**MODULE 6**

**Java Memory Management**

Java Memory Management, with its built-in garbage collection, is one of the language’s finest achievements. It allows developers to create new objects without worrying explicitly about memory allocation and deallocation, because the garbage collector automatically reclaims memory for reuse. This enables faster development with less boilerplate code, while eliminating memory leaks and other memory-related problems. At least in theory.

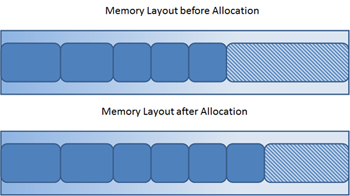
Ironically, Java garbage collection seems to work too well, creating and removing too many objects. Most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system. In order to wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

**How Garbage Collection Really Works**

Many people think garbage collection collects and discards dead objects. In reality, Java garbage collection is doing the opposite! Live objects are tracked and everything else designated garbage. As you’ll see, this fundamental misunderstanding can lead to many performance problems.

Let's start with the heap, which is the area of memory used for dynamic allocation. In most configurations the operating system allocates the heap in advance to be managed by the JVM while the program is running. This has a couple of important ramifications:

* Object creation is faster because global synchronization with the operating system is not needed for every single object. An allocation simply claims some portion of a memory array and moves the offset pointer forward (see Figure 6.1). The next allocation starts at this offset and claims the next portion of the array.
* When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system.



**Figure 6.1**: New objects are simply allocated at the end of the used heap.

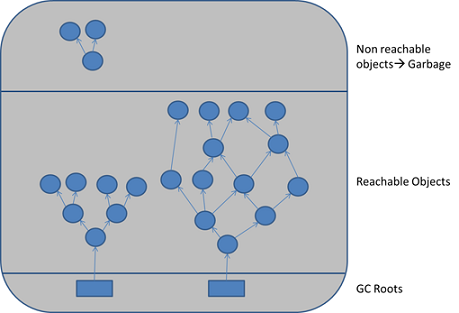
All objects are allocated on the heap area managed by the JVM. Every item that the developer uses is treated this way, including class objects, static variables, and even the code itself. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory. As simple as this sounds, it raises a question: what is the first reference in the tree?

**Garbage-Collection Roots — The Source of All Object Trees**

Every object tree must have one or more root objects. As long as the application can reach those roots, the whole tree is reachable. But when are those root objects considered reachable? Special objects called garbage-collection roots (GC roots; see Figure 6.2) are always reachable and so is any object that has a garbage-collection root at its own root.

There are four kinds of GC roots in Java:

1. **Local variables** are kept alive by the stack of a thread. This is not a real object virtual reference and thus is not visible. For all intents and purposes, local variables are GC roots.
2. **Active Java threads** are always considered live objects and are therefore GC roots. This is especially important for thread local variables.
3. **Static variables** are referenced by their classes. This fact makes them de facto GC roots. Classes themselves can be garbage-collected, which would remove all referenced static variables. This is of special importance when we use application servers, OSGi containers or class loaders in general. We will discuss the related problems in the Problem Patterns section.
4. **JNI References** are Java objects that the native code has created as part of a JNI call. Objects thus created are treated specially because the JVM does not know if it is being referenced by the native code or not. Such objects represent a very special form of GC root, which we will examine in more detail in the Problem Patterns section below.



**Figure 6.2**: GC roots are objects that are themselves referenced by the JVM and thus keep every other object from being garbage-collected.

Therefore, a simple Java application has the following GC roots:

* Local variables in the main method
* The main thread
* Static variables of the main class

**Marking and Sweeping Away Garbage**

To determine which objects are no longer in use, the JVM intermittently runs what is very aptly called a mark-and-sweep algorithm. As you might intuit, it’s a straightforward, two-step process:

1. The algorithm traverses all object references, starting with the GC roots, and marks every object found as alive.
2. All of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept free of unused objects.

Garbage collection is intended to remove the cause for classic memory leaks: unreachable-but-not-deleted objects in memory. However, this works only for memory leaks in the original sense. It’s possible to have unused objects that are still reachable by an application because the developer simply forgot to dereference them. Such objects cannot be garbage-collected. Even worse, such a logical memory leak cannot be detected by any software (see Figure 6.3). Even the best analysis software can only highlight suspicious objects. We will examine memory leak analysis in the Analyzing the Performance Impact of Memory Utilization and Garbage Collection section, below.

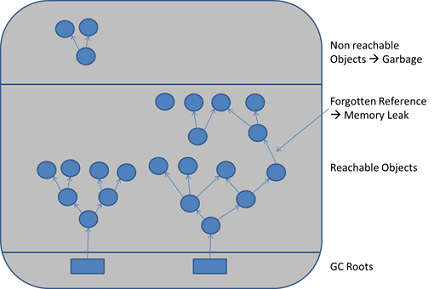


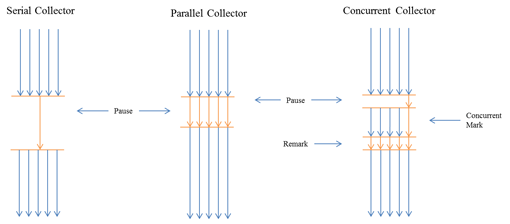
Figure 6.3: When objects are no longer referenced directly or indirectly by a GC root, they will be removed. There are no classic memory leaks. Analysis cannot really identify memory leaks; it can only point out suspicious objects.

**Reducing Garbage-Collection Pause Time**

There are two general ways to reduce garbage-collection pause time and the impact it has on application performance:

* The garbage collection can itself can leverage the existence of multiple CPUs and be executed in parallel. Although the application threads remain fully suspended during this time, the garbage collection can be done in a fraction of the time, effectively reducing the suspension time.
* The second approach is leave the application running, and execute garbage collection concurrently with the application execution.

These two logical solutions have led to the development of serial, parallel, and concurrent garbage-collection strategies, which represent the foundation of all existing Java garbage-collection implementations (see Figure 6.4).



**Figure 6.4**: The differences between garbage-collection algorithms becomes clearest when comparing garbage-collection suspensions.

The serial collector suspends the application and executes the mark-and-sweep algorithm in a single thread. It is the simplest and oldest form of garbage collection in Java and is still the default in the Oracle HotSpot JVM.

The parallel collector uses multiple threads to do its work. It can therefore decrease the GC pause time by leveraging multiple CPUs. It is often the best choice for throughput applications.

The concurrent collector does the majority of its work concurrent with the application execution. It has to suspend the application for only very short amounts of time. This has a big benefit for response-time–sensitive applications, but is not without drawbacks.

**(Mostly) Concurrent Marking and Sweeping**

CConcurrent garbage-collection strategies complicate the relatively simple mark-and-sweep algorithm a bit. The mark phase is usually sub-divided into some variant of the following:

1. In the initial marking, the GC root objects are marked as alive. During this phase, all threads of the application are suspended.
2. During concurrent marking, the marked root objects are traversed and all reachable objects are marked. This phase is fully concurrent with application execution, so all application threads are active and can even allocate new objects. For this reason there might be another phase that marks objects that have been allocated during the concurrent marking. This is sometimes referred to as pre-cleaning and is still done concurrent to the application execution.
3. In the final marking, all threads are suspended and all remaining newly allocated objects are marked as alive. This is indicated in Figure 6.6 by the re-mark label.

The concurrent mark works mostly, but not completely, without pausing the application. The tradeoff is a more complex algorithm and an additional phase that is not necessary in a normal stop-the-world GC: the final marking.

The Oracle JRockit JVM improves this algorithm with the help of a keep area, which, if you’re interested, is described in detail in the JRockit documentation. New objects are kept separately and not considered garbage during the first GC. This eliminates the need for a final marking or re-mark.

In the sweep phase of the CMS, all memory areas not occupied by marked objects are found and added to the free list. In other words, the objects are swept by the GC. This phase can run at least partially concurrent to the application. For instance, JRockit divides the heap into two areas of equal size and sweeps one then the other. During this phase, no threads are stopped, but allocations take place only in the area that is not actively being swept.

The downsides of the CMS algorithm can be quickly identified:

1. As the marking phase is concurrent to the application’s execution, the space allocated for objects can surpass the capacity of the CMS, leading to an allocation error.
2. The free lists immediately lead to memory fragmentation and all this entails.
3. The algorithm is more complicated than the other two and consequently requires more CPU cycles.
4. The algorithm requires more fine tuning and has more configuration options than the other approaches.

These disadvantages aside, the CMS will nearly always lead to greater predictability and better application response time.

**Reducing the Impact of Compacting**

Modern garbage collectors execute their compacting processes in parallel, leveraging multiple CPUs. Nevertheless, nearly all of them have to suspend the application during this process. JVMs with several gigabytes of memory can be suspended for several seconds or more. To work around this, the various JVMs each implements a set of parameters that can be used to compact memory in smaller, incremental steps instead of as a single big block. The parameters are as follows:

* Compacting is executed not for every GC cycle, but only once a certain level of fragmentation is reached (e.g., if more than 50% of the free memory is not continuous).
* One can configure a target fragmentation. Instead of compacting everything, the garbage collector compacts only until a designated percentage of the free memory is available as a continuous block.

This works, but the optimization process is tedious, involves a lot of testing, and needs to be done again and again for every application to achieve optimum results.

**Making Garbage Collection faster**

Short of avoiding garbage collection altogether, there is only one way to make garbage collection faster: ensure that as few objects as possible are reachable during the garbage collection. The fewer objects that are alive, the less there is to be marked. This is the rationale behind the generational heap.

**The Generation Conflict – Young vs. Old**

In a typical application most objects are very short-lived. On the other hand, some objects last for a very long time and even until the application is terminated. When using generational garbage collection, the heap area is divided into two areas—a young generation and an old generation—that are garbage-collected via separate strategies.

Objects are ussually created in the young area. Once an object has survived a couple of GC cycles it is tenured to the old generation. (The JRockit and IBM JVM make exceptions for very large objects, as I will explain later.) After the application has completed its initial startup phase (most applications allocate caches, pools, and other permanent objects during startup), most allocated objects will not survive their first or second GC cycle. The number of live objects that need to be considered in each cycle should be stable and relatively small.

Allocations in the old generation should be infrequent, and in an ideal world would not happen at all after the initial startup phase. If the old generation is not growing and therefore not running out of space, it requires no garbage-collection at all. There will be unreachable objects in the old generation, but as long as the memory is not needed, there is no reason to reclaim it.

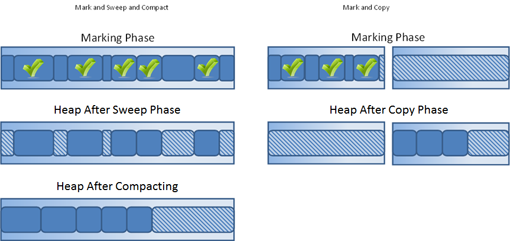
To make this generational approach work, the young generation must be big enough to ensure that all temporary objects die there. Since the number of temporary objects in most applications depends on the current application load, the optimal young generation size is load-related. Therefore, sizing the young generation, known as generation-sizing, is the key to achieving peak load.

Unfortunately, it is often not possible to reach an optimal state where all objects die in the young generation, and so the old generation will often often require a concurrent garbage collector. Concurrent garbage collection together with a minimally growing old generation ensures that the unavoidable, stop-the-world events will at least be very short and predictable.

On the other hand, while there is a very high number of allocations in the young generation at the beginning of each GC cycle, there is only a small portion of objects alive after each GC cycle. This leads to a high level of fragmentation using the GC strategies we have discussed so far. You might think that using free lists would be a good option, but this will slow down allocations. An alternative strategy of executing a full compaction every time has a negative effect on pause time. Instead, most JVMs implement a strategy in the young generation, known as copy collection.

**When Copying is Faster than Marking**

Copy garbage collection divides the heap into two (or more) areas, only one of which is used for allocations. When this area is full, all live objects are copied to the second area, and then the first area is simply declared empty (see Figure 6.5).



**Figure 6.5**: Instead of sweeping away garbage and compacting the heap, the copy collection simply copies live objects someplace else and declares the old region empty.

The advantage here is that no fragmentation occurs, and thus there is no need for free lists and compaction. Allocations are always fast and the GC algorithm is simple. This strategy is effective only if most objects die, which is the default in the young generation. However, this can lead to a problem when the JVM is executing a high number of concurrent transactions.

If the young generation is too small, objects are tenured prematurely to the old generation. If the young generation is too large, too many objects are alive (undead) and the GC cycle will take too long. Contrary to what most people think, these young-generation GCs, often termed minor GCs, are full of stop-the-world events. Their negative impact on response time can be more severe than the occasional old-generation GC.

Not surprisingly, the generational heap does not provide a silver-bullet solution to the garbage-collection problem. The optimal configuration is often a compromise, with an adequately sized young generation to avoid overly long minor GCs, and a concurrent GC in the old generation to deal with prematurely tenured objects.

**The Case for a Non-generational Heap**

The Oracle HotSpot JVM uses a generational heap exclusively, while Oracle JRockit also supports a non-generational heap, and IBM WebSphere defaults to a non-generational heap and recommends that JVMs smaller than 100 MB always use a non-generational heap. In terms of both CPU and memory, a generational GC and the associated copy collection have a certain overhead, which makes sense.

If your application is designed for pure throughput and the number of temporary objects is relatively small, a non-generational GC has advantages. A full parallel GC has a better tradeoff in terms of CPU usage if you don't care about the response time of a single transaction. On the other hand, if the number of temporary objects is relative small, a concurrent GC in a non-generational heap might do the job and with less suspension time than a generational GC. Only a thorough performance test can determine this for sure.

**Improving Allocation Performance**

Two factors have a negative impact on allocation speed: fragmentation and concurrency. We have already dealt with fragmentation. Concurrency’s problem is that all threads in the JVM share memory resources, and all memory allocations must be synchronized. When many threads try to allocate simultaneously, this quickly becomes a choke point. The solution to this is [thread-local allocation](http://download.oracle.com/docs/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/memman.html%22%20%5Ct%20%22_blank).

Each thread receives a small, but exclusive, memory allotment where objects are allocated without the need for synchronization. This increases concurrency and the speed of application execution. (It’s important that this not be confused with the heap area for thread-local variables, where allocated objects are accessible by all threads.)

A single thread-local heap (TLH) to accommodate a large number of threads can still be quite small. The TLH is not treated as a special heap area, but instead is usually a part of the young generation, which creates its own problems.

A generational heap with TLH requires a larger young generation than without a TLH. The same number of objects simply uses more space. On the other hand, a non-generational heap with active TLH is likely to become more fragmented and will need more-frequent compaction.

# Analyzing the Performance Impact of Memory Utilization and Garbage Collection

The goal of any Java memory analysis is to optimize garbage collection (GC) in such a way that its impact on application response time or CPU usage is minimized. It is equally important to ensure the stability of the application. Memory shortages and leaks often lead to instability. To identify memory-caused instability or excessive garbage collection we first need to monitor our application with appropriate tools. If garbage collection has a negative impact on response time, our goal must be to optimize the configuration. The goal of every configuration change must be to decrease that impact. Finally, if configuration changes alone are not enough we must analyze the allocation patterns and memory usage itself. So let's get to it.

## Memory-Monitoring Tools

Since Java 5, the standard JDK monitoring tool has been [JConsole](http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html). The Oracle JDK also includes [jStat](http://docs.oracle.com/javase/1.5.0/docs/tooldocs/share/jstat.html), which enables the monitoring of memory usage and garbage-collector activity from the console, and [Java VisualVM](http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html) (or jvisualvm), which provides rudimentary memory analyses and a profiler. The Oracle JRockit JDK includes [JRockit Mission Control](http://www.oracle.com/technetwork/middleware/jrockit/downloads/index.html%20%22%20%5Ct%20%22_blank) and the [verbose:gc](http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html%22%20%5Ct%20%22_blank) flag of the JVM. Each JVM vendor includes its own monitoring tools, and there are numerous commercial tools available that offer additional functionality.

## Monitoring of Memory Use and GC Activity

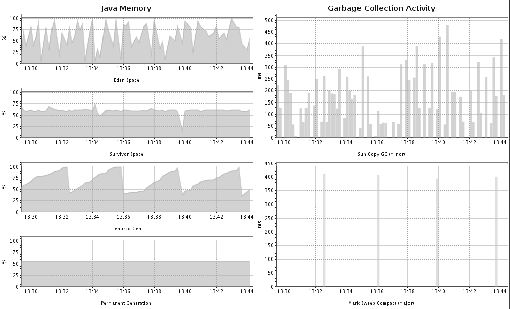
Memory shortage is often the cause of instability and unresponsiveness in a Java applications. Consequently, we need to monitor the impact of garbage collection on response time and memory usage to ensure both stability and performance. However, monitoring memory utilization and garbage collection times is not enough, as these two elements alone do not tell us if the application

response time is affected by garbage collection. Only GC suspensions affect response time directly, and a GC can also run concurrent to the application. We therefore need to correlate the suspensions caused by garbage collection with the application’s response time. Based on this we need to monitor the following:

* Utilization of the different memory pools (Eden, survivor, and old). Memory shortage is the number-one reason for increased GC activity.
* If overall memory utilization is increasing continuously despite garbage collection, there is a memory leak, which will inevitably lead to an out-of-memory error. In this case, a memory heap analysis is necessary.
* The number of young-generation collections provides information on the churn rate (the rate of object allocations). The higher the number, the more objects are allocated. A high number of young collections can be the cause of a response-time problem and of a growing old generation (because the young generation cannot cope with the quantity of objects anymore).
* If the utilization of the old generation fluctuates greatly without rising after GC, then objects are being copied unnecessarily from the young generation to the old generation. There are three possible reasons for this: the young generation is too small, there’s a high churn rate, or there’s too much transactional memory usage.
* High GC activity generally has a negative effect on CPU usage. However, only suspensions (aka stop-the-world events) have a direct impact on response time. Contrary to popular opinion, suspensions are not limited to major GCs. It is therefore important to monitor suspensions in correlation to the application response time.

The JVM memory dashboard (Figure 6.6) shows that the tenured (or old) generation is growing continuously, only to drop back to its old level after an old-generation GC (lower right). This means that there is no memory leak and the cause of the growth is prematurely-tenured objects. The young generation is too small to cope with the allocations of the running transactions. This is also indicated by the high number of young-generation GCs (Oracle/Sun Copy GC). These so-called minor GCs are often ignored and thought to have no impact.

The JVM will be suspended for the duration of the minor garbage collection; it's a stop-the-world event. Minor GCs are usually quite fast, which is why they are called minor, but in this case they have a high impact on response time. The root cause is the same as already mentioned: the young generation is too small to cope. It is important to note that it might not be enough to increase the young generation’s size. A bigger young generation can accommodate more live objects, which in turn will lead to longer GC cycles. The best optimization is always to reduce the number of allocations and the overall memory requirement.

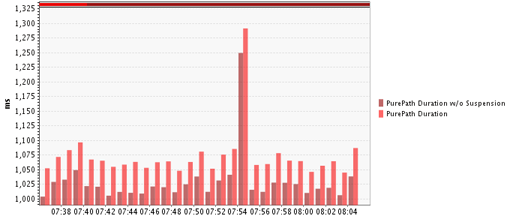


**Figure 6.6**: This shows how one can visualize JVM memory activity

Of course, we cannot avoid GC cycles, and we would not want to. However, we can optimize the configuration to minimize the impact of GC suspensions on response time.

## How to Monitor and Interpret the Impact of GC on Response Time

GC suspensions represent the only direct impact on response time by the garbage collector. The only way to monitor this is via the JVM tool interface (JVM TI), which can be used to register callbacks at the start and end of a suspension. During a stop-the-world event, all active transactions are suspended. We can correlate these suspensions to application transactions by identifying both the start and end of the suspension. Effectively, we can tell how long a particular transaction has been suspended by the GC. (See Figure 6.7.)



**Figure 6.7**: The two bars represent the transaction duration with and without GC suspension. Meaning the difference represents the performance impact of the Garbage Collection

Only a handful of tools allow the direct monitoring of suspensions, dynaTrace being one of them. If you do not have such a tool you can use jStat, JConsole, or a similar tool to monitor GC times. The metrics reported in jStat are also directly exposed by the JVM via JMX. This means you can use any JMX-capable monitoring solution to monitor these metrics.

It is important to differentiate between young- and old-generation GCs (or minor and major, as they are sometimes called; see more in the next section) and equally important to understand both the frequency and the duration of GC cycles. Young-generation GCs will be mostly short in duration but under heavy load can be very frequent. A lot of quick GCs can be as bad for performance as a single long-lasting one (remember that young-generation GCs are always stop-the-world events).

Frequent young-generation GCs have two root causes:

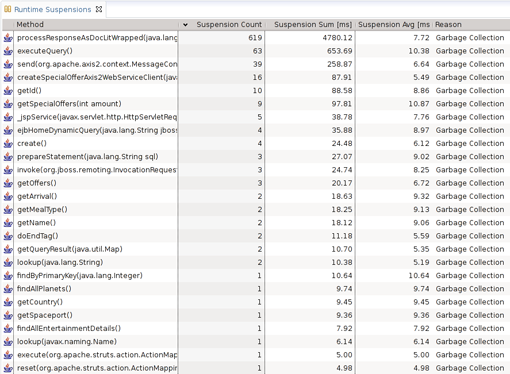
1. A too-small young generation for the application load.
2. High churn rate

If the young generation is too small, we see a growing old generation due to prematurely tenured objects.

If too many objects are allocated too quickly (i.e., if there’s a high churn rate), the young generation fills up as quickly and a GC must be triggered. While the GC can still cope without overflowing to the old generation, it has to do so at the expense of the application’s performance.

A high churn rate might prevent us from ever achieving an optimal generation sizing, so we must fix such a problem in our code before attempting to optimize garbage collection itself. There is a relatively easy way to identify problematic code areas.

The runtime suspensions shown in Figure 6.8 report a massive statistical concentration of garbage collections in one particular function. This is very unusual, because a GC is not triggered by a particular method, but rather by the fillstate of the heap. The fact that one method is suspended more than others suggests that the method itself allocates enough objects to fill up the young generation and thus triggers a GC. Whenever we see such a statistical anomaly we find a prime candidate for allocation analysis and optimization (which we’ll examine further in the Allocation Analysis section of this chapter, below).



**Figure 6.8**: This shows how often and for how long a certain method has been suspended by Garbage collection

### Major vs. Minor Garbage Collections

What I have been referring to as young- and old-generation GCs, are commonly referred to as minor and major GCs. Similarly, it is common knowledge that major GCs suspend your JVM, are bad for performance, and should be avoided if possible. On the other hand, minor GCs are often thought to be of no consequence and are ignored during monitoring. As already explained, minor GCs suspend the application and neither they nor major GCs should be ignored. A major GC is often equated with garbage collection in the old generation; this is, however, not fully correct. While every major GC collects the old generation, not every old-generation GC is a major collection. Consequently, the reason we should minimize or avoid major GCs is misunderstood. Looking at the output of verbose:GC explains what I mean:

[GC 325407K->83000K(776768K), 0.2300771 secs]

[GC 325816K->83372K(776768K), 0.2454258 secs]

[Full GC 267628K->83769K(776768K), 1.8479984 secs]

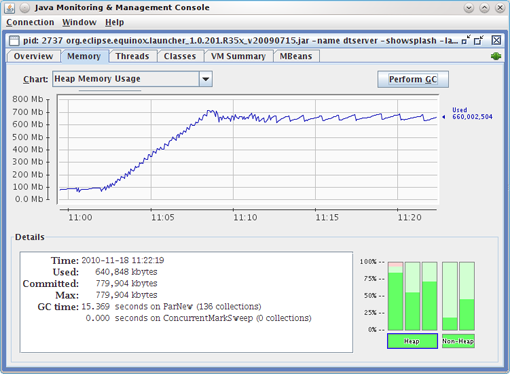
A major GC is a full GC. A major GC collects all areas of the heap, including the young generation and, in the case of the Oracle HotSpot JVM, the permanent generation. Furthermore, it does this during a stop-the-world event, meaning the application is suspended for a lengthy amount of time, often several seconds or minutes. On the other hand, we can have a lot of GC activity in the old generation without ever seeing a major (full) GC or having any lengthy GC-caused suspensions. To observe this, simply execute any application with a concurrent GC. Use jstat –gcutil to monitor the GC behavior of your application.

This output is reported by jstat -gcutil. The first five columns in Table 6.9 show the utilization of the different spaces as a percentage—survivor 0, survivor 1, Eden, old, and permanent—followed by the number of GC activations and their accumulated times—young GC activations and time, full GC activations and time. The last column shows the accumulated time used for garbage collection overall. The third row shows that the old generation has shrunk and jStat reported a full GC (see the bold numbers).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S0** | **S1** | **E** | **O** | **P** | **YGC** | **YGCT** | **FGC** | **FGCT** | **GCT** |
| 90.68 | 0.00 | 32.44 | 85.41 | 45.03 | 134 | 15.270 | 143 | 21.683 | 36.954 |
| 90.68 | 0.00 | 32.45 | 85.41 | 45.03 | 134 | 15.270 | 143 | 21.683 | 36.954 |
| 90.68 | 0.00 | 38.23 | ***85.41*** | 45.12 | 134 | 15.270 | ***143*** | ***21.683*** | ***36.954*** |
| 90.68 | 0.00 | 38.52 | ***85.39*** | 45.12 | 134 | 15.270 | ***144*** | ***21.860*** | ***37.131*** |
| 90.68 | 0.00 | 38.77 | 85.39 | 45.12 | 134 | 15.270 | 144 | 21.860 | 37.131 |
| 90.68 | 0.00 | 49.78 | 85.39 | 45.13 | 134 | 15.270 | 145 | 21.860 | 37.131 |

Table 6.9: jstat -gcutil output report

On the other hand, monitoring the application with JConsole (Figure 6.10), we see no GC runs reported for the old generation, although we see a slightly fluctuating old generation.



**Figure 6.10**: JConsole shows that the amount of used memory in the old generation is fluctuating. At the same time it shows that not a single garbage collection was executed in the old generation.

Which of the two tools, jStat (Figure 6.13) or JConsole (Figure 6.14) is correct?

In fact, both are only partially correct, which is rather misleading. as reported by jStat, the concurrent GC has executed, but it has done so asynchronously to the application and for the old generation only, which is its purpose. It was not a full GC! JStat reports old-generation GCs as full GCs, which is wrong. This is most likely a legacy from days prior to concurrent and incremental GCs.

JConsole reports activations via Java Management Extensions (JMX) and managed beans (MBeans). Previously, these MBeans reported only real major GCs. In the case of the CMS, these occur only when the CMS is not able to do its work concurrent to the application, due to memory fragmentation or high churn rate. For this reason, JConsole would not show any activations.

In a recent release the CMS, Memory MBean has been changed to report only activations of the CMS itself; the downside is that we have no way to monitor real major GCs anymore.

In the IBM WebSphere JVM, major and minor GCs are indistinguishable via JConsole. The only way to distinguish between them is by using the verbose:gc flag.

For these reasons, we mistakenly ignore minor GCs and overrate old-generation GCs by considering them to be major. The truth is we need to monitor JVM suspensions and understand whether the root causes lie in the young generation, due to object churn, or the old generation, due to wrong sizing or a memory leak.

## Analyzing GC for Maximum Throughput

We have discussed the impact of suspensions have on application response time. However, transaction rates or more important for throughput or batch applications. Consequently we do not care about the pause-time impact to a particular transaction, but about the CPU usage and overall suspension duration.

Consider the following. In a response-time application it is not desirable to have a single transaction suspended for 2 seconds. It is, however, acceptable if each transaction is paused for 50 ms. If this application runs for several hours the overall suspension time will, of course, be much more than 2 seconds and use a lot of CPU, but no single transaction will feel this impact! The fact that a throughput application cares only about the overall suspension and CPU usage must be reflected when optimizing the garbage-collection configuration.

To maximize throughput the GC needs to be as efficient as possible, which means executing the GC as quickly as possible. As the whole application is suspended during the GC, we can and should leverage all existing CPUs. At the same time we should minimize the overall resource usage over a longer period of time. The best strategy here is a parallel full GC (in contrast to the incremental design of the CMS).

It is difficult to measure the exact CPU usage of the garbage collection, but there’s an intuitive shortcut. We simply examine the overall GC execution time. This can be easily monitored with every available free or commercial tool. The simplest way is to use jstat -gc 1s. This will show us the utilization and overall GC execution on a second-by-second basis (see Table 6.11).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S0** | **S1** | **E** | **O** | **P** | **YGC** | **YGCT** | **FGC** | **FGCT** | **GCT** |
| 0.00 | 21.76 | 69.60 | 9.17 | 75.80 | **63** | **2.646** | **20** | **9.303** | **11.949** |
| 0.00 | 21.76 | 71.24 | 9.17 | 75.80 | **63** | **2.646** | **20** | **9.303** | **11.949** |
| 0.00 | 21.76 | 71.25 | 9.17 | 75.80 | **63** | **2.646** | **20** | **9.303** | **11.949** |
| 0.00 | 21.76 | 71.26 | 9.17 | 75.80 | **63** | **2.646** | **20** | **9.303** | **11.949** |
| 0.00 | 21.76 | 72.90 | 9.17 | 75.80 | **63** | **2.646** | **20** | **9.303** | **11.949** |
| 0.00 | 21.76 | 72.92 | 9.17 | 75.80 | **63** | **2.646** | **20** | **9.303** | **11.949** |
| 68.74 | 0.00 | 1.00 | 9.17 | 76.29 | ***64*** | **2.719** | **20** | **9.303** | ***12.022*** |
| 68.74 | 0.00 | 29.97 | 9.17 | 76.42 | **64** | **2.719** | **20** | **9.303** | **12.022** |
| 68.74 | 0.00 | 31.94 | 9.17 | 76.43 | **64** | **2.719** | **20** | **9.303** | **12.022** |
| 68.74 | 0.00 | 33.42 | 9.17 | 76.43 | **64** | **2.719** | **20** | **9.303** | **12.022** |

**Table 6.11**: Rather than optimizing for response time, we need only look at the overall GC time.

Using graphical tools, the GC times can be viewed in charts and thus correlated more easily with other metrics, such as throughput. In Figure 6.12 we see that while the garbage collector is executed, suspending quite often in all three applications, it consumes the most time in the GoSpaceBackendit this does not seem to have an impact on throughput.

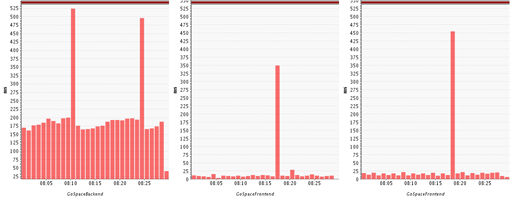


Figure 6.11: The chart shows the duration of GC suspensions of three different JVMs in context to throughput for a particular JVM.

It can be safely assumed that it also requires far more CPU resources than in the other JVMs and has a higher negative impact. With increasing load, the GC times will also rise; this is normal! However, the GC time should never take up more than 10% of the total CPU time used by the application. If CPU usage rises disproportionately to application load, or if the usage is disproportionately heavy, one must consider a change in the GC configuration or an improvement in the application.

## Allocation Analysis

Allocating objects in Java is relatively inexpensive compared to C++, for instance. The generational heap is specifically geared for fast and frequent allocations of temporary objects. But it’s not free! Allocations can quickly become a choke point when a lot of concurrent threads are involved. Reasons include memory fragmentation, more frequent GCs due to too many allocations, and synchronization due to concurrent allocations. Although you should refrain from implementing object pools for the sole sake of avoiding allocations, not allocating an object is the best optimization.

The key is to optimize your algorithms and avoid making unnecessary allocations or allocating the same object multiple times. For instance, we often create temporary objects in loops. Often it is better to create such an object once prior to the loop and just use it. This might sound trivial, but depending on the size of object and the number of loop recursions, it can have a high impact. And while it might be the obvious thing to do, a lot of existing code does not do this. It is a good rule to allocate such temporary “constants” prior to looping, even if performance is no consideration.

Allocation analysis is a technique of isolating areas of the application that create the most objects and thus provide the highest potential for optimization. There are a number of free tools, such as JvisualVM, for performing this form of analysis. You should be aware that these tools all have enormous impact on runtime performance and cannot be used under production-type loads. Therefore we need to do this during QA or specific performance-optimization exercises.

It is also important to analyze the correct use cases, meaning it is not enough to analyze just a unit test for the specific code. Most algorithms are data-driven, and so having production-like input data is very important for any kind of performance analysis, but especially for allocation analysis.

Let's look at a specific example. I used JMeter to execute some load tests against one of my applications. Curiously, JMeter could not generate enough load and I quickly found that it suffered from a high level of GC activity. The findings pointed towards a high churn rate (lots of young-generation GCs). Consequently I used an appropriate tool (in my case, JVisualVM) to do some allocation hot-spot analysis. Before I go into the results, I want to note that it is important to start the allocation analysis after the warm-up phase of a problem. A profiler usually shows hotspots irrespective of time or transaction type. We do not want to skew those hot spots with data from the initial warm-up phase (caching and startup). I simply waited a minute after the start of JMeter before activating the allocation analysis. The actual execution timing of the analysis application must be ignored; in most cases the profiler will lead to a dramatic slowdown.

The analysis (Figure 6.13) shows which types of objects are allocated most and where these allocations occur. Of primary interest are the areas that create many or large objects (large objects create a lot of other objects in their constructor or have a lot of fields). We should also specifically analyze code areas that are known to be massively concurrent under production conditions. Under load, these locations will not only allocate more, but they will also increase synchronization within the memory management itself. In my scenario I found a huge number of Method objects were allocated (Figure 6.16). This is especially suspicious because a Method object is immutable and can be reused easily. There is simply no justification to have so many new Method objects.

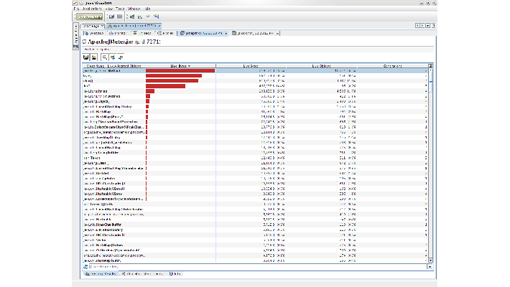
[](http://javabook.compuware.com/content/memory/img/jvisualvm%20allocations.png)

Figure 6.13: JVisualVM tells me that Method objects are the most allocated ones in my application.

By looking at the origin of the allocation (Figure 6.14), I found that the Interpreter object was created every time the script was executed instead of being reused. This led to the allocations of the Method objects. A simple reuse of that Interpreter object removed more than 90% of those allocations.

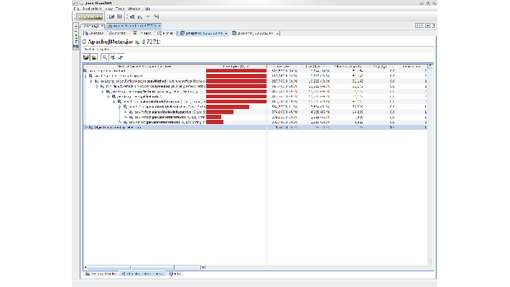
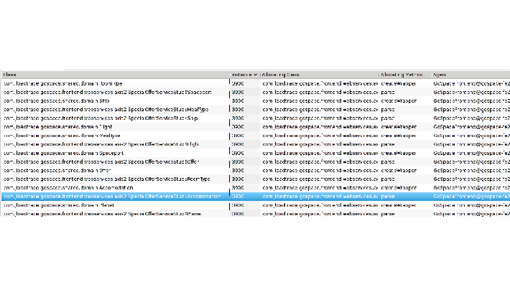
[](http://javabook.compuware.com/content/memory/img/jvisualvm%20allocations1.png)

Figure 6.14: JVisualVM tells where the allocations of the method object happen in my application code.

To avoid going quickly astray, it is important to check each optimization for a positive effect. The transaction must be tested before and after the optimization under full load and, naturally, without the profiler, and the results compared against each other. This is the only way to verify the optimization.

Some commercial tools, such as dynaTrace (see Figures 6.15 and 6.16), also offer the ability to test allocations on a transaction basis under load conditions. This has the added advantage of taking into account data-driven problems and concurrency issues, which are otherwise difficult to reproduce in a development setup.

[](http://javabook.compuware.com/content/memory/img/allocations.png)Figure 6.15: dynaTrace tells me how often a particular object has been allocated in a load-test scenario.

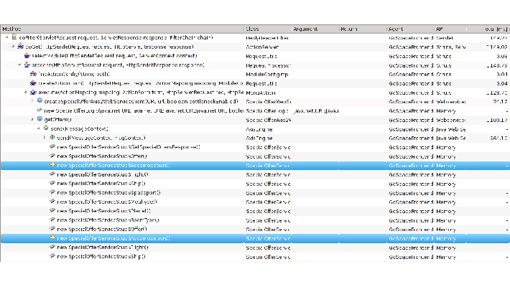
[](http://javabook.compuware.com/content/memory/img/allocations1.png)

Figure 6.16: In addition, dynaTrace can tell me where in my transaction the object was allocated.

## Memory Analysis – Heap Dump

So far we have dealt with the influence of garbage collection and allocations on application performance. I have stated repeatedly that high memory utilization is a cause for excessive garbage collection. In many cases, hardware restrictions make it impossible to simply increase the heap size of the JVM. In other cases increasing the heap size does not solve but only delays the problem because the utilization just keeps growing. If this is the case, it is time to analyze the memory usage more closely by looking at a heap dump.

A heap dump is a snapshot of main memory. It can be created via JVM functions or by using special tools that utilize the JVM TI. Unfortunately, JVM-integrated heap dumps are not standardized. Within the Oracle HotSpot JVM, one can create and analyze heap dumps using [jmap](http://docs.oracle.com/javase/7/docs/technotes/tools/share/jsdocs/jmap.html), [jhat](http://docs.oracle.com/javase/6/docs/technotes/tools/share/jhat.html), and [VisualVM](http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html). JRockit includes its JRockit Mission Control, which offers a number of functions besides memory analysis itself.

The heap dump itself contains a wealth of information, but it’s precisely this wealth that makes analysis hard—you get drowned easily in the quantity of data. Two things are important when analyzing heap dumps: a good Tool and the right technique. In general, the following analyses are possible:

* Identification of memory leaks
* Identification of memory eater

### Identifying Memory Leaks

Every object that is no longer needed but remains referenced by the application can be considered a memory leak. Practically we care only about memory leaks that are growing or that occupy a lot of memory. A typical memory leak is one in which a specific object type is created repeatedly but not garbage-collected (e.g., because it is kept in a growing list or as part of an object tree). To identify this object type, multiple heap dumps are needed, which can be compared using trending dumps.

Trending dumps only dump a statistic of the number of instances on the class level and can therefore be provided very quickly by the JVM. A comparison shows which object types have a strongly rising tendency. Every Java application has a large number amount of String, char[], and other Java-standard objects. In fact, char[] and String will typically have the highest number of instances, but analyzing them would take you nowhere. Even if we were leaking String objects, it would most likely be because they are referenced by an application object, which represents the root cause of the leak. Therefore concentrating on classes of our own application will yield faster results.

Figure 6.17 shows a view on multiple trending dumps that I took from a sample application I suspected of having a memory leak. I ignored all standard Java objects and filtered by my application package. I immediately found an object type that constantly increased in number of instances. As the object in question was also known to my developer, we knew immediately that we had our memory leak.

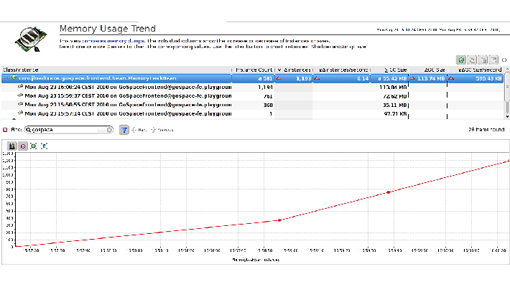
[](http://javabook.compuware.com/content/memory/img/2trenddump.png)

Figure 6.17: The trending dump shows which objects increase in number over time.

### Identifying Memory-Eaters

There are several cases when you want to do a detailed memory analysis.

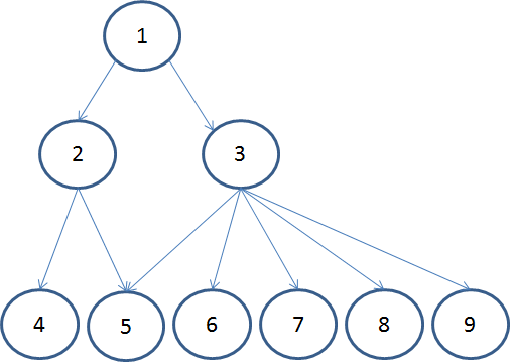
* The Trending Analysis did not lead you to your memory leak
* Your application uses to much memory, but has no obvious memory leak, and you need to optimize
* You could not do a trending analysis because memory is growing too fast and your JVM is crashing

In all three cases the root cause is most likely one or more objects that are at the root of a larger object tree. These objects prevent a whole lot of other objects in the tree from being garbage-collected. In case of an out-of-memory error it is likely that a handful of objects prevent a large number of objects from being freed, hence triggering the out-of-memory error. The purpose of a heap-dump analysis is to find that handful of root objects.

The size of the heap is often a big problem for a memory analysis. Generating a heap dump requires memory itself. If the heap size is at the limit of what is available or possible (32-bit JVMs cannot allocate more than 3.5 GB), the JVM might not be able to generate one. In addition, a heap dump will suspend the JVM. Depending on the size, this can take several minutes or more. It should not be done under load.

Once we have the heap dump we need to analyze it. It is relatively easy to find a big collection. However, manually finding the one object that prevents a whole object tree (or network) from being garbage-collected quickly becomes the proverbial needle in a haystack.

Fortunately solutions like dynaTrace are able to identify these objects automatically. To do this they use a dominator algorithm that stems from graph theory. This algorithm is able to calculate the root of an object tree. In addition to calculating object-tree roots, the memory-analysis tool calculates how much memory a particular tree holds. This way it can calculate which objects prevent a large amount of memory from being freed—in other words, which object dominates memory (see Figure 6.18).



**Figure 6.18**: This is a schematic representation of an object tree. In this presentation object 1 is the root of the tree and thus dominates all the others.

Let us examine this using the example in Figure 6.18. Object 1 is the root of the displayed object tree. If object 1 no longer references object 3, all objects that would be kept alive by 3 are garbage-collected. In this sense, object 3 dominates objects 6 ,7 ,8, and 9. Objects 4 and 5 cannot be garbage-collected because they are still referenced by one other (object 2). However, if object 1 is dereferenced, all the objects shown here will be garbage-collected. Therefore, object 1 dominates them all.

A good tool can calculate the dynamic size of object 1 by totaling the sizes of all the objects it dominates. Once this has been done, it is easy to show which objects dominate the memory. In Figure 6.18, objects 1 and 3 will be the largest hot spots. A practical way of using this is in a memory hotspot view as we have in Figure 6.19:

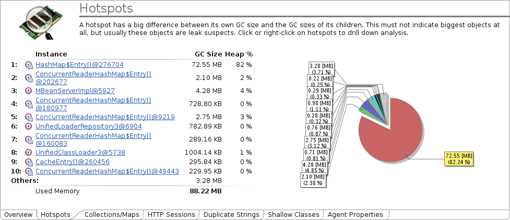
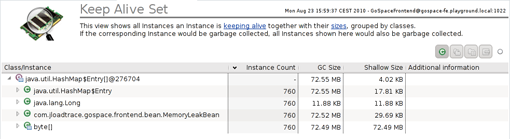


Figure 6.19: This hotspot view shows that the HashMap$Entry array dominates, with more than 80 percent of the heap.

The HashMap$Entry array dominates—it accounts for more than 80 percent of the heap. Were it to be garbage-collected, 80% of our heap would be empty. From here it is a relative simple matter to use the heap dump to figure out why. We can then easily understand and show why that object itself has not been garbage-collected. Even more useful is to know which objects are being dominated and make up that hot spot. In dynaTrace we call this the Keep Alive Set (see Figure 6.20).



**Figure 6.20**: The keep alive set shows the objects that are being dominated or kept alive by the HashMap$Entry Array.

With the aid of such analyses, it is possible to determine within a few minutes whether there is a large memory leak or whether the cache uses too much memory.

### JMeter Concepts

JMeter is organized with a tree metaphor. This tree is visible on the left side of the screen and property sheets for each tree element appear on the right. JMeter is configured with a Test Plan, which is the root element of the tree. Test Plans are containers that store everything else that describes your test. JMeter also contains a WorkBench, which is for temporary storage of test elements and isn't needed for operation. Test Plans can contain multiple Thread Groups, and Thread Groups can contain:

* Timers.
* Listeners.
* Controllers.
* Configuration Elements.

JMeter can also assign multiple threads to your tests. The default number of threads for a Thread Group is 1, but this is easy to change. Use multiple Thread Groups in your test plan to simulate multiple users.

Timers are used to determine how much load to put on your application. They control how often JMeter sends a request to your application. If you want to emulate 10 users that each click a new page every three seconds, set the delay on the timer to 300 milliseconds. JMeter includes one constant delay timer and two timers that offset the delay by a random deviation.

Listeners are the output handlers for JMeter. JMeter outputs the amount of time, in milliseconds, that it takes for a web server to respond to the JMeter request. JMeter should have at least one listener defined, possibly more. There are four listeners in JMeter:

* File Reporter: logs output to a user-defined file. Be sure to open the file from the configuration panel before running your test and close the file when you are finished.
* View Results: displays the web page requested from the server as HTML source.
* Graph Results: displays each response time as a dot on an x-y graph. JMeter also tracks the average and the deviation.
* Spline Visualizer: generates a smooth curve of the results over time.

Controllers can be split into two groups — protocols and logic. Protocol controllers are Web Testing, Database Testing, and FTP Testing. These controllers are configured with user names, passwords, URLs, and so on. The HTTP and JDBC controllers are the most useful. However, the FTP controller is probably most useful if you are writing an FTP server.

Logic controllers are the Once Only controller and the Interleave controller. The Once Only controller can be used if a given page needs to be initialized, but not afterwards. The Interleave controller is for simulating multiple user names and passwords with one thread group.

Configuration elements let you set up controllers with additional information. Each controller can be set up with multiple test cases, which are called "samples" in JMeter. These samples include the FTP Sample, the Url Sample, and the Sql Query. If you are running just one test case in a test plan, it's easier to configure the controllers themselves and not use configuration elements.

The Web Testing controller also makes use of the Cookie Manager and HTTP Authorization configuration elements. The Cookie Manager stores cookies that you can set up. This mimics the cookie functionality of a web browser. HTTP Authorization stores user names and passwords for base URLs that require a login. This authentication is done at the HTTP protocol level, which is why there is a separate configuration element.

The Database Testing controller uses the Database Login and Database Connection Pool configuration elements. The Database Login can also be configured on the controller. Database Connection Pool is used for sharing connections between tests. This could be useful if you have limited client licenses for your database, for example.

### Using JMeter

JMeter is open-source software and can be freely downloaded at http://jakarta .apache.org/jmeter/. JMeter can also use three optional libraries — a SAX XML parser, the JavaMail API, and JSSE — which enables SSL support for testing web applications that use HTTPS.

To demonstrate how you can use JMeter to test and improve performance, I will use two Java servlets that run under Tomcat (another open-source program from the Jakarta Apache project; http://jakarta.apache.org/site/binindex.html). The first servlet prints all the numbers from 0 to 511 on an HTML page, while the second servlet does exactly the same thing — only faster.

JMeter releases before 1.6 final are unable to change the port number that JMeter uses for HTTP testing. To remedy this, you will need to run Tomcat on port 80, the normal HTTP server port. By default, Tomcat uses port 8080, and this is configured in the <tomcat-dir>/conf/server.xml file. Do a text search for "8080." On my release of Tomcat, this was line 225, and was under an XML comment that said "Normal HTTP." Change the "8080" to "80." For UNIX users, this means you will have to run Tomcat as root, as normal users on UNIX can have daemons that listen to port numbers less than 1024.

To launch Tomcat with the servlet, run startup.bat (for Windows) or startup.sh (for UNIX) in the /bin subdirectory of Tomcat. Load the servlet in your web browser with the URL http://localhost/ jmeter/SlowExample to be sure it installed properly.

### Performance Problems

The easiest way to see how JMeter works is to use a real web application that has been coded in two different ways. I call these "Slow" and "Fast." They both do the same thing, but the implementation is different. I start by using JMeter to test the slow example.

1. Start JMeter from the command line. JMeter displays in a new window;

2. Right-click the Test Plan element in the tree on the left side and select the Add submenu from the menu that appears. Select Thread Group from that submenu.

3. The Thread Group is nested under the Test Plan, but won't be visible until the expansion icon to the left of Test Plan is clicked.

4. Select the Thread Group element and a Panel appears on the right side of the display. Change the Name to SlowExample. The number of threads doesn't need to be changed.

5. Add a Timer to the SlowExample thread group. Right-click on SlowExample, select the Add submenu, then the Timer submenu, and finally, the Constant Timer.

6. Expand the SlowExample element and click on the Constant Timer. The constant delay should be 300, which is in milliseconds. Change this number to 100, to put more of a load on the application being tested;

7. Right-click on SlowExample again and add two Listeners. You are going to use a View Results Listener and a Graph Results Listener.

8. The View Results Listener does not need to be configured. The Graph Results Listener has a name, which you can change to SlowGraph. These two panels will contain results data once the test plan is run.

9. Add a Web Testing Controller to SlowExample. Select the Web Testing Controller and change the domain to localhost (or whichever machine is running Tomcat). Set the path to /jmeter/SlowExample, and the method to Get. If these are not set properly, JMeter may complain that the *start* function is not implemented.

10. To run the first test, select the SlowGraph Graph Results Listener from the tree on the left-hand side, and start the test script with the Start command in the Run menu.

11. The first result you get on the Graph Results Listener will probably be slow. This is the time it takes for JMeter to compile the script. It is best to start JMeter, run the script for a few seconds, and then stop JMeter with the Stop command in the Run menu. Restart the test script after clearing the results with the Clear command, also in the Run menu.

12. The View Results Listener will show you the HTML code returned from the server for the request. This is mostly useful for debugging purposes.

13. Stop the test with the Stop command on the Run menu.

Now improve the servlet's performance. This servlet uses some poor Java programming techniques. The use of the *concat()* method on *String* classes should be avoided in Java because the *String* class is immutable, which means a new string has to be constructed in memory. Instead, use *StringBuffer* classes, with the *append()* method. This is the recommended way of adding two strings together. If you're not a Java programmer, the details of this performance improvement isn't actually important; what is important is that I will increase the speed of the servlet in the loop. I'll also append the loop counter, *ctr*, as an integer, without turning it into a string as an intermediate step;

So that you can get a text file of the data, add a File Reporter to the Thread Group:

14. Clear the testing results for SlowExample (Run->Clear).

15. Select SlowExample and change the name to FastExample.

16. Select SlowGraph and change the name to FastGraph.

17. Select Web Testing and change the path to /jmeter/FastExample.

18. Add a File Reporter to FastExample. Right-click FastExample, select the Add submenu, and under the submenu Listeners, choose File Reporter.

19. Select the File Reporter and set the output file to fast.log. Make sure the Append and Verbose Output checkboxes are checked. Click the Open button;.

20. Select FastGraph, and then run the test case. Be sure to clear the results out once the first few hits have occurred.

21. When you're finished with the test, select the File Reporter and close the file, so it gets written to disk and the logging stops; On my machine, the new Fast servlet took an average of four milliseconds to run — quite an improvement.

### Interpreting Results

Keep a spreadsheet with response times, dates, machine names, and pages tested, along with previous logs of data from the File Reporter. This lets you analyze historical data; if the response time for a page jumped a month ago, check the source control system to see what changes were made to the code.

If your application is backed by a database, graph response times versus the amount of data in the system. This is easy to do with a spreadsheet. If your response times go up in a linear trend, your database doesn't have the proper indexes or your algorithm may need to be rewritten.