Garbage Collection in the Java HotSpot Virtual Machine

Gain a better understanding of how garbage collection in the Java HotSpot VM works, and learn to take full advantage of it when designing, developing, and deploying your Java applications.

by Tony Printezis

The Java virtual machine (JVM) specification dictates that any JVM implementation must include a garbage collector (GC) to reclaim unused memory (i.e., unreachable objects) [JVMS2 1999]. However, the behavior and efficiency of a garbage collector can heavily influence the performance and responsiveness of any application that relies on it. This article gives an introduction to the garbage collection techniques that Sun Microsystems Inc. adopted in the Java HotSpot Virtual Machine, its production Java virtual machine. The aim is for the reader to gain a better understanding of how garbage collection in the Java HotSpot VM works and, as a result, be able to take full advantage of it when designing, developing, and deploying their applications.

"Heap storage for objects is reclaimed by an automatic storage management system (typically a garbage collector); objects are never explicitly deallocated."
— Java Virtual Machine Specification, Section 3.5.3 [JVMS2 1999]

Generational Garbage Collection
The Java HotSpot VM uses a generational garbage collector, a well-known technique that exploits the following two observations:

- Most allocated objects will die young.
- Few references from older to younger objects exist.

These two observations are collectively known as the weak generational hypothesis, which generally holds true for Java applications. To take advantage of this hypothesis, the Java HotSpot VM splits the heap into two physical areas, which are referred to as generations, one young and the other old:

- *Young Generation*. Most newly allocated objects are allocated in the young generation (see Figure 1), which is typically small and collected frequently. Since most objects in it are expected to die quickly, the number of objects that survive a young generation collection (also referred to as a minor collection) is expected to be low. In general, minor collections are very efficient because they concentrate on a space that is usually small and is likely to contain a lot of garbage objects.
- *Old Generation*. Objects that are longer-lived 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 (also referred to as major collections) are infrequent, but when they do occur they are quite lengthy.

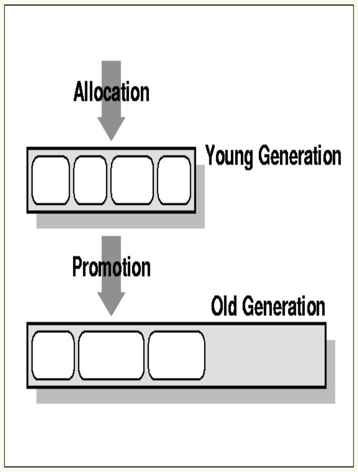

Figure 1. Generational Garbage Collection

To keep minor collections short, the GC must be able to identify live objects in the young generation without having to scan the entire (and potentially larger) old generation. It can achieve this in several ways. In the Java HotSpot VM, the GC uses a data structure called a card table. The old generation is split into 512-byte chunks called cards. The card table is an array with one byte entry per card in the heap. Every update to a reference field of an object also ensures that the card containing the updated reference field is marked *dirty* by setting its entry in the card table to the appropriate value. During a minor collection, only the areas that correspond to dirty cards are scanned to potentially discover old-to-young references (see Figure 2).

In cooperation with the bytecode interpreter and the just-in-time (JIT) compiler, the Java HotSpot VM uses a write barrier to maintain the card table. This barrier is a small fragment of code that sets an entry of the card table to the dirty value. The interpreter executes a write barrier every time it executes a bytecode that updates a reference field. Additionally, the JIT compiler emits the write barrier after emitting the code that updates a reference field. Although write barriers do impact performance on the execution a bit, their presence allows for much faster minor collections, which typically improves the end-to-end throughput of an application.

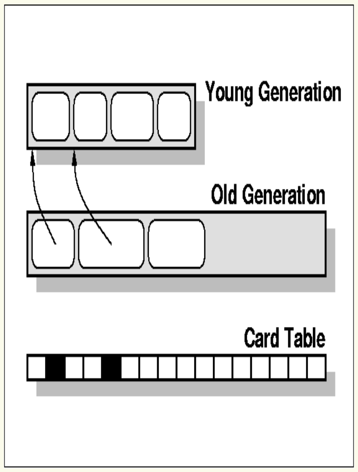

Figure 2. Use of a Card Table

A big advantage of generational garbage collection is that each generation can be managed by the garbage collection algorithm that is most appropriate for its characteristics. A fast GC usually manages the young generation, as minor collections are very frequent. This GC might be a little space wasteful, but since the young generation typically is a small portion of the heap, this is not a big problem. On the other hand, a GC that is space efficient usually manages the old generation, as the old generation takes up most of the heap. This GC might not be quite as fast, but because major collections are infrequent, it doesn't have a big performance impact.

To take full advantage of generational garbage collection, applications should conform to the weak generational hypothesis, as it is what generational garbage collection exploits. For the Java applications that do not do so, a generational garbage collector unfortunately might add more overhead. In practice, such applications are not very common.

The Young Generation

Figure 3 illustrates the layout of the young generation of the Java HotSpot VM (the spaces are not drawn to proportion). It is split into three separate areas:

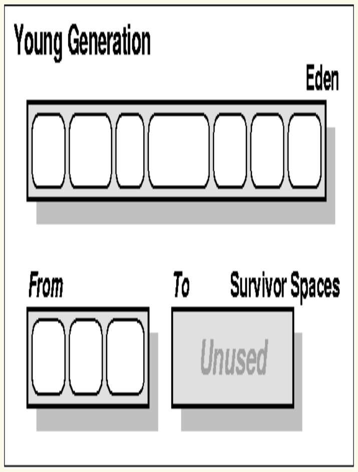

Figure 3. Young Generation Layout

- *The Eden*. This is where most new objects are allocated (not all, as large objects may be allocated directly into the old generation). The Eden is always empty after a minor collection.
- *The Two Survivor Spaces*. These hold objects that have survived at least one minor collection but have been given another chance to die before being promoted to the old generation. As illustrated in Figure 3, only one of them holds objects, while the other is unused.

Figure 4 illustrates the operation of a minor collection (objects that have been found to be garbage are marked with a gray X). Live objects in the Eden that survive the collection are copied to the unused survivor space. Live objects in the survivor space that is in use, which will be given another chance to die in the young generation, are also copied to the unused survivor space. Finally, live objects in the survivor space that is in use, which are deemed "old enough," are promoted to the old generation.

At the end of the minor collection, the two survivor spaces swap roles (see Figure 5). The Eden is entirely empty; only one survivor space is in use; and the occupancy of the old generation has grown slightly. Because live objects are copied during its operation, this type of collector is called a copying collector.

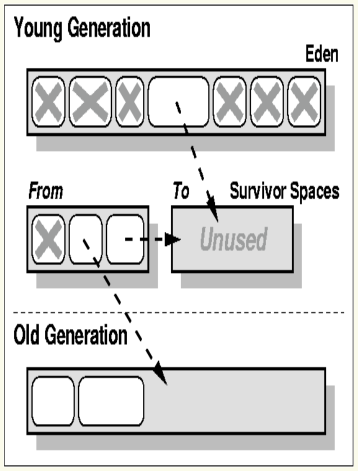

Figure 4. During a Minor Collection

Fast Allocation

The operation of the allocator is tightly coupled with the operation of the garbage collector. The collector has to record where in the heap the free space it reclaims is located. In turn, the allocator needs to discover where the free space in the heap is before it can re-use it to satisfy allocation requests. The copying collector that collects the young generation of the Java HotSpot VM has the advantage of always leaving the Eden empty, which allows allocations into the Eden to be very efficient by using the bump-the-pointer technique. According to this technique, the end of the last allocated object being tracked (usually called top) and when a new allocation request needs to be satisfied, the allocator needs only to check whether it will fit between top and the end of the Eden. If it does, top is bumped to the end of the newly allocated object.

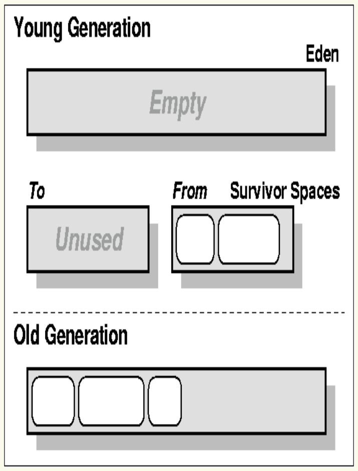

Figure 5. After a Minor Collection

However, most interesting Java applications are multi-threaded and their allocation operations need to be multi-threaded safe. If they simply used global locks to ensure this, then allocation into Eden would become a bottleneck and degrade performance. Instead, the Java HotSpot VM has adopted a technique called Thread-Local Allocation Buffers (TLABs), which improves multi-threaded allocation throughput by giving each thread its own buffer (i.e., a small chunk of the Eden) from which to allocate. Since only one thread can be allocating into each TLAB, allocation can take place quickly with the bump-the-pointer technique (and without any locking). When a thread fills up its TLAB and needs to get a new one (an infrequent operation), however, it needs to do so in a multi-threaded safe way. In the Java HotSpot VM, the new Object() operation is, most of the time, around 10 native instructions. It is the operation of the garbage collector, which empties the Eden, that enables this fast allocation scheme.

Garbage Collectors: Spoiled for Choice

> *"The Java Virtual Machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements."*
> —- Java Virtual Machine Specification, Section 3.5.3 [JVMS2 1999]

In fact, the Java HotSpot VM has three different garbage collectors, each of which is targeted to a different set of applications. The next three sections describe them.

The Serial Collector
The configuration of the Serial Collector is a young generation over an old generation managed by a sliding compacting mark-sweep, also known as a mark-compact, collector. Both minor and major collections take place in a *stop-the-world* fashion (i.e., the application is stopped while a collection is taking place). Only after the collection has finished is the application restarted (see Figure 6).

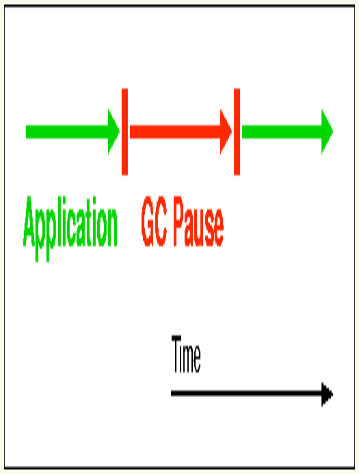

Figure 6. The Serial Collector

The mark-compact collector first identifies which objects are still live in the old generation. It then slides them towards the beginning of the heap, leaving any free space in a single contiguous chunk at the end of the heap. This allows any future allocations into the old generation, which will most likely take place as objects are being promoted from the young generation, to use the fast bump-the-pointer technique. Figure 7 illustrates the operation of such a collector (in sub-figure *a*, objects marked with a gray X are assumed to be garbage, and the shaded area in sub-figure *b* denotes free space).

The Serial Collector is the 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 CPU for garbage collection work (hence, its name). Still, on today's hardware, the Serial Collector can efficiently manage a lot of non-trivial applications with a few 100MBs of heap, with relatively short worst-case pauses (around a couple of seconds for major collections).

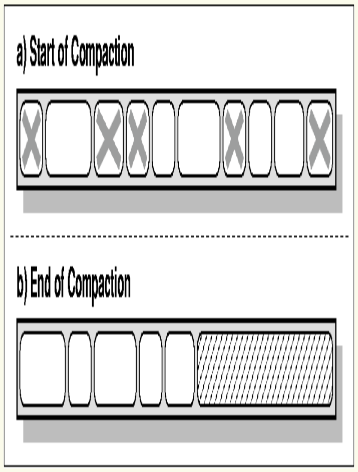

Figure 7. Compaction of the Old Generation

The Parallel Collector: Throughput Matters!

These days, a lot of important Java applications run on (sometimes dedicated) servers with a lot of physical memory and multiple CPUs. Ideally, the garbage collector can take advantage of all available CPUs and not leave most of them idle while only one does garbage collection work.

To decrease garbage collection overhead and hence increase application throughput, on server-style machines, the Java HotSpot VM includes the Parallel Collector, also called the Throughput Collector. Its operation is similar to that of the Serial Collector (i.e., it is a stop-the-world collector with the young generation over a mark-compact old generation). However, the minor collections take place in parallel, using all available CPUs (see Figure 8). The major collections are performed serially, but there are plans to "parallelize" them in the near future.

Applications that can benefit from the Parallel Collector are those that do not have pause time constraints (as infrequent—but potentially long—major collections will still occur), that run on machines with more than one CPU, and that don't have a need for end-to-end throughput. Examples of such applications include batch processing, scientific computing, etc. The Parallel Collector, compared to the Serial Collector, will improve minor collection efficiency and as a result will improve application throughput. Whereas the major collection times will remain largely unchanged (as, in both cases, they are done serially and with the same algorithm).

The Mostly-Concurrent Collector: Latency Matters!

For a number of applications, end-to-end throughput is not as important as fast response time. In the stop-the-world model, when a collection is taking place, the application itself is not running and external requests will not be satisfied until the application is restarted. Minor collections do not typically cause very long pauses. However, major collections, even though infrequent, can impose very long pauses, especially when large heaps are involved.

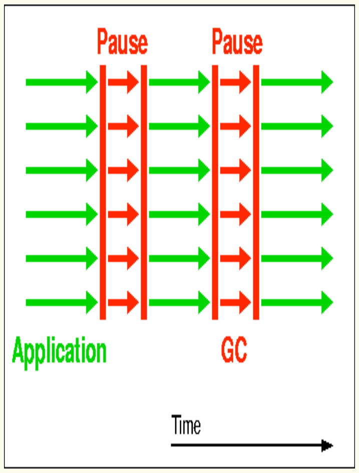

Figure 8. The Parallel Collector: Throughput Matters!

To deal with this, the Java HotSpot VM includes the Mostly-Concurrent Collector, also known as the Concurrent Mark-Sweep (CMS) or the Low Latency Collector. It manages its young generation the same way the Parallel and Serial Collectors do. Its old generation, however, is managed by an algorithm that performs most of its work concurrently, imposing only two short pauses per collection cycle.

Figure 9 illustrates how a collection cycle works in the Mostly-Concurrent Collector. It starts with a short pause, called initial mark, that identifies the set of objects that are immediately reachable from outside the heap. Then, during the concurrent marking phase, it marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields (hence, modifying the object graph) 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 deal with 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. The card table data structure is re-used to also keep track of modified objects. Because the remark pause is more substantial than the initial mark, it is "parallelized" to increase its efficiency. At the end of the remark phase, all live objects in the heap are guaranteed to have been marked. Since revisiting objects during the remark phase increases 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.

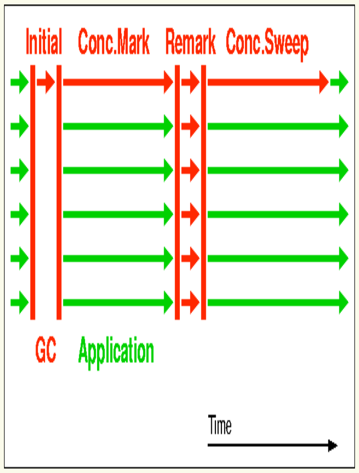

Figure 9. The Mostly-Concurrent Collector: Latency Matters!

Having identified all live objects in the old generation, the final phase of the collection cycle is the concurrent sweeping phase, which sweeps over the heap, de-allocating garbage objects in-place without relocating the live ones. Figure 10 illustrates the operation of the sweeping phase: in sub-figure *a*, objects marked with a gray X are assumed to be garbage, and the shaded areas in sub-figure *b* denote free space. In sub-figure *b*, free space is not contiguous (unlike in the previous two collectors, as illustrated in Figure 7), and the collector needs to employ a data structure (free lists, in this case) that records which parts of the heap contain free space. As a result, allocation into the old generation is more expensive, as allocation from free lists is not as efficient as the bump-the-pointer technique. This imposes extra overhead to minor collections, as most allocations in the old generation take place when objects are promoted during minor collections.

Another disadvantage that the Mostly-Concurrent Collector has, which the previous two don't, is that it typically has larger heap requirements. A few reasons explain why. First, a concurrent marking cycle will last much longer than that of a stop-the-world collector. And it is only during the sweeping phase that space is actually reclaimed. Given that the application is allowed to run during the marking phase, it is also allowed to allocate memory, hence the occupancy of the old generation potentially will grow during the marking phase and drop only during the sweeping phase. Additionally, despite the collector's guarantee to identify all live objects during the marking phase, it doesn't actually guarantee that it will identify all objects that are garbage. Some objects that will become garbage during the marking phase may or may not be reclaimed during the cycle. If they are not, then they will be reclaimed during the next cycle. Garbage objects that are wrongly identified as live are usually referred to as floating garbage.

Finally, fragmentation issues due to the lack of compaction might also prevent the collector from using the available space as efficiently as possible. If the old generation is full before the collection cycle in progress has actually reclaimed sufficient space, the Mostly-Concurrent Collector will revert to an expensive stop-the-world compacting phase, similar to that of the Parallel and Serial Collectors.

Compared to the Parallel Collector, the Mostly-Concurrent 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. Due to its concurrency, it also takes CPU cycles away from the application during a collection cycle. Applications that can benefit from it are ones that require fast response times (such as data-tracking servers, Web servers, etc.), and it is in fact widely used in this context.

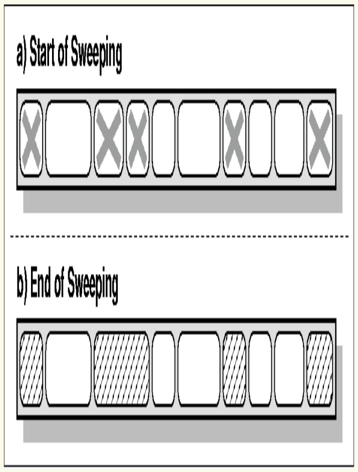

Figure 10. Sweeping of the Old Generation

Further Reading for the Interested Reader

This article presented a brief introduction to how garbage collection works in the Java HotSpot VM. It introduced the notion of generational garbage collection, described the way the young generation works in the Java HotSpot VM, gave an overview of the three different garbage collectors that are available, and indicated for which situation each of them is appropriate.

Find more resources related to this topic at the following Web sites:

- *Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine* (Sun documentation)
- *Garbage Collector Ergonomics* (Sun documentation)
- *Java Platform, Standard Edition, v 1.4.2 API Specification* (Sun documentation).
- *Generational and Concurrent Garbage Collection* (IBM developerWorks)

For an informative introduction to garbage collection, R. E. Jones' *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* covers all the techniques mentioned in this article.

Acknowledgements
The author is grateful to Miriam Kadansky and Peter Kessler for their many helpful comments on this article.

References

- [JVMS2 1999] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification—Second Edition*. Addison-Wesley, 1999.

***Tony Printezis** is a member of the Java Technology Research Group at SunLabs East. He spends most of his time working on dynamic memory management for the Java platform, concentrating mainly on the scalability, responsiveness, parallelism, and visualization of garbage collectors.*