Memory Management in the Java HotSpot™ Virtual Machine

Sun Microsystems
April 2006

Table of Contents

1 Introduction **3**
2 Explicit vs. Automatic Memory Management **4**
3 Garbage Collection Concepts **4**
Desirable Garbage Collector Characteristics **4**
Design Choices **5**
Performance Metrics **5**
Generational Collection **6**
4 Garbage Collectors in the J2SE 5.0 HotSpot JVM **7**
Hotspot Generations **7**
Garbage Collection Types **7**
Fast Allocation **8**
Serial Collector **8**
Parallel Collector **9**
Parallel Compacting Collector **10**
Concurrent Mark-Sweep (CMS) Collector **11**
5 Ergonomics – Automatic Selections and Behavior Tuning **13**
Automatic Selection of Collector, Heap Sizes, and Virtual Machine **13**
Behavior-based Parallel Collector Tuning **14**
6 Recommendations **15**
When to Select a Different Garbage Collector **15**
Heap Sizing **15**
Tuning Strategy for the Parallel Collector **16**
What to Do about OutOfMemoryError **16**
7 Tools to Evaluate Garbage Collection Performance **17**
–XX:+PrintGCDetails Command Line Option **17**
–XX:+PrintGCTimeStamps Command Line Option **17**
jmap **17**
jstat **17**
HPROF: Heap Profiler **18**
HAT: Heap Analysis Tool **18**
8 Key Options Related to Garbage Collection **18**
9 For More Information **20**

1 Introduction

One strength of the Java™ 2 Platform, Standard Edition (J2SE™) is that it performs automatic memory management, thereby shielding the developer from the complexity of explicit memory management.

This paper provides a broad overview of memory management in the Java HotSpot virtual machine (JVM) in Sun's J2SE 5.0 release. It describes the garbage collectors available to perform the memory management, and gives some advice regarding choosing and configuring a collector and setting sizes for the memory areas on which the collector operates. It also serves as a resource, listing some of the most commonly-used options that affect garbage collector behavior and providing numerous links to more detailed documentation.

Section 2 is for readers who are new to the concept of automatic memory management. It has a brief discussion of the benefits of such management versus requiring programmers to explicitly deallocate space for data. Section 3 then presents an overview of general garbage collection concepts, design choices, and performance metrics. It also introduces a commonly-used organization of memory into different areas called *generations* based on the expected lifetimes of objects. This separation into generations has proven effective at reducing garbage collection pause times and overall costs in a wide range of applications.

The rest of the paper provides information specific to the HotSpot JVM. Section 4 describes the four garbage collectors that are available, including one that is new in J2SE 5.0 update 6, and documents the generational memory organization they all utilize. For each collector, Section 4 summarizes the types of collection algorithms used and specifies when it would be appropriate to choose that collector.

Section 5 describes a technique new in the J2SE 5.0 release that combines (1) automatic selection of garbage collector, heap sizes, and HotSpot JVM (client or server) based on the platform and operating system on which the application is running, and (2) dynamic garbage collection tuning based on user-specified desired behavior. This technique is referred to as *ergonomics*.

Section 6 provides recommendations for selecting and configuring a garbage collector. It also provides some advice as to what to do about `OutOfMemoryError`s. Section 7 briefly describes some of the tools that can be utilized to evaluate garbage collection performance, and Section 8 lists the most commonly-used command line options that relate to garbage collector selection and behavior. Finally, Section 9 supplies links to more detailed documentation for the various topics covered by this paper.

2 Explicit vs. Automatic Memory Management

Memory management is the process of recognizing when allocated objects are no longer needed, deallocating (freeing) the memory used by such objects, and making it available for subsequent allocations. In some programming languages, memory management is the programmer's responsibility. The complexity of that task leads to many common errors that can cause unexpected or erroneous program behavior and crashes. As a result, a large proportion of developer time is often spent debugging and trying to correct such errors.

One problem that often occurs in programs with explicit memory management is *dangling references*. It is possible to deallocate the space used by an object to which some other object still has a reference. If the object with that (dangling) reference tries to access the original object, but the space has been reallocated to a new object, the result is unpredictable and not what was intended.

Another common problem with explicit memory management is *space leaks*. These leaks occur when memory is allocated and no longer referenced but is not released. For example, if you intend to free the space utilized by a linked list but you make the mistake of just deallocating the first element of the list, the remaining list elements are no longer referenced but they go out of the program's reach and can neither be used nor recovered. If enough leaks occur, they can keep consuming memory until all available memory is exhausted.

An alternate approach to memory management that is now commonly utilized, especially by most modern object-oriented languages, is automatic management by a program called a *garbage collector*. Automatic memory management enables increased abstraction of interfaces and more reliable code.

Garbage collection avoids the dangling reference problem, because an object that is still referenced somewhere will never be garbage collected and so will not be considered free. Garbage collection also solves the space leak problem described above since it automatically frees all memory no longer referenced.

3 Garbage Collection Concepts

A garbage collector is responsible for

- allocating memory
- ensuring that any referenced objects remain in memory, and
- recovering memory used by objects that are no longer reachable from references in executing code.

Objects that are referenced are said to be *live*. Objects that are no longer referenced are considered *dead* and are termed *garbage*. The process of finding and freeing (also known as *reclaiming*) the space used by these objects is known as *garbage collection*.

Garbage collection solves many, but not all, memory allocation problems. You could, for example, create objects indefinitely and continue referencing them until there is no more memory available. Garbage collection is also a complex task taking time and resources of its own.

The precise algorithm used to organize memory and allocate and deallocate space is handled by the garbage collector and hidden from the programmer. Space is commonly allocated from a large pool of memory referred to as the *heap*.

The timing of garbage collection is up to the garbage collector. Typically, the entire heap or a subpart of it is collected either when it fills up or when it reaches a threshold percentage of occupancy.

The task of fulfilling an allocation request, which involves finding a block of unused memory of a certain size in the heap, is a difficult one. The main problem for most dynamic memory allocation algorithms is to avoid fragmentation (see below), while keeping both allocation and deallocation efficient.

Desirable Garbage Collector Characteristics

A garbage collector must be both safe and comprehensive. That is, live data must never be erroneously freed, and garbage should not remain unclaimed for more than a small number of collection cycles.

It is also desirable that a garbage collector operate efficiently, without introducing long pauses during which the application is not running. However, as with most computer-related systems, there are often trade-offs between time, space, and frequency. For example, if a heap size is small, collection will be fast but the heap will fill up more quickly, thus requiring more frequent collections. Conversely, a large heap will take longer to fill up and thus collections will be less frequent, but they may take longer.

Another desirable garbage collector characteristic is the limitation of *fragmentation*. When the memory for garbage objects is freed, the free space may appear in small chunks in various areas such that there might not be enough space in any one contiguous area to be used for allocation of a large object. One approach to eliminating fragmentation is called *compaction*, discussed among the various garbage collector design choices below.

Scalability is also important. Allocation should not become a scalability bottleneck for multithreaded applications on multiprocessor systems, and collection should also not be such a bottleneck.

Design Choices

A number of choices must be made when designing or selecting a garbage collection algorithm:

- **Serial versus Parallel**
  With *serial* collection, only one thing happens at a time. For example, even when multiple CPUs are available, only one is utilized to perform the collection. When *parallel* collection is used, the task of garbage collection is split into parts and those subparts are executed simultaneously, on different CPUs. The simultaneous operation enables the collection to be done more quickly, at the expense of some additional complexity and potential fragmentation.

- **Concurrent versus Stop-the-world**
  When *stop-the-world* garbage collection is performed, execution of the application is completely suspended during the collection. Alternatively, one or more garbage collection tasks can be executed *concurrently*, that is, simultaneously, with the application. Typically, a concurrent garbage collector does most of its work concurrently, but may also occasionally have to do a few short stop-the-world pauses. Stop-the-world garbage collection is simpler than concurrent collection, since the heap is frozen and objects are not changing during the collection. Its disadvantage is that it may be undesirable for some applications to be paused. Correspondingly, the pause times are shorter when garbage collection is done concurrently, but the collector must take extra care, as it is operating over objects that might be updated at the same time by the application. This adds some overhead to concurrent collectors that affects performance and requires a larger heap size.

- **Compacting versus Non-compacting versus Copying**
  After a garbage collector has determined which objects in memory are live and which are garbage, it can *compact* the memory, moving all the live objects together and completely reclaiming the remaining memory. After compaction, it is easy and fast to allocate a new object at the first free location. A simple pointer can be utilized to keep track of the next location available for object allocation. In contrast with a compacting collector, a *non-compacting* collector releases the space utilized by garbage objects *in-place*, i.e., it does not move all live objects to create a large reclaimed region in the same way a compacting collector does. The benefit is faster completion of garbage collection, but the drawback is potential fragmentation. In general, it is more expensive to allocate from a heap with in-place deallocation than from a compacted heap. It may be necessary to search the heap for a contiguous area of memory sufficiently large to accommodate the new object. A third alternative is a *copying* collector, which copies (or *evacuates*) live objects to a different memory area. The benefit is that the source area can then be considered empty and available for fast and easy subsequent allocations, but the drawback is the additional time required for copying and the extra space that may be required.

Performance Metrics

Several metrics are utilized to evaluate garbage collector performance, including:

- **Throughput**—the percentage of total time not spent in garbage collection, considered over long periods of time.
- **Garbage collection overhead**—the inverse of throughput, that is, the percentage of total time spent in garbage collection.
- **Pause time**—the length of time during which application execution is stopped while garbage collection is occurring.
- **Frequency of collection**—how often collection occurs, relative to application execution.
- **Footprint**—a measure of size, such as heap size.
- **Promptness**—the time between when an object becomes garbage and when the memory becomes available.

An interactive application might require low pause times, whereas overall execution time is more important to a non-interactive one. A real-time application would demand small upper bounds on both garbage collection pauses and the proportion of time spent in the collector in any period. A small footprint might be the main concern of an application running in a small personal computer or embedded system.

Generational Collection

When a technique called *generational collection* is used, memory is divided into *generations*, that is, separate pools holding objects of different ages. For example, the most widely-used configuration has two generations: one for young objects and one for old objects.

Different algorithms can be used to perform garbage collection in the different generations, each algorithm optimized based on commonly observed characteristics for that particular generation. Generational garbage collection exploits the following observations, known as the *weak generational hypothesis*, regarding applications written in several programming languages, including the Java programming language:

- Most allocated objects are not referenced (considered live) for long, that is, they die young.
- Few references from older to younger objects exist.

Young generation collections occur relatively frequently and are efficient and fast because the young generation space is usually small and likely to contain a lot of objects that are no longer referenced.

Objects that survive some number of young generation collections are eventually *promoted*, or *tenured*, to the old generation. See Figure 1. This generation is typically larger than the young generation and its occupancy grows more slowly. As a result, old generation collections are infrequent, but take significantly longer to complete.

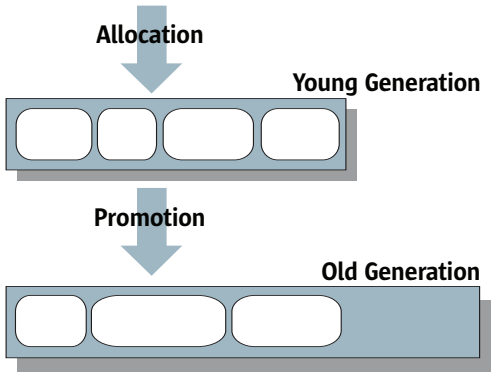

Figure 1. Generational garbage collection

The garbage collection algorithm chosen for a young generation typically puts a premium on speed, since young generation collections are frequent. On the other hand, the old generation is typically managed by an algorithm that is more space efficient, because the old generation takes up most of the heap and old generation algorithms have to work well with low garbage densities.

4 Garbage Collectors in the J2SE 5.0 HotSpot JVM

The Java HotSpot virtual machine includes four garbage collectors as of J2SE 5.0 update 6. All the collectors are generational. This section describes the generations and the types of collections, and discusses why object allocations are often fast and efficient. It then provides detailed information about each collector.

HotSpot Generations

Memory in the Java HotSpot virtual machine is organized into three generations: a young generation, an old generation, and a permanent generation. Most objects are initially allocated in the *young generation*. The *old generation* contains objects that have survived some number of young generation collections, as well as some large objects that may be allocated directly in the old generation. The *permanent generation* holds objects that the JVM finds convenient to have the garbage collector manage, such as objects describing classes and methods, as well as the classes and methods themselves.

The young generation consists of an area called *Eden* plus two smaller *survivor spaces*, as shown in Figure 2. Most objects are initially allocated in Eden. (As mentioned, a few large objects may be allocated directly in the old generation.) The survivor spaces hold objects that have survived at least one young generation collection and have thus been given additional chances to die before being considered "old enough" to be promoted to the old generation. At any given time, one of the survivor spaces (labeled *From* in the figure) holds such objects, while the other is empty and remains unused until the next collection.

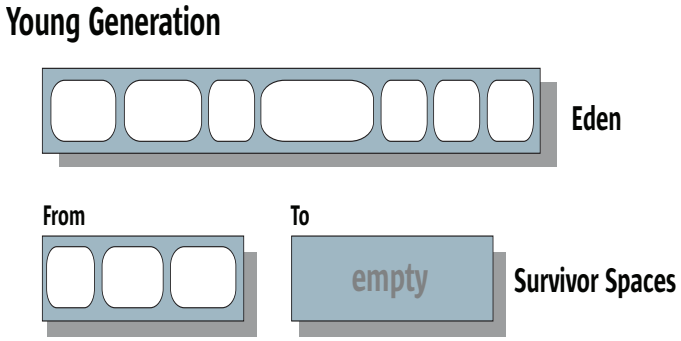

Figure 2. Young generation memory areas

Garbage Collection Types

When the young generation fills up, a *young generation collection* (sometimes referred to as a *minor collection*) of just that generation is performed. When the old or permanent generation fills up, what is known as a *full collection* (sometimes referred to as a *major collection*) is typically done. That is, all generations are collected. Commonly, the young generation is collected first, using the collection algorithm designed specifically for that generation, because it is usually the most efficient algorithm for identifying garbage in the young generation. Then what is referred to below as the *old generation collection algorithm* for a given collector is run on both the old and permanent generations. If compaction occurs, each generation is compacted separately.

Sometimes the old generation is too full to accept all the objects that would be likely to be promoted from the young generation to the old generation if the young generation was collected first. In that case, for all but the CMS collector, the young generation collection algorithm is not run. Instead, the old generation collection algorithm is used on the entire heap. (The CMS old generation algorithm is a special case because it cannot collect the young generation.)

Fast Allocation

As you will see from the garbage collector descriptions below, in many cases there are large contiguous blocks of memory available from which to allocate objects. Allocations from such blocks are efficient, using a simple *bump-the-pointer* technique. That is, the end of the previously allocated object is always kept track of. When a new allocation request needs to be satisfied, all that needs to be done is to check whether the object will fit in the remaining part of the generation and, if so, to update the pointer and initialize the object.

For multithreaded applications, allocation operations need to be multithread-safe. If global locks were used to ensure this, then allocation into a generation would become a bottleneck and degrade performance. Instead, the HotSpot JVM has adopted a technique called *Thread-Local Allocation Buffers* (TLABs). This improves multithreaded allocation throughput by giving each thread its own buffer (i.e., a small portion of the generation) from which to allocate. Since only one thread can be allocating into each TLAB, allocation can take place quickly by utilizing the bump-the-pointer technique, without requiring any locking. Only infrequently, when a thread fills up its TLAB and needs to get a new one, must synchronization be utilized. Several techniques to minimize space wastage due to the use of TLABs are employed. For example, TLABs are sized by the allocator to waste less than 1% of Eden, on average. The combination of the use of TLABs and linear allocations using the bump-the-pointer technique enables each allocation to be efficient, only requiring around 10 native instructions.

Serial Collector

With the *serial collector*, both young and old collections are done serially (using a single CPU), in a stop-the-world fashion. That is, application execution is halted while collection is taking place.

Young Generation Collection Using the Serial Collector

Figure 3 illustrates the operation of a young generation collection using the serial collector. The live objects in Eden are copied to the initially empty survivor space, labeled *To* in the figure, except for ones that are too large to fit comfortably in the *To* space. Such objects are directly copied to the old generation. The live objects in the occupied survivor space (labeled *From*) that are still relatively young are also copied to the other survivor space, while objects that are relatively old are copied to the old generation. Note: If the *To* space becomes full, the live objects from Eden or *From* that have not been copied to it are tenured, regardless of how many young generation collections they have survived. Any objects remaining in Eden or the *From* space after live objects have been copied are, by definition, not live, and they do not need to be examined. (These garbage objects are marked with an X in the figure, though in fact the collector does not examine or mark these objects.)

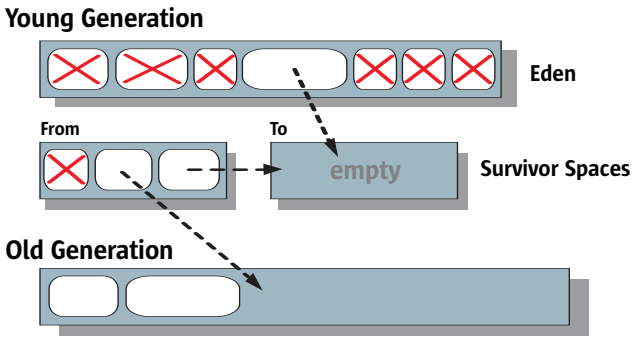

Figure 3. Serial young generation collection

After a young generation collection is complete, both Eden and the formerly occupied survivor space are empty and only the formerly empty survivor space contains live objects. At this point, the survivor spaces swap roles. See Figure 4.

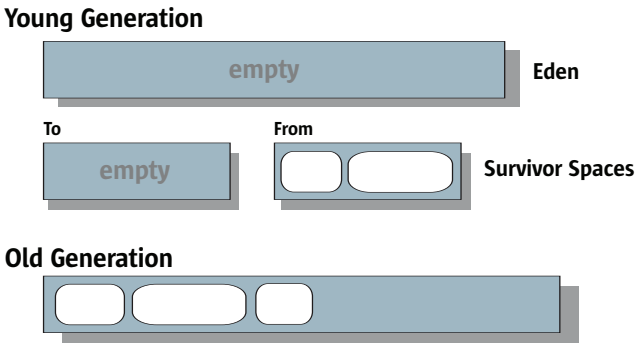

Figure 4. After a young generation collection

Old Generation Collection Using the Serial Collector

With the serial collector, the old and permanent generations are collected via a *mark-sweep-compact* collection algorithm. In the mark phase, the collector identifies which objects are still live. The sweep phase "sweeps" over the generations, identifying garbage. The collector then performs *sliding compaction*, sliding the live objects towards the beginning of the old generation space (and similarly for the permanent generation), leaving any free space in a single contiguous chunk at the opposite end. See Figure 5. The compaction allows any future allocations into the old or permanent generation to use the fast, bump-the-pointer technique.

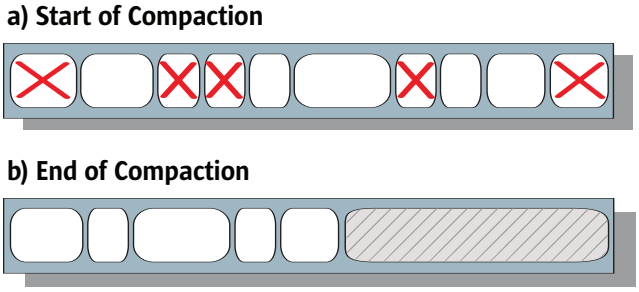

Figure 5. Compaction of the old generation

When to Use the Serial Collector

The serial collector is the collector of choice for most applications that are run on client-style machines and that do not have a requirement for low pause times. On today's hardware, the serial collector can efficiently manage a lot of nontrivial applications with 64MB heaps and relatively short worst-case pauses of less than half a second for full collections.

Serial Collector Selection

In the J2SE 5.0 release, the serial collector is automatically chosen as the default garbage collector on machines that are not server-class machines, as described in Section 5. On other machines, the serial collector can be explicitly requested by using the `-XX:+UseSerialGC` command line option.

Parallel Collector

These days, many Java applications run on machines with a lot of physical memory and multiple CPUs. The *parallel collector*, also known as the *throughput collector*, was developed in order to take advantage of available CPUs rather than leaving most of them idle while only one does garbage collection work.

Young Generation Collection Using the Parallel Collector

The parallel collector uses a parallel version of the young generation collection algorithm utilized by the serial collector. It is still a stop-the-world and copying collector, but performing the young generation

collection in parallel, using many CPUs, decreases garbage collection overhead and hence increases application throughput. Figure 6 illustrates the differences between the serial collector and the parallel collector for the young generation.

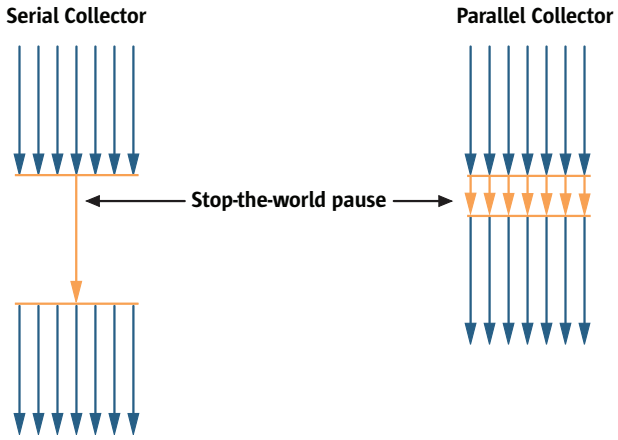

Figure 6. Comparison between serial and parallel young generation collection

Old Generation Collection Using the Parallel Collector
Old generation garbage collection for the parallel collector is done using the same serial mark-sweep-compact collection algorithm as the serial collector.

When to Use the Parallel Collector
Applications that can benefit from the parallel collector are those that run on machines with more than one CPU and do not have pause time constraints, since infrequent, but potentially long, old generation collections will still occur. Examples of applications for which the parallel collector is often appropriate include those that do batch processing, billing, payroll, scientific computing, and so on.

You may want to consider choosing the parallel compacting collector (described next) over the parallel collector, since the former performs parallel collections of all generations, not just the young generation.

Parallel Collector Selection
In the J2SE 5.0 release, the parallel collector is automatically chosen as the default garbage collector on server-class machines (defined in Section 5). On other machines, the parallel collector can be explicitly requested by using the `-XX:+UseParallelGC` command line option.

Parallel Compacting Collector

The *parallel compacting collector* was introduced in J2SE 5.0 update 6. The difference between it and the parallel collector is that it uses a new algorithm for old generation garbage collection. Note: Eventually, the parallel compacting collector will replace the parallel collector.

Young Generation Collection Using the Parallel Compacting Collector
Young generation garbage collection for the parallel compacting collector is done using the same algorithm as that for young generation collection using the parallel collector.

Old Generation Collection Using the Parallel Compacting Collector
With the parallel compacting collector, the old and permanent generations are collected in a stop-the-world, mostly parallel fashion with sliding compaction. The collector utilizes three phases. First, each generation is logically divided into fixed-sized regions. In the *marking phase*, the initial set of live objects directly reachable from the application code is divided among garbage collection threads, and then all live objects are marked in parallel. As an object is identified as live, the data for the region it is in is updated with information about the size and location of the object.

The s*ummary phase* operates on regions, not objects. Due to compactions from previous collections, it is typical that some portion of the left side of each generation will be dense, containing mostly live objects. The amount of space that could be recovered from such dense regions is not worth the cost of compacting them. So the first thing the summary phase does is examine the density of the regions, starting with the leftmost one, until it reaches a point where the space that could be recovered from a region and those to the right of it is worth the cost of compacting those regions. The regions to the left of that point are referred to as the *dense prefix*, and no objects are moved in those regions. The regions to the right of that point will be compacted, eliminating all dead space. The summary phase calculates and stores the new location of the first byte of live data for each compacted region. Note: The summary phase is currently implemented as a serial phase; parallelization is possible but not as important to performance as parallelization of the marking and compaction phases.

In the *compaction phase*, the garbage collection threads use the summary data to identify regions that need to be filled, and the threads can independently copy data into the regions. This produces a heap that is densely packed on one end, with a single large empty block at the other end.

When to Use the Parallel Compacting Collector
As with the parallel collector, the parallel compacting collector is beneficial for applications that are run on machines with more than one CPU. In addition, the parallel operation of old generation collections reduces pause times and makes the parallel compacting collector more suitable than the parallel collector for applications that have pause time constraints. The parallel compacting collector might *not* be suitable for applications run on large shared machines (such as SunRays), where no single application should monopolize several CPUs for extended periods of time. On such machines, consider either decreasing the number of threads used for garbage collection (via the `-XX:ParallelGCThreads=`*n* command line option) or selecting a different collector.

Parallel Compacting Collector Selection
If you want the parallel compacting collector to be used, you must select it by specifying the command line option `-XX:+UseParallelOldGC`.

Concurrent Mark-Sweep (CMS) Collector

For many applications, end-to-end throughput is not as important as fast response time. Young generation collections do not typically cause long pauses. However, old generation collections, though infrequent, can impose long pauses, especially when large heaps are involved. To address this issue, the HotSpot JVM includes a collector called the *concurrent mark-sweep (CMS) collector*, also known as the *low-latency collector*.

Young Generation Collection Using the CMS Collector
The CMS collector collects the young generation in the same manner as the parallel collector.

Old Generation Collection Using the CMS Collector
Most of the collection of the old generation using the CMS collector is done concurrently with the execution of the application.

A collection cycle for the CMS collector starts with a short pause, called the *initial mark*, that identifies the initial set of live objects directly reachable from the application code. Then, during the *concurrent marking phase*, the collector marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields while the marking phase is taking place, not all live objects are guaranteed to be marked at the end of the concurrent marking phase. To handle this, the application stops again for a second pause, called *remark*, which finalizes marking by revisiting any objects that were modified during the concurrent marking phase. Because the remark pause is more substantial than the initial mark, multiple threads are run in parallel to increase its efficiency.

At the end of the remark phase, all live objects in the heap are guaranteed to have been marked, so the subsequent *concurrent sweep phase* reclaims all the garbage that has been identified. Figure 7 illustrates the differences between old generation collection using the serial mark-sweep-compact collector and the CMS collector.

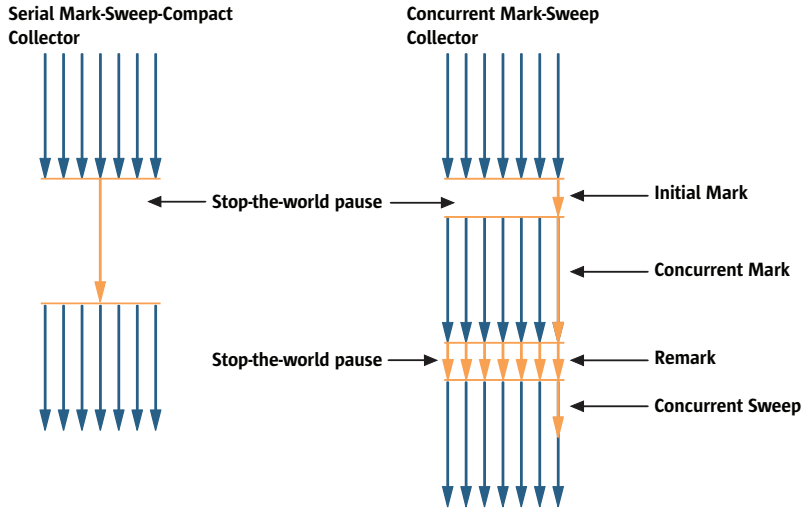

Figure 7. Comparison between serial and CMS old generation collection

Since some tasks, such as revisiting objects during the remark phase, increase the amount of work the collector has to do, its overhead increases as well. This is a typical trade-off for most collectors that attempt to reduce pause times.

The CMS collector is the only collector that is non-compacting. That is, after it frees the space that was occupied by dead objects, it does not move the live objects to one end of the old generation. See Figure 8.

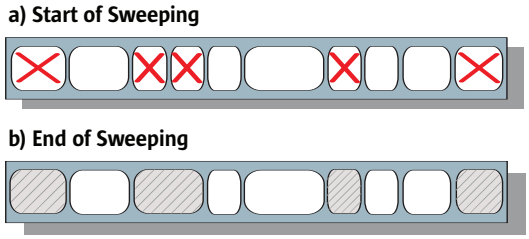

Figure 8. CMS sweeping (but not compacting) of old generation

This saves time, but since the free space is not contiguous, the collector can no longer use a simple pointer indicating the next free location into which the next object can be allocated. Instead, it now needs to employ free lists. That is, it creates some number of lists linking together unallocated regions of memory, and each time an object needs to be allocated, the appropriate list (based on the amount of memory needed) must be searched for a region large enough to hold the object As a result, allocations into the old generation are more expensive than they are with a simple bump-the-pointer technique. This also imposes extra overhead to young generation collections, as most allocations in the old generation occur when objects are promoted during young generation collections.

Another disadvantage the CMS collector has is a requirement for larger heap sizes than the other collectors. Given that the application is allowed to run during the marking phase, it can continue to allocate memory, thereby potentially continuing to grow the old generation. Additionally, although the collector guarantees to identify all live objects during a marking phase, some objects may become garbage during that phase and they will not be reclaimed until the next old generation collection. Such objects are referred to as *floating garbage*.

Finally, fragmentation may occur due to lack of compaction. To deal with fragmentation, the CMS collector tracks popular object sizes, estimates future demand, and may split or join free blocks to meet demand.

Unlike the other collectors, the CMS collector does not start an old generation collection when the old generation becomes full. Instead, it attempts to start a collection early enough so that it can complete before that happens. Otherwise, the CMS collector reverts to the more time-consuming stop-the-world mark-sweep-compact algorithm used by the parallel and serial collectors. To avoid this, the CMS collector starts at a time based on statistics regarding previous collection times and how quickly the old generation becomes occupied. The CMS collector will also start a collection if the occupancy of the old generation exceeds something called the initiating occupancy. The value of the initiating occupancy is set by the command line option `-XX:CMSInitiatingOccupancyFraction=`*n*, where *n* is a percentage of the old generation size. The default is 68.

In summary, compared to the parallel collector, the CMS collector decreases old generation pauses—sometimes dramatically—at the expense of slightly longer young generation pauses, some reduction in throughput, and extra heap size requirements.

Incremental Mode
The CMS collector can be used in a mode in which the concurrent phases are done incrementally. This mode is meant to lessen the impact of long concurrent phases by periodically stopping the concurrent phase to yield back processing to the application. The work done by the collector is divided into small chunks of time that are scheduled between young generation collections. This feature is useful when applications that need the low pause times provided by the concurrent collector are run on machines with small numbers of processors (e.g., 1 or 2). For more information on usage of this mode, see the "Tuning Garbage Collection with the 5.0 Java™ Virtual Machine" paper referred to in Section 9.

When to Use the CMS Collector
Use the CMS collector if your application needs shorter garbage collection pauses and can afford to share processor resources with the garbage collector when the application is running. (Due to its concurrency, the CMS collector takes CPU cycles away from the application during a collection cycle.) Typically, applications that have a relatively large set of long-lived data (a large old generation), and that run on machines with two or more processors, tend to benefit from the use of this collector. An example would be web servers. The CMS collector should be considered for any application with a low pause time requirement. It may also give good results for interactive applications with old generations of a modest size on a single processor.

CMS Collector Selection
If you want the CMS collector to be used, you must explicitly select it by specifying the command line option `-XX:+UseConcMarkSweepGC`. If you want it to be run in incremental mode, also enable that mode via the `-XX:+CMSIncrementalMode` option.

5 Ergonomics – Automatic Selections and Behavior Tuning

In the J2SE 5.0 release, default values for the garbage collector, heap size, and HotSpot virtual machine (client or server) are automatically chosen based on the platform and operating system on which the application is running. These automatic selections better match the needs of different types of applications, while requiring fewer command line options than in previous releases.

In addition, a new way of dynamically tuning collection has been added for the parallel garbage collectors. With this approach, the user specifies the desired behavior, and the garbage collector dynamically tunes the sizes of the heap regions in an attempt to achieve the requested behavior. The combination of platform-dependent default selections and garbage collection tuning that uses desired behavior is referred to as *ergonomic*s. The goal of ergonomics is to provide good performance from the JVM with a minimum of command line tuning.

Automatic Selection of Collector, Heap Sizes, and Virtual Machine

A server-class machine is defined to be one with

- 2 or more physical processors and
- 2 or more gigabytes of physical memory

This definition of a server-class machine applies to all platforms, with the exception of 32-bit platforms running a version of the Windows operating system.

On machines that are not server-class machines, the default values for JVM, garbage collector, and heap sizes are

- the client JVM
- the serial garbage collector
- Initial heap size of 4MB
- Maximum heap size of 64MB

On a server-class machine, the JVM is always the server JVM unless you explicitly specify the `-client` command line option to request the client JVM. On a server-class machine running the server JVM, the default garbage collector is the parallel collector. Otherwise, the default is the serial collector.

On a server-class machine running either JVM (client or server) with the parallel garbage collector, the default initial and maximum heap sizes are

- Initial heap size of 1/64th of the physical memory, up to 1GB. (Note that the minimum initial heap size is 32MB, since a server-class machine is defined to have at least 2GB of memory and 1/64th of 2GB is 32MB.)
- Maximum heap size of 1/4th of the physical memory, up to 1GB.

Otherwise, the same default sizes as for non-server-class machines are used (4MB initial heap size and 64MB maximum heap size). Default values can always be overridden by command line options. Relevant options are shown in Section 8.

Behavior-based Parallel Collector Tuning

In the J2SE 5.0 release, a new method of tuning has been added for the parallel garbage collectors, based on desired behavior of the application with respect to garbage collection. Command line options are used to specify the desired behavior in terms of goals for maximum pause time and application throughput.

Maximum Pause Time Goal

The maximum pause time goal is specified with the command line option

`-XX:MaxGCPauseMillis=`*n*

This is interpreted as a hint to the parallel collector that pause times of *n* milliseconds or less are desired. The parallel collector will adjust the heap size and other garbage collection-related parameters in an attempt to keep garbage collection pauses shorter than *n* milliseconds. These adjustments may cause the garbage collector to reduce overall throughput of the application, and in some cases the desired pause time goal cannot be met.

The maximum pause time goal is applied to each generation separately. Typically, if the goal is not met, the generation is made smaller in an attempt to meet the goal. No maximum pause time goal is set by default.

Throughput Goal

The throughput goal is measured in terms of the time spent doing garbage collection and the time spent outside of garbage collection (referred to as application time). The goal is specified by the command line option

`-XX:GCTimeRatio=`*n*

The ratio of garbage collection time to application time is

`1 / (1 +` *n*`)`

For example `-XX:GCTimeRatio=19` sets a goal of 5% of the total time for garbage collection. The default goal is 1% (i.e. *n*= 99). The time spent in garbage collection is the total time for all generations. If the throughput goal is not being met, the sizes of the generations are increased in an effort to increase the time the application can run between collections. A larger generation takes more time to fill up.

Footprint Goal
If the throughput and maximum pause time goals have been met, the garbage collector reduces the size of the heap until one of the goals (invariably the throughput goal) cannot be met. The goal that is not being met is then addressed.

Goal Priorities
The parallel garbage collectors attempt to meet the maximum pause time goal first. Only after it is met do they address the throughput goal. Similarly, the footprint goal is considered only after the first two goals have been met.

6 Recommendations

The ergonomics described in the previous section lead to automatic garbage collector, virtual machine, and heap size selections that are reasonable for a large percentage of applications. Thus, the initial recommendation for selecting and configuring a garbage collector is to do nothing! That is, do not specify usage of a particular garbage collector, etc. Let the system make automatic choices based on the platform and operating system on which your application is running. Then test your application. If its performance is acceptable, with sufficiently high throughput and sufficiently low pause times, you are done. You don't need to troubleshoot or modify garbage collector options.

On the other hand, if your application seems to have performance problems related to garbage collection, then the easiest thing you can do first is think about whether the garbage collector selected by default is appropriate, given your application and platform characteristics. If not, explicitly select the collector that you think is appropriate, and see whether the performance becomes acceptable.

You can measure and analyze performance using tools such as those described in Section 7. Based on the results, you can consider modifying options, such as those that control heap sizes or garbage collection behavior. Some of the most commonly-specified options are shown in Section 8. Please note: The best approach to performance tuning is to measure first, then tune. Measure using tests relevant for how your code will actually be used. Also, beware of over-optimizing, since application data sets, hardware, and so on—even the garbage collector implementation!—may change over time.

This section provides information on selecting a garbage collector and specifying heap sizes. Then it provides suggestions for tuning the parallel garbage collectors, and gives some advice regarding what to do about `OutOfMemoryError`s.

When to Select a Different Garbage Collector

Section 4 tells, for each collector, the situations in which usage of that collector is recommended. Section 5 describes the platforms on which either the serial or the parallel collector is automatically chosen by default. If your application or environmental characteristics are such that a different collector than the default is warranted, explicitly request that collector via one of the following command line options:

```
-XX:+UseSerialGC
-XX:+UseParallelGC
-XX:+UseParallelOldGC
-XX:+UseConcMarkSweepGC
```

Heap Sizing

Section 5 tells what the default initial and maximum heap sizes are. Those sizes may be fine for many applications, but if your analysis of a performance problem (see Section 7) or of an `OutOfMemoryError` (discussed later in this section) indicates a problem with the size of a particular generation or of the entire heap, you can modify the sizes via command line options specified in Section 8. For example, the default maximum heap size of 64MB on non-server-class machines is often too small, so you can specify a larger size via the `-Xmx` option. Unless you have problems with long pause times, try granting as much memory as possible to the heap. Throughput is proportional to the amount of memory available. Having sufficient available memory is the most

important factor affecting garbage collection performance.

After deciding the total amount of memory you can afford to give to the total heap, you can then consider adjusting the sizes of the different generations. The second most influential factor affecting garbage collection performance is the proportion of the heap dedicated to the young generation. Unless you find problems with excessive old generation collections or pause times, grant plenty of memory to the young generation. However, when you're using the serial collector, do not grant the young generation more than half the total heap size.

When you are using one of the parallel garbage collectors, it is preferable to specify desired behavior rather than exact heap size values. Let the collector automatically and dynamically modify the heap sizes in order to achieve that behavior, as described next.

Tuning Strategy for the Parallel Collector

If the garbage collector chosen (automatically or explicitly) is the parallel collector or parallel compacting collector, then go ahead and specify a throughput goal (see Section 5) that is sufficient for your application. Do not choose a maximum value for the heap unless you know that you need a heap greater than the default maximum heap size. The heap will grow or shrink to a size that will support the chosen throughput goal. Some oscillations in the heap size during initialization and during a change in the application behavior can be expected.

If the heap grows to its maximum, in most cases that means that the throughput goal cannot be met within that maximum size. Set the maximum size to a value that is close to the total physical memory on the platform but that does not cause swapping of the application. Execute the application again. If the throughput goal is still not met, then the goal for the application time is too high for the available memory on the platform.

If the throughput goal can be met, but there are pauses that are too long, select a maximum pause time goal. Choosing a maximum pause time goal may mean that your throughput goal will not be met, so choose values that are an acceptable compromise for the application.

The size of the heap will oscillate as the garbage collector tries to satisfy competing goals, even if the application has reached a steady state. The pressure to achieve a throughput goal (which may require a larger heap) competes with the goals for a maximum pause time and a minimum footprint (which both may require a smaller heap).

What to Do about OutOfMemoryError

One common issue that many developers have to address is that of applications that terminate with `java.lang.OutOfMemoryError`. That error is thrown when there is insufficient space to allocate an object. That is, garbage collection cannot make any further space available to accommodate a new object, and the heap cannot be further expanded. An `OutOfMemoryError` does not necessarily imply a memory leak. The issue might simply be a configuration issue, for example if the specified heap size (or the default size if not specified) is insufficient for the application.

The first step in diagnosing an `OutOfMemoryError` is to examine the full error message. In the exception message, further information is supplied after "`java.lang.OutOfMemoryError`". Here are some common examples of what that additional information may be, what it may mean, and what to do about it:

- `Java heap space`
  This indicates that an object could not be allocated in the heap. The issue may be just a configuration problem. You could get this error, for example, if the maximum heap size specified by the `-Xmx` command line option (or selected by default) is insufficient for the application. It could also be an indication that objects that are no longer needed cannot be garbage collected because the application is unintentionally holding references to them. The HAT tool (see Section 7) can be used to view all reachable objects and understand which references are keeping each one alive. One other potential source of this error could be the excessive use of finalizers by the application such that the thread to invoke the finalizers cannot keep up with the rate of addition of finalizers to the queue. The *jconsole* management tool can be used to monitor the number of objects that are pending finalization.

- `PermGen space`
  This indicates that the permanent generation is full. As described earlier, that is the area of the heap where the JVM stores its metadata. If an application loads a large number of classes, then the permanent generation may need to be increased. You can do so by specifying the command line option `-XX:MaxPermSize=`*n*, where *n* specifies the size.
- `Requested array size exceeds VM limit`
  This indicates that the application attempted to allocate an array that is larger than the heap size. For example, if an application tries to allocate an array of 512MB but the maximum heap size is 256MB, then this error will be thrown. In most cases the problem is likely to be either that the heap size is too small or that a bug results in the application attempting to create an array whose size is calculated to be incorrectly huge.

Some of the tools described in Section 7 can be utilized to diagnose `OutOfMemoryError` problems. A few of the most useful tools for this task are the Heap Analysis Tool (HAT), the *jconsole* management tool, and the *jmap* tool with the *–histo* option.

7 Tools to Evaluate Garbage Collection Performance

Various diagnostic and monitoring tools can be utilized to evaluate garbage collection performance. This section provides a brief overview of some of them. For more information, see the "Tools and Troubleshooting" links in Section 9.

–XX:+PrintGCDetails Command Line Option

One of the easiest ways to get initial information about garbage collections is to specify the command line option `-XX:+PrintGCDetails`. For every collection, this results in the output of information such as the size of live objects before and after garbage collection for the various generations, the total available space for each generation, and the length of time the collection took.

–XX:+PrintGCTimeStamps Command Line Option

This outputs a timestamp at the start of each collection, in addition to the information that is output if the command line option `-XX:+PrintGCDetails` is used. The timestamps can help you correlate garbage collection logs with other logged events.

jmap

jmap is a command line utility included in the Solaris™ Operating Environment and Linux (but not Windows) releases of the Java Development Kit (JDK™). It prints memory–related statistics for a running JVM or core file. If it is used without any command line options, then it prints the list of shared objects loaded, similar to what the Solaris *pmap* utility outputs. For more specific information, the *–heap*, *–histo*, or *–permstat* options can be used.

The *–heap* option is used to obtain information that includes the name of the garbage collector, algorithm–specific details (such as the number of threads being used for parallel garbage collection), heap configuration information, and a heap usage summary

The *–histo* option can be used to obtain a class–wise histogram of the heap. For each class, it prints the number of instances in the heap, the total amount of memory consumed by those objects in bytes, and the fully qualified class name. The histogram is useful when trying to understand how the heap is used.

Configuring the size of the permanent generation can be important for applications that dynamically generate and load a large number of classes (Java Server Pages™ and web containers, for example). If an application loads "too many" classes, then an `OutOfMemoryError` is thrown. The *–permstat* option to the *jmap* command can be used to get statistics for the objects in the permanent generation.

jstat

The *jstat* utility uses the built–in instrumentation in the HotSpot JVM to provide information on performance and resource consumption of running applications. The tool can be used when diagnosing performance issues, and in particular issues related to heap sizing and garbage collection. Some of its many options can print statistics regarding garbage collection behavior and the capacities and usage of the various generations.

HPROF: Heap Profiler

HPROF is a simple profiler agent shipped with JDK 5.0. It is a dynamically–linked library that interfaces to the JVM using the Java Virtual Machine Tools Interface (JVM TI). It writes out profiling information either to a file or to a socket in ASCII or binary format. This information can be further processed by a profiler front–end tool.

HPROF is capable of presenting CPU usage, heap allocation statistics, and monitor contention profiles. In addition, it can output complete heap dumps and report the states of all the monitors and threads in the Java virtual machine. HPROF is useful when analyzing performance, lock contention, memory leaks, and other issues. See Section 9 for a link to HPROF documentation.

HAT: Heap Analysis Tool

The Heap Analysis Tool (HAT) helps debug *unintentional object retention*. This term is used to describe an object that is no longer needed but is kept alive due to references through some path from a live object. HAT provides a convenient means to browse the object topology in a heap snapshot that is generated using HPROF. The tool allows a number of queries, including "show me all reference paths from the rootset to this object." See Section 9 for a link to HAT documentation.

8 Key Options Related to Garbage Collection

A number of command line options can be used to select a garbage collector, specify heap or generation sizes, modify garbage collection behavior, and obtain garbage collection statistics. This section shows some of the most commonly–used options. For a more complete list and detailed information regarding the various options available, see Section 9. Note: Numbers you specify can end with "m" or "M" for megabytes, "k" or "K" for kilobytes, and "g" or "G" for gigabytes.

Garbage Collector Selection

Option	Garbage Collector Selected
–XX:+UseSerialGC	Serial
–XX:+UseParallelGC	Parallel
–XX:+UseParallelOldGC	Parallel compacting
–XX:+UseConcMarkSweepGC	Concurrent mark–sweep (CMS)

Garbage Collector Statistics

Option	Description
–XX:+PrintGC	Outputs basic information at every garbage collection.
–XX:+PrintGCDetails	Outputs more detailed information at every garbage collection.
–XX:+PrintGCTimeStamps	Outputs a time stamp at the start of each garbage collection event. Used with –XX:+PrintGC or –XX:+PrintGCDetails to show when each garbage collection begins.

Heap and Generation Sizes

Option	Default	Description
–Xms*n*	See Section 5	Initial size, in bytes, of the heap.
–Xmx*n*	See Section 5	Maximum size, in bytes, of the heap.
–XX:MinHeapFreeRatio=*minimum* and –XX:MaxHeapFreeRatio=*maximum*	40 (min) 70 (max)	Target range for the proportion of free space to total heap size. These are applied per generation. For example, if minimum is 30 and the percent of free space in a generation falls below 30%, the size of the generation is expanded so as to have 30% of the space free. Similarly, if maximum is 60 and the percent of free space exceeds 60%, the size of the generation is shrunk so as to have only 60% of the space free.
–XX:NewSize=*n*	Platform–dependent	Default initial size of the new (young) generation, in bytes.
–XX:NewRatio=*n*	2 on client JVM, 8 on server JVM	Ratio between the young and old generations. For example, if *n* is 3, then the ratio is 1:3 and the combined size of Eden and the survivor spaces is one fourth of the total size of the young and old generations.
–XX:SurvivorRatio=*n*	32	Ratio between each survivor space and Eden. For example, if *n* is 7, each survivor space is one–ninth of the young generation (not one–eighth, because there are two survivor spaces).
–XX:MaxPermSize=*n*	Platform–dependent	Maximum size of the permanent generation.

Options for the Parallel and Parallel Compacting Collectors

Option	Default	Description
–XX:ParallelGCThreads=*n*	The number of CPUs	Number of garbage collector threads.
–XX:MaxGCPauseMillis=*n*	No default	Indicates to the collector that pause times of *n* milliseconds or less are desired.
–XX:GCTimeRatio=*n*	99	Number that sets a goal that 1/(1+*n*) of the total time be spent on garbage collection.

Options for the CMS Collector

Option	Default	Description
–XX:+CMSIncrementalMode	Disabled	Enables a mode in which the concurrent phases are done incrementally, periodically stopping the concurrent phase to yield back the processor to the application.
–XX:+CMSIncrementalPacing	Disabled	Enables automatic control of the amount of work the CMS collector is allowed to do before giving up the processor, based on application behavior.
–XX:ParallelGCThreads=*n*	The number of CPUs	Number of garbage collector threads for the parallel young generation collections and for the parallel parts of the old generation collections.

9 For More Information

HotSpot Garbage Collection and Performance Tuning

- *Garbage Collection in the Java HotSpot Virtual Machine*
  (http://www.devx.com/Java/Article/21977)
- *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*
  (http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html)

Ergonomics

- *Server–Class Machine Detection*
  (http://java.sun.com/j2se/1.5.0/docs/guide/vm/server–class.html)
- *Garbage Collector Ergonomics*
  (http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc–ergonomics.html)
- *Ergonomics in the 5.0 Java™ Virtual Machine*
  (http://java.sun.com/docs/hotspot/gc5.0/ergo5.html)

Options

- *Java™ HotSpot VM Options*
  (http://java.sun.com/docs/hotspot/VMOptions.html)
- *Solaris and Linux options*
  (http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html)
- *Windows options*
  (http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/java.html)

Tools and Troubleshooting

- *Java™ 2 Platform, Standard Edition 5.0 Trouble–Shooting and Diagnostic Guide*
  (http://java.sun.com/j2se/1.5/pdf/jdk50_ts_guide.pdf)
- *HPROF: A Heap/CPU Profiling Tool in J2SE 5.0*
  (http://java.sun.com/developer/technicalArticles/Programming/HPROF.html)
- *Hat: Heap Analysis Tool*
  (https://hat.dev.java.net/)

Finalization

- *Finalization, threads, and the Java technology–based memory model*
  (http://devresource.hp.com/drc/resources/jmemmodel/index.jsp)
- *How to Handle Java Finalization's Memory–Retention Issues*
  (http://www.devx.com/Java/Article/30192)

Miscellaneous

- *J2SE 5.0 Release Notes*
  (http://java.sun.com/j2se/1.5.0/relnotes.html)
- *Java™ Virtual Machines*
  (http://java.sun.com/j2se/1.5.0/docs/guide/vm/index.html)
- *Sun Java™ Real–Time System (Java RTS)*
  (http://java.sun.com/j2se/realtime/index.jsp)
- General book on garbage collection: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* by Richard Jones and Rafael Lins, John Wiley & Sons, 1996.