Garbage Collection
(aka Automatic Memory Management)

Douglas Q. Hawkins
http://www.dougqh.net
dougqh@gmail.com

Why?
Unfortunately, the biggest reason why we need to care about garbage collection is because it is the place where the abstract of the virtual machines leak the most. In fact, big data systems built on Java have to side-step the garbage collector. In part, this is because garbage collection techniques are just keeping up with the advances in hardware. The state of the art in garbage collection has to continue to evolve to handle the same things we struggle with multi-core, multi-threaded and also have to deal with the gory details of caches and memory latency and non-uniform memory access.

...but this is not a comprehensive talk about tuning for that I’d refer you to the new book on Java Performance or the Sun Memory Management whitepaper.
HotSpot has bunch of different collectors, so I want you to come away from this talk understand why it has though different collectors. And, to have a basic understanding of theory behind those collectors, so that you know better how to tune them.

This talk will cover a broad range of topics: including fundamental concepts of garbage collection, specifics of allocation and garbage collection in different VMs. And, touch on alternatives to garbage collection.
The fundamentals of a tracing GC are relatively straightforward. Identify the live objects - whatever is not found to be alive must be dead. Remove the dead objects. Optionally, merge chunks of free space - possibly by compacting the live objects to be closer together.
One of the easiest to understand implementations is also one of the oldest: mark and sweep. This algorithm dates back 50 years to the earliest implementations of Lisp. Remarkably, variations of this algorithm remain useful today.

Mark and sweep is a straightforward implementation of the basic tracing idea. First, we start from a set of roots which include any active stack frames, static field references, among other things. From these roots, we identify objects that are used in the heap. After identifying the first level of objects, we follow references in those objects to the next set of objects, and so on.

Whatever objects that we don't visit are garbage and can be swept away.

**NOTE:** That the two objects in the lower right refer to each other, but are not reachable through the roots and can still be collected. (This is the main advantage of tracing collectors over reference counting.)

The problem with mark & sweep is that it leaves wholes scattered throughout the heap. As we'll see in the next slide, this has some interesting implications for allocation.

To get rid of the holes, we can use a variation on mark & sweep called mark compact. Mark compact is not just a sweep followed by a compact. The sweep and the compact steps are useful performed as a single step.

The most basic way to perform compact is two finger compaction. In two finger compaction, one pointer starts from the left and another from the right. The pointer on the left identifies dead slots, the pointer on the right identifies live objects to relocate. The problem with this approach is that old and new objects end up mixed somewhat randomly in the heap. This hurts locality resulting in more cache misses.
How we garbage collect objects (deallocated) actually has a strong influence on how we allocate objects.

If we've compacted our heap, then we can use a very simple allocation technique. We track just track a single pointer. If there is enough space between the pointer and the end of the heap, we have room to allocate. This technique is $O(1)$ time and $O(1)$ space. Because the technique involves incrementing a single pointer, it is called "bump the pointer" allocation.

If we have holes in the heap, then we have to maintain a free list. When we need to allocate, we traverse the list looking for a big enough whole in which to place our object. This technique is $O(n)$ time and $O(n)$ space.

The simple free list mentioned above can lead to a lot of small blocks at the beginning of the free list that cannot contain the object being allocated. To combat this problem, we can use a segregated fits approach. Here we keep separate free lists for different block sizes. This approach is still $O(n)$ time and $O(n)$ space, but has better average case.

A slightly less obvious collection strategy is to use a copy collector. A copy collector works by making copies of the live objects in separate area of the heap.

This is done by starting from the roots and copying the live objects over to a clean space. When complete the old space is completely empty and can be reclaimed en masse.

Copying objects may seem wasteful particularly for large objects -- and it is, but as we'll see if most of the objects are not live that ends up being a non-issue.

This process of moving survivors to another space is called evacuation. And, because, this approach involves two separate spaces this is sometimes called a semi-space collector. This type of collector is also sometimes called a scavange collector.

Then the objects they reference and so. As shown here, this is done breadth first. However, a breadth first approach poses problems for
A slightly less obvious collection strategy is to use a copy collector. A copy collector works by making copies of the live objects in separate area of the heap.

This is done by starting from the roots and copying the live objects over to a clean space. When complete the old space is completely empty and can be reclaimed enmass.

Copying objects may seem wasteful particularly for large objects -- and it is, but as we'll see if most of the objects are not live that ends up being a non-issue.

This process of moving survivors to another space is called evacuation. And, because, this approach involves two separate spaces this is sometimes called a semi-space collector. This type of collector is also sometimes called a scavenge collector.

Then the objects they reference and so. As shown here, this is done breadth first. However, a breadth first approach poses problems for
The problem with all the strategies discussed so far is that they require a random walk of the heap to identify live objects. Unfortunately, a random walk of the heap is extremely cache hostile. We could improve performance if we only walked part of the heap.

Fortunately, a simple and nearly universal observation allows us to do just that -- that observation - most objects either: die young or live forever. The chart shown here comes from a stereo-typical Java program. You can see a strong peak towards the short-lived portion of the spectrum.

http://www.devx.com/Java/Article/21977
Remembered Set

Stack Frames  Young  Old

Static Variables
System.in  System.out

...but how does separating young and old help us scan the heap less often?
A "remembered set" is used to keep track of references between the generations. This remembered set is kept up to date through the use of a write barrier. Whenever, a reference is modified if the object being updated is in the old generation, the remembered set is updated.

During most collections just the young generation is "condemned", at this time, the remembered set effectively acts as an extra set of roots for the young generation.

Whether we need a remembered set for the old generation depends on whether the old generation can be collected independently from the young generation. If it can be collected independently, we'd need another remembered set to track references from young to old. In general for n-regions, we need n*(n-1) remembered sets - i.e. O(n^2).

As an benefit, we can also use different collection strategies for the two distinct generations / regions.
Now, that you know the core concepts let's talk about memory management in HotSpot. As you'll see, HotSpot uses a generational heap with a slight variation on the copy/semi-space collector for the young generation, and a couple different options on how to collect the old generation.

http://download.oracle.com/javase/1.5.0/docs/guide/vm/gc-ergonomics.html

Different heap size defaults in Java 6 - lesser of 1/4 of physical or 1GB
Adaptive sizing - now the default with -XX:+UseParallelGC (starting with 1.5?)
Looking at the graph from earlier, minor collections handle the large peak at the beginning and the small bump that follows. The major collections cleaned up what remains thereafter. The small peak at the far right is immortals objects which even the major collections will eventually ignore because of the "dense prefix" optimization.

http://java.sun.com/docs/hotspot/gc/
http://www.devx.com/Java/Article/21977
http://java.sun.com/j2se/reference/whitepapers/
memorymanagement_whitepaper.pdf
First, we'll look at object allocation in HotSpot... HotSpot calls the allocation area “Eden” and uses a "pump the pointer" approach to allocate; however, it has an optimization to improve multi-threaded performance. If HotSpot used a single pointer, that pointer would always need to be synchronized between threads. Instead each thread requests a thread local allocation buffer (TLAB), objects for each thread are allocated within the TLAB using a "bump pointer technique" with a thread local pointer.

As of Java 5, TLABs are dynamically sized by HotSpot.

http://blogs.sun.com/jonthecollector/entry/the_real_thing
http://blogs.sun.com/jonthecollector/entry/a_little_thread_privacy>Please read the rest of your text.
For the old generation, we can choose to use either mark compaction or mark and sweep. As we'll see when we talk about parallelism and concurrency, each of these approaches have their pros and cons.

http://www.devx.com/Java/Article/21977/0

HotSpot has one more generation -- the permanent generation where it allocates classes. In newer versions of HotSpot, the permanent generation isn't really permanent, but it is still close. If even one object remains alive from a class, then all classes from that ClassLoader will stay in the permanent generation.

http://opensource.atlassian.com/confluence/spring/pages/viewpage.action?pageId=2669
http://dow.ngra.de/2009/06/15/classloaderlocal-how-to-avoid-classloader-leaks-on-application-redeploy/
Newer JVMs can adaptively tune, but you may not want to rely on that.
Adaptive Tuning

Newer JVMs can automatically tune, but you may not want to rely on that.

-XX:MaxGCPauseMillis
-XX:GCTimeRatio

Meets these goals by adjusting the heap size.

Basic Principles

Maximize Collection during Minor GCs
Avoid Major GCs at much as possible
Promote as little as possible

Maximize Available Memory

Adjust for 2 of 3 GC Goals
Throughput
Latency
Footprint
Set Xms = Xmx
initial heap size max heap size

Stops region resizing
which would require a full GC

3-4x

Young (Xmn): 1-1.5x
Old: 2-3x

Set XX:PermSize = XX:MaxPermSize

Again, stops region resizing
which would require a full GC

1.2-1.5x
Earlier HotSpot VMs, used a serial garbage collection for both the young and old generation. Newer VMs can use multiple threads simultaneously to achieve parallel collection; however, that does not mean that they do not stop the world.

So, why do we have to "stop the world"? The problem is the need to move/relocate objects. Unlike what you might expect, the problem is not the need to copy when relocating; it's the need to update all references. Unfortunately, those references from roots and other objects need to all be updated at once which requires stopping the application temporarily. If we don't relocate, we can around this problem.

This is where concurrent mark & sweep (CMS) comes in. By not relocating objects, concurrent mark and sweep can be concurrent - i.e. not stop the world. To do this, it tracks object references as your application runs. This phase is called "concurrent marking".

Before collecting garbage, it has to make sure it's marking is completely up-to-date, so stops the world briefly to perform a "re-mark" phase. The length of this pause will be proportional to the number of reference mutations being performed by your application.

With re-mark complete, HotSpot now knows which objects are no longer being used and can collect while your application continues to run.

http://www.devx.com/Java/Article/21977/0/page/3
So, what are your options for garbage collection in HotSpot. First, you can use serial collection for both young old, but there's little reason to do so.

You can also use parallel just for the young generation, but again there's little reason to.

You can used concurrent mark & sweep for the old generation which uses parallel new to help with some of the book keeping. Unfortunately, when it needs to compact, CMS falls back on serial old.

The default is to use parallel young and parallel old (at least until Java 7).

http://www.petefreitag.com/articles/gctuning/
http://blogs.sun.com/jonthecollector/entry/our_collectors
http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html

Your choice in collector mostly comes down to whether throughput or latency is more important.

If you are running a batch processing system, you might prefer throughput. You'll have longer pauses, but overall the job will get done faster. In this case, you want to use the default parallel young and parallel old.

If you are running a web site, then long pauses may be unacceptable. In that case, you'd want to use CMS, but if you have to be careful that your application does not fall back to serial compact which could result in a really long pause.

If Latency is Too High, Switch to ConcMarkSweep

CMS kicks in when old generation is 65% full.

Increase Old Generation an additional 25-30%

When compaction is needed it is serial, so avoid compaction at all costs!

GCs in Other JVMs
Now, that we’ve talked about HotSpot, let’s look at how other JVMs differ. IBM’s J9 can use either continuous heap or generational heap. When using generational mode, J9 uses a copying semi-space collector for the young generation. Alternating the roles of the Allocate and Survivor regions between collections. The tenured distribution is collected through incremental mark compaction. Unlike HotSpot, J9 does not have a permanent generation; however, it does assume that classes will typically live an extend period of time -- so it allocates directly in the Tenured region.

http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/memman.html

Like J9, JRockit can also use a non-generational heap; however, even when operating in generational mode JRockit uses an extreme simple tenuring policy. In JRockit, needs objects are allocated in a "Keep Area". "Keep Area" objects are not considered (at all) for collection during GC. Instead objects in the "Keep Area" are evacuated into the "Survivor" region. During GC objects in the "Survivor" region that are still alive are promoted to the "Tenured" region. In effect, this means that JRockit always has tenuring promotion value of 2. Like J9, JRockit uses incremental mark compact to collect the Tenured region. However, unlike J9, JRockit allocates classes in the same fashion as regular objects.

http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/memman.html
To achieve incremental mark compact, J9 and JRockit use a hybrid mark sweep and copy approach. Like concurrent mark & sweep, the mark & sweep approach will cause fragmentation in the heap. To remove the fragmentation, incremental compaction is achieved through copying. Here a "To" region in the heap is identified that proceeds a "From" region. Live objects in the "From" region are then evacuated into the "To" region. In this way, sliding compaction can be distributed over several garbage collection cycles.
To combat the limitations of the concurrent HotSpot collectors, Java 7 (and 6) include G1 - the Garbage First collector. Not to be outdone, J9 includes a very similar collector dubbed "Balanced GC".

These collectors work by dividing the heap into a large number of uniformly sized regions. Unfortunately, of the $O(n^2)$ cost in tracking references between regions there is a limited on how many regions can be used. For this reason, Balanced GC aims to have between 1024 and 2047 regions regardless of the heap size.


Let's look at object allocation and collection in G1 and Balanced GC...

Unlike the generational collectors, we've seen previously. Regions in G1 are not always for allocation, young objects, or old objects. How each reason is used will change over the life of the application. Objects will be allocated in a previously empty region. To better deal with NUMA, Balanced GC biases regions to different processors. Like CMS, the utilization of each region is tracked concurrently.

When performing a minor GC, G1 picks a subset of young regions to collect. These are picked according to the amount of garbage present. If region turns out to be all garbage then the whole region can be reclaimed immediately -- thus "garbage first". Otherwise, the region is condemned and objects are evacuated to a previously unused region.

For major collection, G1 picks some a sub-set of young and old regions. Here again objects are evacuated to previous empty region. Because of the evacuation to other regions, G1 and Balanced GC still have to stop the world, but focusing on regions that are mostly garbage, they aim to make the evacuation pauses as short as possible.
Azul C4
Continuously Concurrent Compacting Collector

There is one more server class JVM, that we have not discussed -- Azul. Azul was formed by former Sun staff that worked on HotSpot. They set out to design JVM with special to features to better handle large server applications. Early versions of Azul, only ran on custom hardware, but now they can run on (mostly) stock Linux.

Azul has a truly pauseless GC -- i.e. fully concurrent. To do this, they uses incremental compaction similar to J9 and JRockit, but with an added twist. A read barrier is used to detect when an out of date reference is being followed. If that happens, a mini GC happens to complete the relocation of that object, so that the application thread may continue on its way.

http://www.azulsystems.com/products/zing/c4-java-garbage-collector-wp

Despite all the advances in GC, sometimes we need to use other options to get around GC limitations.
Sometimes, there’s a need to cache an extra piece of information as long as some related object is alive.
To do that, we can use a weak reference.
A WeakReference works by pointing to an object as long as at least one hard reference to the same object continues to exist. After that, the GC can collect the object and the WeakReference will return null.

Similar to a WeakReference, a SoftReference may also be cleaned up by the GC. However, this time no hard references are required. The SoftReference acts as a hint to the GC that is okay to clean up the reference if necessary, but that you’d prefer to keep it around.
Be Careful With References

Reference<User> ref = ...  
if ( ref.get() != null ) {  
    System.out.println(ref.get().getName());  
}

Possible NullPointerException

Reference<User> ref = ...  
User user = ref.get();  
if ( user != null ) {  
    System.out.println(user.getName());  
}

Guava MapMaker & CacheBuilder

ConcurrentMap<Key, Graph> graphs = new MapMaker()  
    .concurrencyLevel(4)  
    .weakKeys()  
    .maximumSize(10000)  
    .expireAfterWrite(10, TimeUnit.MINUTES)  
    .makeComputingMap(  
        new Function<Key, Graph>() {  
            public Graph apply(Key key) {  
                return createExpensiveGraph(key);  
            }  
        });

Because the GC is free to remove the underlying object whenever, all code using Reference objects is inherently multi-thread. This means you need to be careful not to call Reference.get() twice in the same method. In the example above, the second call to Reference.get() could return null causing the code to raise a NullPointerException if the GC kicked after the if.

The main reason to use Reference objects is for caches. Because Reference classes tend to be a bit tricky to use, I'd recommend using Guava’s MapMaker or CacheBuilder when working with references. (NOTE: Unlike Java’s built-in WeakHashMap, MapMaker uses with weakKeys() uses referential equals not Object.equals.)

http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/MapMaker.html  
There is one more reference type that serves as a replacement for finalize.
Using a PhantomReference with a ReferenceQueue, you can achieve finalization without having to worry about the finalize’s shortcomings...
- multi-threaded
- no order guarantee
- object resurrection

If you are dealing with truly gigantic amounts of memory in Java, GC can become a real problem. To get around this, you can use unmanaged memory in the form of direct ByteBuffers. When using ByteBuffer, you typically allocate a slab of memory in a buffer and then serialize objects to store them in the buffer.

This means there is some overhead to serialize / deserialize an object when moving to/from normal heap.

Cassandra, Apache HBase, and Terracotta all use direct ByteBuffers to avoid issues with Full GC.
Additional Reading

Java Performance
By Charlie Hunt and Biju John

Everything I Ever Learned About JVM Performance Tuning
By Attila Szegedi

Memory Management in the HotSpot Java Virtual Machine

The Garbage Collection Handbook
By Richard Jones, Antony Hoskin, Eliot Moss

Q&A