another area, and then a GC is executed. The steps where the data moves from the three spaces of the young generation to the old generation cannot be found in this GC type. This type was created to replace the CMS GC, which has causes a lot of issues and complaints in the long term.

The biggest advantage of the G1 GC is its **performance**. It is faster than any other GC types that we have discussed so far. But in JDK 6, this is called an *early access* and can be used only for a test. It is officially included in JDK 7. In my personal opinion, we need to go through a long test period (at least 1 year) before NHN can use JDK7 in actual services, so you probably should wait a while. Also, I heard a few times that a JVM crash occurred after applying the G1 in JDK 6. Please wait until it is more stable.

I will talk about the **GC tuning** in the next issue, but I would like to ask you one thing in advance. If the size and the type of all objects created in the application are identical, all the GC options for WAS used in our company can be the same. But the size and the lifespan of the objects created by WAS vary depending on the service, and the type of equipment varies as well. In other words, just because a certain service uses the GC option "A," it does not mean that the same option will bring the best results for a different service. It is necessary to find the best values for the WAS threads, WAS instances for each equipment and each GC option by constant tuning and monitoring. This did not come from my personal experience, but from the discussion of the engineers making Oracle JVM for JavaOne 2010.

In this issue, we have only glanced at the GC for Java. Please look forward to our next issue, where I will talk about **how to monitor the Java GC status and tune GC**.

### Common Heap Related Switches

There are many different command line switches that can be used with Java. This section describes some of the more commonly used switches that are also used in this OBE.

| **Switch** | **Description** |
| --- | --- |
| -Xms | Sets the initial heap size for when the JVM starts. |
| -Xmx | Sets the maximum heap size. |
| -Xmn | Sets the size of the Young Generation. |
| -XX:PermSize | Sets the starting size of the Permanent Generation. |
| -XX:MaxPermSize | Sets the maximum size of the Permanent Generation |

### The Serial GC

The serial collector is the default for client style machines in Java SE 5 and 6. With the serial collector, both minor and major garbage collections are done serially (using a single virtual CPU). In addition, it uses a mark-compact collection method. This method moves older memory to the beginning of the heap so that new memory allocations are made into a single continuous chunk of memory at the end of the heap. This compacting of memory makes it faster to allocate new chunks of memory to the heap.

#### Usage Cases

The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work (therefore, its name). Still, on today's hardware, the Serial GC can efficiently manage a lot of non-trivial applications with a few hundred MBs of Java heap, with relatively short worst-case pauses (around a couple of seconds for full garbage collections).

Another popular use for the Serial GC is in environments where a high number of JVMs are run on the same machine (in some cases, more JVMs than available processors!). In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs, even if the garbage collection might last longer. And the Serial GC fits this trade-off nicely.

Finally, with the proliferation of embedded hardware with minimal memory and few cores, the Serial GC could make a comeback.

#### Command Line Switches

To enable the Serial Collector use:  
-XX:+UseSerialGC

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

### The Parallel GC

The parallel garbage collector uses multiple threads to perform the young genertion garbage collection. By default on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection. The number of garbage collector threads can be controlled with command-line options:  
-XX:ParallelGCThreads=<desired number>

On a host with a single CPU the default garbage collector is used even if the parallel garbage collector has been requested. On a host with two CPUs the parallel garbage collector generally performs as well as the default garbage collector and a reduction in the young generationgarbage collector pause times can be expected on hosts with more than two CPUs. The Parallel GC comes in two flavors.

#### Usage Cases

The Parallel collector is also called a throughput collector. Since it can use multilple CPUs to speed up application throughput. This collector should be used when a lot of work need to be done and long pauses are acceptable. For example, batch processing like printing reports or bills or performing a large number of database queries.

#### -XX:+UseParallelGC

With this command line option you get a multi-thread young generation collector with a single-threaded old generation collector. The option also does single-threaded compaction of old generation.

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

#### -XX:+UseParallelOldGC

With the -XX:+UseParallelOldGC option, the GC is both a multithreaded young generation collector and multithreaded old generation collector. It is also a multithreaded compacting collector. HotSpot does compaction only in the old generation. Young generation in HotSpot is considered a copy collector; therefore, there is no need for compaction.

Compacting describes the act of moving objects in a way that there are no holes between objects. After a garbage collection sweep, there may be holes left between live objects. Compacting moves objects so that there are no remaining holes. It is possible that a garbage collector be a non-compacting collector. Therefore, the difference between a parallel collector and a parallel compacting collector could be the latter compacts the space after a garbage collection sweep. The former would not.

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelOldGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

### The Concurrent Mark Sweep (CMS) Collector

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.

#### Usage Cases

The CMS collector should be used for applications that require low pause times and can share resources with the garbage collector. Examples include desktop UI application that respond to events, a webserver responding to a request or a database responding to queries.

#### Command Line Switches

To enable the CMS Collector use:  
-XX:+UseConcMarkSweepGC  
and to set the number of threads use:  
-XX:ParallelCMSThreads=<n>

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2 -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

### The G1 Garbage Collector

The Garbage First or G1 garbage collector is available in Java 7 and is designed to be the long term replacement for the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector that has quite a different layout from the other garbage collectors described previously. However, detailed discussion is beyond the scope of this OBE.

#### Command Line Switches

To enable the G1 Collector use:  
-XX:+UseG1GC

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -XX:+UseG1GC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

# How to Monitor Java Garbage Collection

**What is GC Monitoring?**

**Garbage Collection Monitoring** refers to the *process of figuring out how JVM is running GC*. For example, we can find out:

1. when an object in young has moved to old and by how much,
2. or when stop-the-world has occurred and for how long.

GC monitoring is carried out *to see if JVM is running GC efficiently*, and *to check if additional GC tuning is necessary*. Based on this information, the application can be edited or GC method can be changed (**GC tuning**).

**How to Monitor GC?**

There are different ways to monitor GC, but the only difference is how the GC operation information is shown. GC is done by JVM, and since the GC monitoring tools disclose the GC information provided by JVM, you will get the same results no matter how you monitor GC. Therefore, you do not need to learn all methods to monitor GC, but since it only requires a little amount of time to learn each GC monitoring method, knowing a few of them can help you use the right one for different situations and environments.

The tools or JVM options listed below cannot be used universally regardless of the HVM vendor. This is because there is no need for a "standard" for disclosing GC information. In this example we will use **HotSpot JVM**(Oracle JVM). Since NHN is using Oracle (Sun) JVM, there should be no difficulties in applying the tools or JVM options that we are explaining here.

First, the GC monitoring methods can be separated into **CUI** and **GUI** depending on the access interface. The typical CUI GC monitoring method involves using a separate CUI application called "**jstat**", or selecting a JVM option called "**verbosegc**" when running JVM.

GUI GC monitoring is done by using a separate GUI application, and three most commonly used applications would be "jconsole", "jvisualvm" and "Visual GC".

Let's learn more about each method.

**jstat**

**jstat** is a monitoring tool in HotSpot JVM. Other monitoring tools for HotSpot JVM are **jps** and **jstatd**. Sometimes, you need all three tools to monitor a Java application.

**jstat** does not provide only the GC operation information display. It also provides class loader operation information or Just-in-Time compiler operation information. Among all the information jstat can provide, in this article we will only cover its functionality to *monitor* GC operating information.

**jstat** is located in $JDK\_HOME/bin, so if *java* or *javac* can run without setting a separate directory from the command line, so can jstat.

You can try running the following in the command line.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $> jstat –gc  $<vmid$> 1000    S0C       S1C       S0U    S1U      EC         EU          OC         OU         PC         PU         YGC     YGCT    FGC      FGCT     GCT  3008.0   3072.0    0.0     1511.1   343360.0   46383.0     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588  3008.0   3072.0    0.0     1511.1   343360.0   47530.9     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588  3008.0   3072.0    0.0     1511.1   343360.0   47793.0     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588    $> |

Just like in the example, the real type data will be output along with the following columns: **S0C    S1C     S0U     S1U    EC     EU     OC     OU     PC**.

**vmid** (Virtual Machine ID), as its name implies, is the **ID** for the VM. Java applications running either on a local machine or on a remote machine can be specified using vmid. The vmid for Java application running on a local machine is called **lvmid** (Local vmid), and usually is PID. To find out the lvmid, you can write the PID value using a **ps** command or Windows task manager, but we suggest **jps** because PID and lvmid does not always match. **jps**stands for Java PS. jps shows *vmids* and main method information. Just like ps shows PIDs and process names.

Find out the vmid of the Java application that you want to monitor by using jps, then use it as a parameter in jstat. If you use jps alone, only bootstrap information will show when several WAS instances are running in one equipment. We suggest that you use **ps -ef | grep java** command along with **jps**.

GC performance data needs constant observation, therefore when running jstat, try to output the GC monitoring information on a regular basis.

For example, running "**jstat –gc <vmid> 1000**" (or 1s) will display the GC monitoring data on the console every 1 second. "**jstat –gc <vmid> 1000 10**" will display the GC monitoring information once every 1 second for 10 times in total.

There are many options other than **-gc**, among which GC related ones are listed below.

| Option Name | Description |
| --- | --- |
| gc | It shows the current size for each heap area and its current usage (Ede, survivor, old, etc.), total number of GC performed, and the accumulated time for GC operations. |
| gccapactiy | It shows the minimum size (ms) and maximum size (mx) of each heap area, current size, and the number of GC performed for each area. (Does not show current usage and accumulated time for GC operations.) |
| gccause | It shows the "information provided by -gcutil" + reason for the last GC and the reason for the current GC. |
| gcnew | Shows the GC performance data for the new area. |
| gcnewcapacity | Shows statistics for the size of new area. |
| gcold | Shows the GC performance data for the old area. |
| gcoldcapacity | Shows statistics for the size of old area. |
| gcpermcapacity | Shows statistics for the permanent area. |
| gcutil | Shows the usage for each heap area in percentage. Also shows the total number of GC performed and the accumulated time for GC operations. |

Only looking at frequency, you will probably use **-gcutil** (or -gccause), **-gc** and **-gccapacity** the most in that order.

* **-gcutil** is used to check the usage of heap areas, the number of GC performed, and the total accumulated time for GC operations,
* while **-gccapacity** option and others can be used to check the actual size allocated.

You can see the following output by using the **-gc** option:

|  |  |
| --- | --- |
| 1  2  3  4 | S0C      S1C    …   GCT  1248.0   896.0  …   1.246  1248.0   896.0  …   1.246  …        …      …   … |

| Column | Description | Jstat Option |
| --- | --- | --- |
| S0C | Displays the current size of Survivor0 area in KB | -gc -gccapacity -gcnew -gcnewcapacity |
| S1C | Displays the current size of Survivor1 area in KB | -gc -gccapacity -gcnew -gcnewcapacity |
| S0U | Displays the current usage of Survivor0 area in KB | -gc -gcnew |
| S1U | Displays the current usage of Survivor1 area in KB | -gc -gcnew |
| EC | Displays the current size of Eden area in KB | -gc -gccapacity -gcnew -gcnewcapacity |
| EU | Displays the current usage of Eden area in KB | -gc -gcnew |
| OC | Displays the current size of old area in KB | -gc -gccapacity -gcold -gcoldcapacity |
| OU | Displays the current usage of old area in KB | -gc -gcold |
| PC | Displays the current size of permanent area in KB | -gc -gccapacity -gcold -gcoldcapacity -gcpermcapacity |
| PU | Displays the current usage of permanent area in KB | -gc -gcold |
| YGC | The number of GC event occurred in young area | -gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| YGCT | The accumulated time for GC operations for Yong area | -gc -gcnew -gcutil -gccause |
| FGC | The number of full GC event occurred | -gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| FGCT | The accumulated time for full GC operations | -gc -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| GCT | The total accumulated time for GC operations | -gc -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause |
| NGCMN | The minimum size of new area in KB | -gccapacity -gcnewcapacity |
| NGCMX | The maximum size of max area in KB | -gccapacity -gcnewcapacity |
| NGC | The current size of new area in KB | -gccapacity -gcnewcapacity |
| OGCMN | The minimum size of old area in KB | -gccapacity -gcoldcapacity |
| OGCMX | The maximum size of old area in KB | -gccapacity -gcoldcapacity |
| OGC | The current size of old area in KB | -gccapacity -gcoldcapacity |
| PGCMN | The minimum size of permanent area in KB | -gccapacity -gcpermcapacity |
| PGCMX | The maximum size of permanent area in KB | -gccapacity -gcpermcapacity |
| PGC | The current size of permanent generation area in KB | -gccapacity -gcpermcapacity |
| PC | The current size of permanent area in KB | -gccapacity -gcpermcapacity |
| PU | The current usage of permanent area in KB | -gc -gcold |
| LGCC | The cause for the last GC occurrence | -gccause |
| GCC | The cause for the current GC occurrence | -gccause |
| TT | Tenuring threshold. If copied this amount of times in young area (S0 ->S1, S1->S0), they are then moved to old area. | -gcnew |
| MTT | Maximum Tenuring threshold. If copied this amount of times inside young arae, then they are moved to old area. | -gcnew |
| DSS | Adequate size of survivor in KB | -gcnew |

Different jstat options show different types of columns, which are listed below. Each column information will be displayed when you use the "jstat option" listed on the right.

The advantage of **jstat** is that it can always monitor the GC operation data of Java applications running on local/remote machine, as long as a console can be used. From these items, the following result is output when**–gcutil** is used. At the time of GC tuning, pay careful attention to **YGC, YGCT, FGC, FGCT** and **GCT**.

|  |  |
| --- | --- |
| 1  2  3  4 | S0      S1       E        O        P        YGC    YGCT     FGC    FGCT     GCT  0.00    66.44    54.12    10.58    86.63    217    0.928     2     0.067    0.995  0.00    66.44    54.12    10.58    86.63    217    0.928     2     0.067    0.995  0.00    66.44    54.12    10.58    86.63    217    0.928     2     0.067    0.995 |

These items are important because they show how much time was spent in running GC.

In this example, **YGC** is 217 and **YGCT** is 0.928. So, after calculating the arithmetical average, you can see that it required about *4 ms* (0.004 seconds) for each young GC. Likewise, the average full GC time us *33ms*.

But the arithmetical average often does not help analyzing the actual GC problem. This is due to the severe deviations in GC operation time. (In other words, if the average time is *0.067 seconds* for a full GC, one GC may have lasted 1 ms while the other one lasted *57 ms*.) In order to check the individual GC time instead of the arithmetical average time, it is better to use **-verbosegc**.

**-verbosegc**

**-verbosegc** is one of the JVM options specified when running a Java application. While *jstat* can monitor any JVM application that has not specified any options, **-verbosegc** needs to be specified in the beginning, so it could be seen as an unnecessary option (since jstat can be used instead). However, as **-verbosegc** displays easy to understand output results whenever a GC occurs, it is very helpful for monitoring rough GC information.

|  | jstat | -verbosegc |
| --- | --- | --- |
| Monitoring Target | Java application running on a machine that can log in to a terminal, or a remote Java application that can connect to the network by using jstatd | Only when -verbogc was specified as a JVM starting option |
| Output information | Heap status (usage, maximum size, number of times for GC/time, etc.) | Size of ew and old area before/after GC, and GC operation time |
| Output Time | Every designated time | Whenever GC occurs |
| Whenever useful | When trying to observe the changes of the size of heap area | When trying to see the effect of a single GC |

The followings are other options that can be used with **-verbosegc**.

* -XX:+PrintGCDetails
* -XX:+PrintGCTimeStamps
* -XX:+PrintHeapAtGC
* -XX:+PrintGCDateStamps (from JDK 6 update 4)

If only **-verbosegc** is used, then **-XX:+PrintGCDetails** is applied by default. Additional options for **–verbosgc** are not exclusive and can be mixed and used together.

When using **-verbosegc**, you can see the results in the following format whenever a minor GC occurs.

|  |  |
| --- | --- |
| [GC [<collector>: <starting occupancy1> -> <ending occupancy1>, <pause time1> secs] <starting occupancy3> -> <ending occupancy3>, <pause time3> secs] | |
| Collector | Name of Collector Used for minor gc |
| starting occupancy1 | The size of young area before GC |
| ending occupancy1 | The size of young area after GC |
| pause time1 | The time when the Java application stopped running for minor GC |
| starting occupancy3 | The total size of heap area before GC |
| ending occupancy3 | The total size of heap area after GC |
| pause time3 | The time when the Java application stopped running for overall heap GC, including major GC |

This is an example of **-verbosegc** output for **minor GC**:

|  |  |
| --- | --- |
| 1  2  3  4 | S0    S1     E      O      P        YGC    YGCT    FGC    FGCT     GCT  0.00  66.44  54.12  10.58  86.63    217    0.928     2    0.067    0.995  0.00  66.44  54.12  10.58  86.63    217    0.928     2    0.067    0.995  0.00  66.44  54.12  10.58  86.63    217    0.928     2    0.067    0.995 |

This is the example of output results after an **Full GC** occurred.

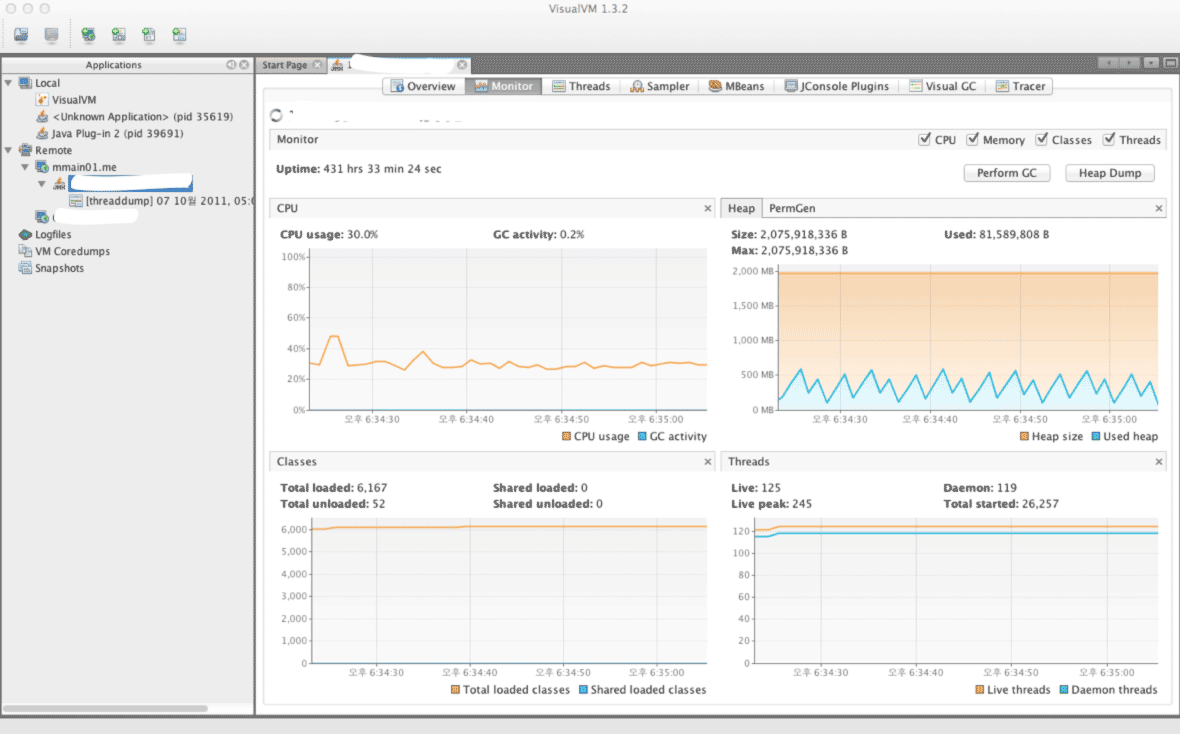
|  |  |
| --- | --- |
| 1 | [Full GC [Tenured: 3485K->4095K(4096K), 0.1745373 secs] 61244K->7418K(63104K), [Perm : 10756K->10756K(12288K)], 0.1762129 secs] [Times: user=0.19 sys=0.00, real=0.19 secs] |

If a [CMS collector](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/#cms-gc) is used, then the following CMS information can be provided as well.

As **-verbosegc** option outputs a log every time a GC event occurs, it is easy to see the changes of the heap usage rates caused by GC operation.

**(Java) VisualVM  + Visual GC**

Java Visual VM is a GUI profiling/monitoring tool provided by Oracle JDK.

[](http://www.cubrid.org/files/attach/images/220547/126/316/visual-vm.png)

**Figure 1: VisualVM Screenshot.**

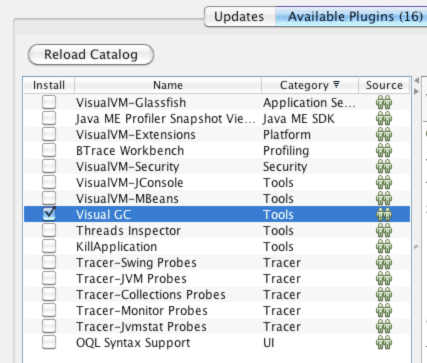
Instead of the version that is included with JDK, you can download Visual VM directly from its website. For the sake of convenience, the version included with JDK will be referred to as Java VisualVM (jvisualvm), and the version available from the website will be referred to as Visual VM (visualvm). The features of the two are not exactly identical, as there are slight differences, such as when installing plug-ins. Personally, I prefer the Visual VM version, which can be downloaded from the website.

After running Visual VM, if you select the application that you wish to monitor from the window on the left side, you can find the "*Monitoring*" tab there. You can get the basic information about GC and Heap from this Monitoring tab.

Though the basic GC status is also available through the basic features of VisualVM, you cannot access detailed information that is available from either **jstat** or **-verbosegc** option.

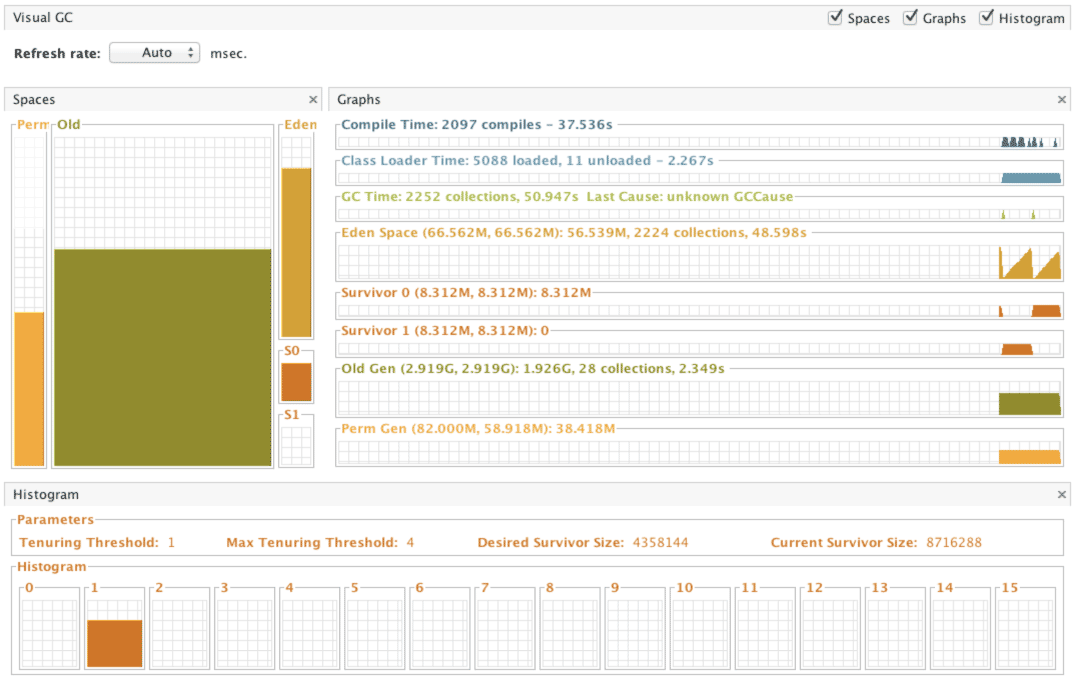
If you want the detailed information provided by jstat, then it is recommended to install the Visual GC plug-in.

Visual GC can be accessed in real time from the *Tools* menu.



**Figure 2: Viusal GC Installation Screenshot.**

By using Visual GC, you can see the information provided by running **jstatd** in a more intuitive way.

[](http://www.cubrid.org/files/attach/images/220547/126/316/visual-gc-execution.png)

**Figure 3: Visual GC execution screenshot.**

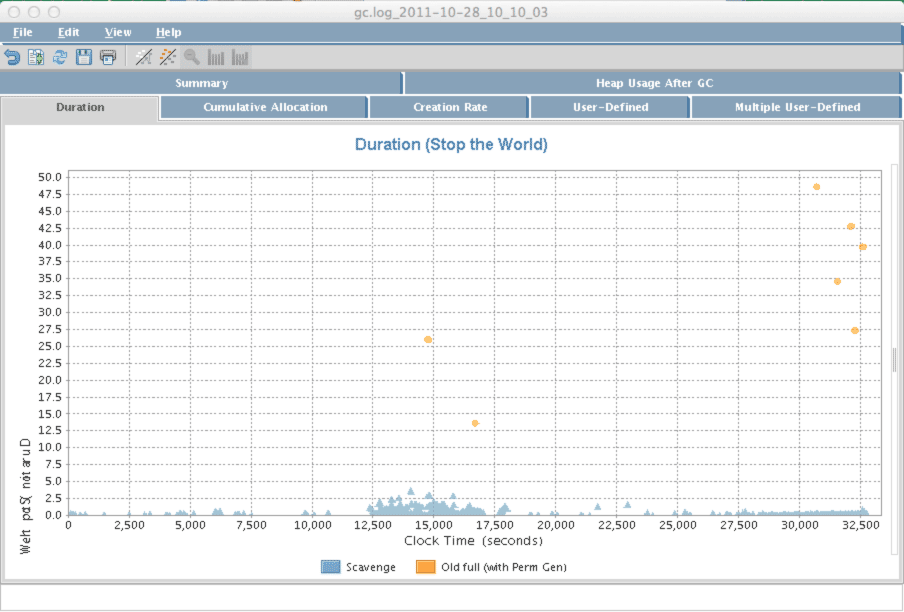
**HPJMeter**

[HPJMeter](https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPJMETER) is convenient for analyzing **-verbosegc** output results. If Visual GC can be considered as the GUI equivalent of *jstat*, then HPJMeter would be the GUI equivalent of *-verbosgc*. Of course, GC analysis is just one of the many features provided by HPJMeter. HPJMeter is a performance monitoring tool developed by HP. It can be used in HP-UX, as well as Linux and MS Windows.

Originally, a tool called **HPTune** used to provide the GUI analysis feature for **-verbosegc**. However, since the HPTune feature has been integrated into HPJMeter since version 3.0, there is no need to download HPTune separately.

When executing an application, the **-verbosegc** output results will be redirected to a separate file.

You can open the redirected file with HPJMeter, which allows faster and easier GC performance data analysis through the intuitive GUI.

[](http://www.cubrid.org/files/attach/images/220547/126/316/hpjmeter.png)

**Figure 4: HPJMeter.**