Java theory and practice: A brief history of garbage collection

How does garbage collection work?

Brian Goetz

The Java language may be the most widely used programming language to rely on garbage collection, but it is by no means the first. Garbage collection has been an integral part of many programming languages, including Lisp, Smalltalk, Eiffel, Haskell, ML, Scheme, and Modula-3, and has been in use since the early 1960s. In this installment of Java theory and practice, Brian Goetz describes the most common techniques for garbage collection. Over the next several months, he'll look at the garbage collection strategies employed by the 1.4 JVM, some performance implications of various garbage collection strategies, and how (as well as how not) to assist the garbage collector to yield better performance.

View more content in this series

The benefits of garbage collection are indisputable -- increased reliability, decoupling of memory management from class interface design, and less developer time spent chasing memory management errors. The well-known problems of dangling pointers and memory leaks simply do not occur in Java programs. (Java programs can exhibit a form of memory leak, more accurately called unintentional object retention, but this is a different problem.) However, garbage collection is not without its costs -- among them performance impact, pauses, configuration complexity, and nondeterministic finalization.

An ideal garbage collection implementation would be totally invisible -- there would be no garbage collection pauses, no CPU time would be lost to garbage collection, the garbage collector wouldn't interact negatively with virtual memory or the cache, and the heap wouldn't need to be any larger than the residency (heap occupancy) of the application. Of course, there are no perfect garbage collectors, but garbage collectors have improved significantly over the past ten years.

Options -- and choices

The 1.3 JDK includes three different garbage collection strategies; the 1.4.1 JDK includes six, and over a dozen command-line options for configuring and tuning garbage collection. How do they differ? Why do we need so many?
The various garbage collection implementations use different strategies for identification and reclamation of unreachable objects, and they interact differently with the user program and scheduler. Different sorts of applications will have different requirements for garbage collection -- real-time applications will demand short and bounded-duration collection pauses, whereas enterprise applications may tolerate longer or less predictable pauses in favor of higher throughput.

**How does garbage collection work?**

There are several basic strategies for garbage collection: reference counting, mark-sweep, mark-compact, and copying. In addition, some algorithms can do their job incrementally (the entire heap need not be collected at once, resulting in shorter collection pauses), and some can run while the user program runs (concurrent collectors). Others must perform an entire collection at once while the user program is suspended (so-called stop-the-world collectors). Finally, there are hybrid collectors, such as the generational collector employed by the 1.2 and later JDKs, which use different collection algorithms on different areas of the heap.

When evaluating a garbage collection algorithm, we might consider any or all of the following criteria:

- **Pause time.** Does the collector stop the world to perform collection? For how long? Can pauses be bounded in time?
- **Pause predictability.** Can garbage collection pauses be scheduled at times that are convenient for the user program, rather than for the garbage collector?
- **CPU usage.** What percentage of the total available CPU time is spent in garbage collection?
- **Memory footprint.** Many garbage collection algorithms require dividing the heap into separate memory spaces, some of which may be inaccessible to the user program at certain times. This means that the actual size of the heap may be several times bigger than the maximum heap residency of the user program.
- **Virtual memory interaction.** On systems with limited physical memory, a full garbage collection may fault nonresident pages into memory to examine them during the collection process. Because the cost of a page fault is high, it is desirable that a garbage collector properly manage locality of reference.
- **Cache interaction.** Even on systems where the entire heap can fit into main memory, which is true of virtually all Java applications, garbage collection will often have the effect of flushing data used by the user program out of the cache, imposing a performance cost on the user program.
- **Effects on program locality.** While some believe that the job of the garbage collector is simply to reclaim unreachable memory, others believe that the garbage collector should also attempt to improve the reference locality of the user program. Compacting and copying collectors relocate objects during collection, which has the potential to improve locality.
- **Compiler and runtime impact.** Some garbage collection algorithms require significant cooperation from the compiler or runtime environment, such as updating reference counts whenever a pointer assignment is performed. This creates both work for the compiler, which must generate these bookkeeping instructions, and overhead for the runtime environment,
which must execute these additional instructions. What is the performance impact of these requirements? Does it interfere with compile-time optimizations?

Regardless of the algorithm chosen, trends in hardware and software have made garbage collection far more practical. Empirical studies from the 1970s and 1980s show garbage collection consuming between 25 percent and 40 percent of the runtime in large Lisp programs. While garbage collection may not yet be totally invisible, it sure has come a long way.

**The basic algorithms**

The problem faced by all garbage collection algorithms is the same -- identify blocks of memory that have been dispensed by the allocator, but are unreachable by the user program. What do we mean by unreachable? Memory blocks can be reached in one of two ways -- if the user program holds a reference to that block in a root, or if there is a reference to that block held in another reachable block. In a Java program, a root is a reference to an object held in a static variable or in a local variable on an active stack frame. The set of reachable objects is the transitive closure of the root set under the points-to relation.

**Reference counting**

The most straightforward garbage collection strategy is reference counting. Reference counting is simple, but requires significant assistance from the compiler and imposes overhead on the mutator (the term for the user program, from the perspective of the garbage collector). Each object has an associated reference count -- the number of active references to that object. If an object's reference count is zero, it is garbage (unreachable from the user program) and can be recycled. Every time a pointer reference is modified, such as through an assignment statement, or when a reference goes out of scope, the compiler must generate code to update the referenced object's reference count. If an object's reference count goes to zero, the runtime can reclaim the block immediately (and decrement the reference counts of any blocks that the reclaimed block references), or place it on a queue for deferred collection.

Many ANSI C++ library classes, such as `string`, employ reference counting to provide the appearance of garbage collection. By overloading the assignment operator and exploiting the deterministic finalization provided by C++ scoping, C++ programs can use the `string` class as if it were garbage collected. Reference counting is simple, lends itself well to incremental collection, and the collection process tends to have good locality of reference, but it is rarely used in production garbage collectors for a number of reasons, such as its inability to reclaim unreachable cyclic structures (objects that reference each other directly or indirectly, like a circularly linked list or a tree that contains back-pointers to the parent node).

**Tracing collectors**

None of the standard garbage collectors in the JDK uses reference counting; instead, they all use some form of tracing collector. A tracing collector stops the world (although not necessarily for the entire duration of the collection) and starts tracing objects, starting at the root set and following references until all reachable objects have been examined. Roots can be found in program registers, in local (stack-based) variables in each thread's stack, and in static variables.
Mark-sweep collectors

The most basic form of tracing collector, first proposed by Lisp inventor John McCarthy in 1960, is the mark-sweep collector, in which the world is stopped and the collector visits each live node, starting from the roots, and marks each node it visits. When there are no more references to follow, collection is complete, and then the heap is swept (that is, every object in the heap is examined), and any object not marked is reclaimed as garbage and returned to the free list. Figure 1 illustrates a heap prior to garbage collection; the shaded blocks are garbage because they are unreachable by the user program:

Figure 1. Reachable and unreachable objects

Mark-sweep is simple to implement, can reclaim cyclic structures easily, and doesn't place any burden on the compiler or mutator like reference counting does. But it has deficiencies -- collection pauses can be long, and the entire heap is visited in the sweep phase, which can have very negative performance consequences on virtual memory systems where the heap may be paged.

The big problem with mark-sweep is that every active (that is, allocated) object, whether reachable or not, is visited during the sweep phase. Because a significant percentage of objects are likely to be garbage, this means that the collector is spending considerable effort examining and handling garbage. Mark-sweep collectors also tend to leave the heap fragmented, which can cause locality issues and can also cause allocation failures even when sufficient free memory appears to be available.

Copying collectors

In a copying collector, another form of tracing collector, the heap is divided into two equally sized semi-spaces, one of which contains active data and the other is unused. When the active space fills up, the world is stopped and live objects are copied from the active space into the inactive space. The roles of the spaces are then flipped, with the old inactive space becoming the new active space.

Copying collection has the advantage of only visiting live objects, which means garbage objects will not be examined, nor will they need to be paged into memory or brought into the cache. The duration of collection cycles in a copying collector is driven by the number of live objects. However,
copying collectors have the added cost of copying the data from one space to another, adjusting all references to point to the new copy. In particular, long-lived objects will be copied back and forth on every collection.

**Heap compaction**

Copying collectors have another benefit, which is that the set of live objects are compacted into the bottom of the heap. This not only improves locality of reference of the user program and eliminates heap fragmentation, but also greatly reduces the cost of object allocation -- object allocation becomes a simple pointer addition on the top-of-heap pointer. There is no need to maintain free lists or look-aside lists, or perform best-fit or first-fit algorithms -- allocating \( n \) bytes is as simple as adding \( n \) to the top-of-heap pointer and returning its previous value, as suggested in Listing 1:

**Listing 1. Inexpensive memory allocation in a copying collector**

```c
int *malloc(int n) {
    if (heapTop - heapStart < n)
        doGarbageCollection();
    void *wasStart = heapStart;
    heapStart += n;
    return wasStart;
}
```

Developers who have implemented sophisticated memory management schemes for non-garbage-collected languages may be surprised at how inexpensive allocation is -- a simple pointer addition -- in a copying collector. This may be one of the reasons for the pervasive belief that object allocation is expensive -- earlier JVM implementations did not use copying collectors, and developers are still implicitly assuming allocation cost is similar to other languages, like C, when in fact it may be significantly cheaper in the Java runtime. Not only is the cost of allocation smaller, but for objects that become garbage before the next collection cycle, the deallocation cost is zero, as the garbage object will be neither visited nor copied.

**Mark-compact collectors**

The copying algorithm has excellent performance characteristics, but it has the drawback of requiring twice as much memory as a mark-sweep collector. The *mark-compact* algorithm combines mark-sweep and copying in a way that avoids this problem, at the cost of some increased collection complexity. Like mark-sweep, mark-compact is a two-phase process, where each live object is visited and marked in the marking phase. Then, marked objects are copied such that all the live objects are compacted at the bottom of the heap. If a complete compaction is performed at every collection, the resulting heap is similar to the result of a copying collector -- there is a clear demarcation between the active portion of the heap and the free area, so that allocation costs are comparable to a copying collector. Long-lived objects tend to accumulate at the bottom of the heap, so they are not copied repeatedly as they are in a copying collector.

**Well, which one?**

OK, so which of these approaches does the JDK take for garbage collection? In some sense, all of them. Early JDKs used a single-threaded mark-sweep or mark-sweep-compact collector. JDKs 1.2
and later employ a hybrid approach, called *generational collection*, where the heap is divided into several sections based on an object's age, and different generations are collected separately using different collection algorithms.

Generational garbage collection turns out to be very effective, although it introduces several additional bookkeeping requirements at runtime. In next month's *Java theory and practice*, we'll explore how generational garbage collection works and how it is employed by the 1.4.1 JVM, in addition to all the other garbage collection options offered by the 1.4.1 JVM. In the installment following that, we'll look at the performance impact of garbage collection, including debunking some performance myths related to memory management.
Related topics

- This article from *IBM Systems Journal* describes some of the **lessons learned** in building the IBM 1.1.x Developer Kits for the Java platform, including the details of mark-sweep and mark-sweep-compact garbage collection.
- The Garbage Collection mailing list maintains a [garbage collection FAQ](https://www.ibm.com/developerWorks/developerWorks/garbagecollection/).
- The Boost library, a library of very useful C++ classes, includes the `shared_ptr` smart pointer class that demonstrates the use of reference counting to provide a rudimentary form of garbage collection in C++.