

Appendix C. Digging Through Your Garbage

Popular criticisms of the JVM include that it's prone to 'memory leaks', and that it performs frequent and lengthy garbage collections at unpredictable times. The collection pauses are especially painful for games which should guarantee fast responsiveness at all times. Since Java doesn't offer a version of C's `free()` or C++'s `delete()`, then the programmer has no way to control or guide the JVM's feeble-minded garbage collector.

The main aim of this appendix is to show that the preceding paragraph is mostly rubbish (or, more aptly, garbage):

- Memory leaks can be easily detected and fixed. Memory leaks aren't the fault of the run time system, but a symptom of bad programming style.
- The JVM supports a vast range of garbage collection capabilities (including *six* different collectors).
- It's straightforward to adjust the JVM's behaviour to change the length and frequency of garbage collections. For example, it can be configured to perform numerous short collections, or fewer longer-duration collections.
- Although Java doesn't offer `free()` or `delete()`, there are programming techniques which can encourage and help garbage collection. However, most 'tweaking' of the collection strategies is done via command line options to the JVM.

It's worth mentioning that the poor performance of a Java application isn't necessarily the fault of the JVM. Many games use external resources, such as OpenGL or DirectX, databases, and the Internet, which are beyond Java's control. There may be problems with the underlying operating environment, such as low memory, a slow CPU, or a poorly configured network.

An Outline of this Appendix

I'll start by considering memory leaks: what they are, how to find them in code, and how to get rid of them.

I'll supply background information on the generational approach to garbage collection used in the JVM, information I'll need in order to decide how to tweak the collectors.

I'll describe several techniques for reducing collector pause time, by experimenting with two examples: *JumpingJack* (the side-scroller from chapter 12) and *FPSHooter3D* (the first-person shooter from chapter 24). The basic approach is to supply options to the JVM; no changes are needed to the application code.

I'll graphically display the copious amounts of performance information generated by the JVM. Garbage collections statistics will be visualized with *HPjtune* (<http://www.hp.com/products1/unix/java/java2/hpjtune/>), and profiling details rendered with *HPjmeter* (<http://www.hp.com/products1/unix/java/hpjeta/>). Both are freeware tools that work with a wide range of JVMs.

I'll finish by describing techniques for gathering heap-related statistics by adding code to Java applications.

An underlying assumption of this appendix is that the programmer will be using J2SE 1.4.2 or 5.0. There have been many changes to the JVM over the years, and some of the techniques I'll be discussing don't work in earlier JVMs.

1. Dealing with Memory Leaks

The phrase "memory leak" is quite misleading, especially to novice programmers. It's suggestive of data structures that are somehow 'leaking' into other parts of memory, corrupting the heap, causing the application to gradually go insane and die.

In reality, memory leaks are data structures stored in the heap, which the programmer forgets to remove. The heap gradually fills up, causing the run time system to slow down and perhaps crash.

In Java, a memory leak is an object that's never dereferenced, and so can't be garbage collected. Dereferencing is the decoupling of an object from all its references (its names). For example:

```
Stack stk1 = new Stack();
stk1.push(42);
int x = stk1.pop();
stk1 = null;
```

After `stk1` is assigned `null`, its `Stack` object is dereferenced, and so becomes eligible for garbage collection.

Some authors prefer to use the phrase "unintentionally retained object" rather than "memory leak"; I'll stick with the more popular phrase.

1.1. A Memory Leak Example

The following program illustrates a common form of memory leak: forgetting to clean up collections. The `MemoryLeak` class is based on an example in: *Java Doctor*, by Ali Syed and Jamiel Shiekh. A draft version of their profiling chapter is available at <http://www.theserverside.com/articles/article.tss?l=JavaDoctorBookInReview>.

```
import java.util.*;

public class MemoryLeak
{
    public static void main(String[] args) throws Exception
    {
        ArrayList arrList = new ArrayList();
        StringBuffer sb;

        for(int i=0; i < 70000000; i++){
            sb = new StringBuffer(100); // new object
            if(i%50 == 0) { // add every 50th object to list
                arrList.add(sb);
                Thread.currentThread().sleep(2); // slows down execution
            }
            // if(i%8000 == 0) arrList.clear(); // fixes memory leak
        }
    }
}
```

```
    }  
  }  
}
```

The for-loop creates many StringBuffer objects. However, the assignment of a new StringBuffer object to sb at the start of a loop iteration causes the object created in the previous iteration to be dereferenced. As a consequence, the previous sb object will eventually be garbage collected.

Unfortunately, every 50th StringBuffer object also has an additional reference stored in the arrList ArrayList. These extra references mean that their objects can't be deleted, even though the sb reference was re-assigned.

The gradual growth of the ArrayList, and the use of sleep(), lets the memory leak develop slowly. It takes several minutes for the heap to fill up and trigger an OutOfMemory exception:

```
java MemoryLeak  
Exception in thread "main" java.lang.OutOfMemoryError: Java heap  
space
```

This type of failure is quite typical in real applications with memory leaks: the leak grows gradually, and so may not be noticed during the program's development and testing phases.

1.2. Visualizing the Memory Leak

Information on the way the heap changes over time can be collected by calling java.exe with the `-Xloggc:<file>` option: the garbage collection calls are logged in the specified file. For example:

```
java -Xloggc:gc.txt MemoryLeak
```

The application should be run for several minutes at least, and then the gc.txt log file examined using HPjtune (downloadable from <http://www.hp.com/products1/unix/java/java2/hpjtune/>):

```
java -jar HPjtune.jar
```

HPj tune has several windows that display garbage collection data in different forms. The "Heap Usage After GC" window is the most useful for detecting memory leaks, since it plots heap usage after collection against time. The graph for a 158 second run of MemoryLeak is shown in Figure 1.

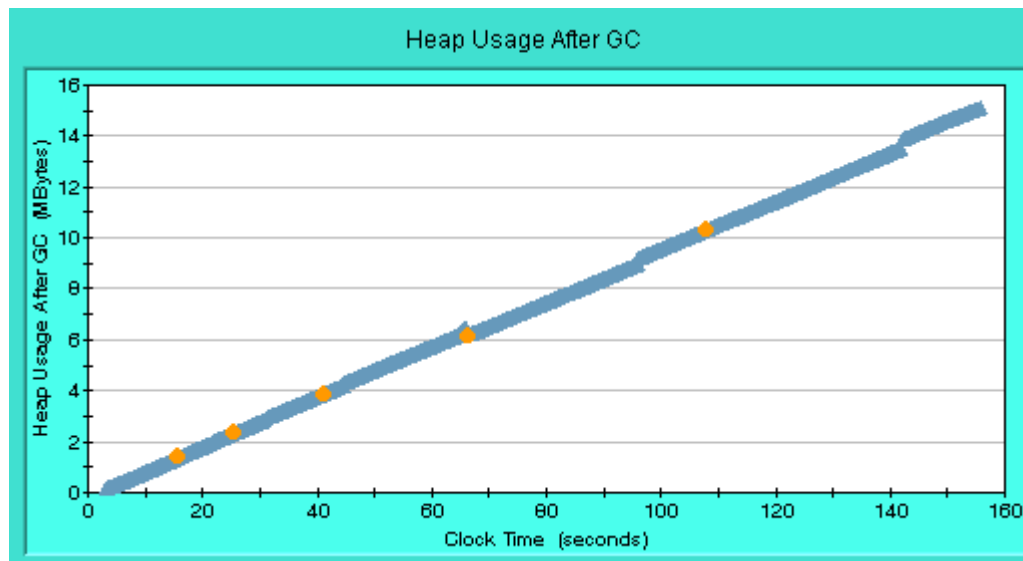


Figure 1. Heap Usage Against Time in HPj tune.

Several kinds of coloured shapes are used as plot coordinates, but their meaning is irrelevant for memory leaks; the important point is that the graph is continuously rising, a sure sign of a memory leak.

In a normal application, the curve should rise sharply at the start, as the heap adjusts itself to the memory needs of the application. Then it should level off, perhaps with periodic dips and rises as garbage is collected and new objects created.

1.3. Finding the Memory Leak in the Code

For a simple program like MemoryLeak.java, the cause of the memory leak is fairly obvious, but this is unlikely to be the true for real applications.

I can get the JVM to help me in my detective work, by asking it to generate profiling data. The data will include a method call graph, and the numbers and sizes of object created. Also of use is information on the objects left when the application terminates (sometimes called *residual* objects).

A memory leak will reveal itself as a large number of residual objects of a single type, which appear in the profile irrespective of when I terminate the application. The profile's call graph can help me find where all those objects are being created in the program.

The application is called with profiling options set:

```
java -Xrunhprof:heap=sites,cpu=samples MemoryLeak
```

The JVM will write sites and samples details to the java.hprof.txt textfile. The sites information lists the heap allocations for objects, cross-indexed by stack traces for the method calls that make those objects. The samples information gives CPU usage details for those traces.

I can view the profiling data graphically with HPjmeter
(<http://www.hp.com/products1/unix/java/hpjetaer/>):

```
java -jar HPjetaer.jar
```

A first step to finding memory leaks is to get an idea of the most common residual objects. The "Residual Objects (Count)" window for MemoryLeaks is shown in Figure 2.

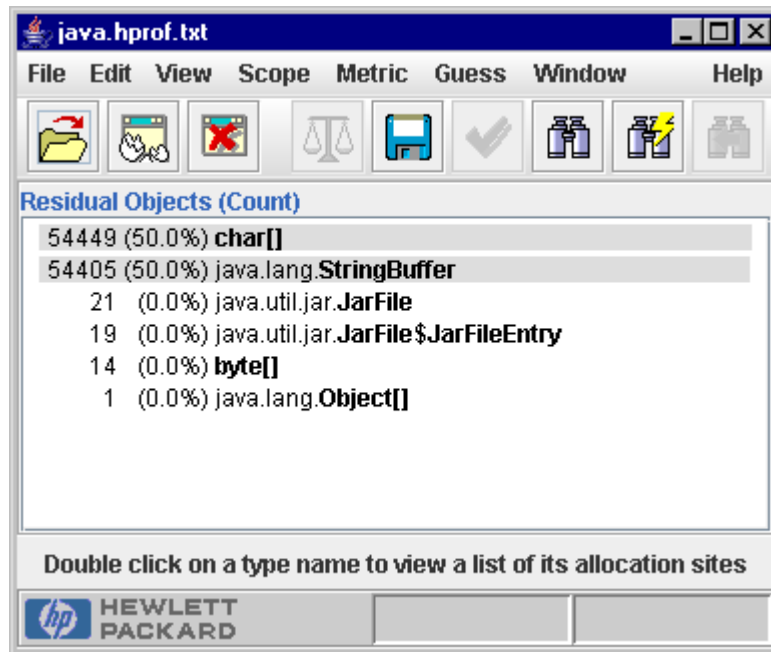
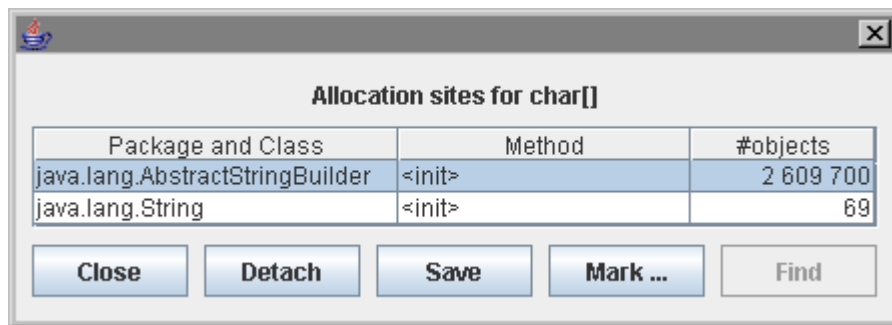


Figure 2. Residual Objects (Count) for MemoryLeaks.java.

These numbers should be compared with the number of created objects, listed in HPjmeter's "Created Objects (Count)" window. In this run of MemoryLeak, over 2.6 million char arrays and StringBuffer objects were created. The inability to collect so many objects (about 110,000) strongly suggests that the memory leak is occurring with those char arrays and StringBuffers.

Aside from counts of objects, the "Created Objects (Bytes)" and "Residual Objects (Bytes)" windows displays statistics about the sizes of the objects.

Double clicking on the `char[]` line in the residual objects window in Figure 2 brings up an allocation sites window (Figure 3). It lists the methods and classes where the data was created.



Package and Class	Method	#objects
java.lang.AbstractStringBuilder	<init>	2 609 700
java.lang.String	<init>	69

Figure 3. Allocation Sites for the Char Arrays.

The creator of the char arrays is the constructor method in `AbstractStringBuilder`. A look at the allocation sites for the 54,405 `StringBuffer` objects shows the `AbstractStringBuilder` constructor to be their creator as well.

I selected and 'marked' the `<init>` (constructor) method for `AbstractStringBuilder` in the allocation sites window, so I can conveniently search for it later.

I now switched to the "Call Graph Tree (CPU)" window, to see which methods were called most frequently (Figure 4).

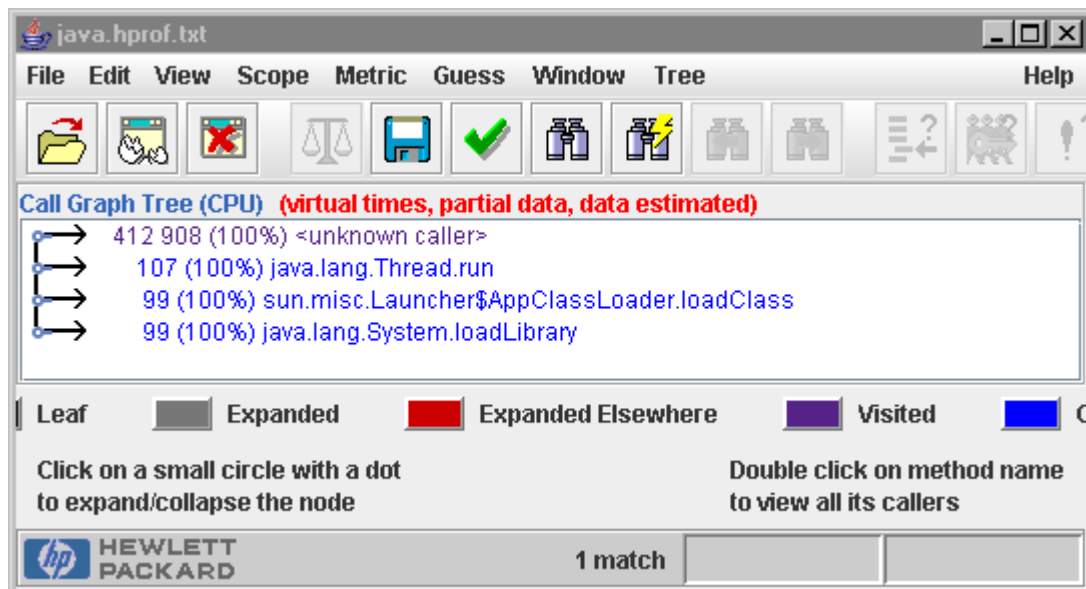


Figure 4. The Call Graph Tree for MemoryLeak.

The tree can be explored by opening and closing its sub-branches. Often the memory leak is located in the most frequently called method, which can be found by following the large call numbers down into the call graph. For example, the most frequently called method is located somewhere in the `<unknown caller>` branch of Figure 4.

I had a strong suspicion about the identity of the method behind the memory leak (the `AbstractStringBuilder` constructor), so I wanted to find it in the graph. This was easy

to do since I had previously marked the method in the allocation sites window (Figure 3).

A search for a marked method is triggered by pressing the 'lightning binoculars' icon in Figure 4, which opens up the call graph in a similar way to Figure 5.

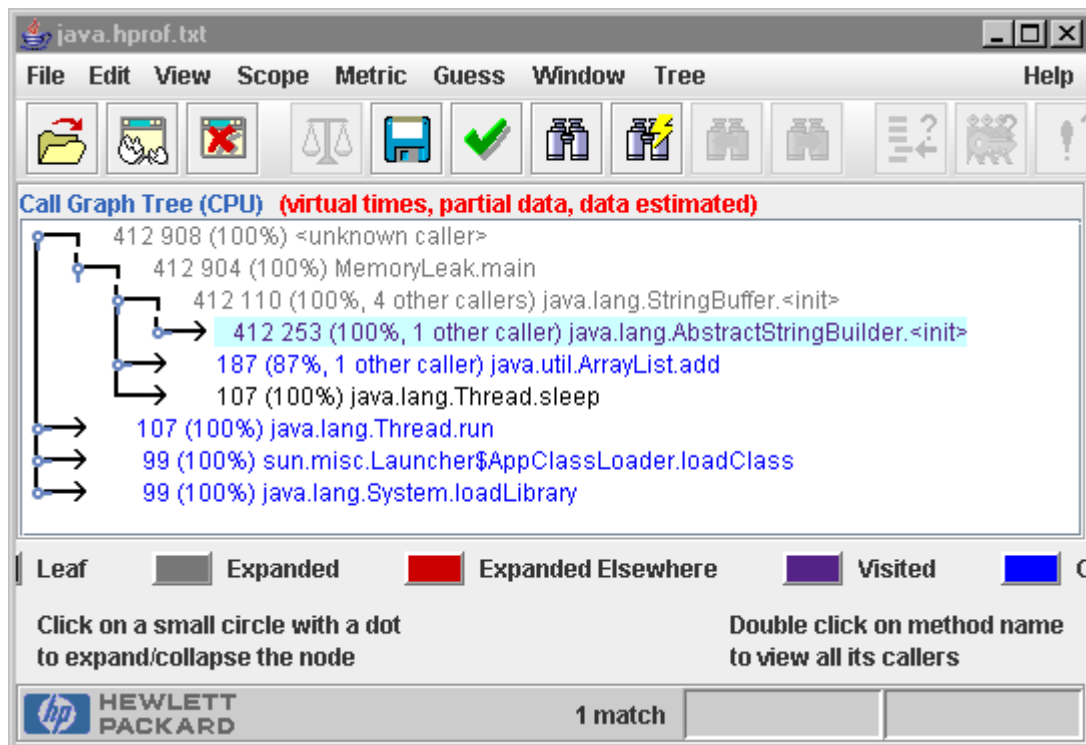


Figure 5. Calls to AbstractStringBuilder.

Figure 5 shows that the AbstractStringBuilder constructor is called to build the StringBuffer objects in main(). Also, the constructor is the consumer of most of the application's CPU time.

I can conclude that the memory leak is caused by the StringBuffer objects created in main(). I now have to examine the code and trace the usage of those objects. Some of them (110,000 out of 2.6 million) are not being dereferenced.

1.4. Removing the Memory Leak

The problem with the MemoryLeak class is easily fixed, by ensuring that the references to the StringBuffer objects in the ArrayList are periodically deleted:

```
if(i%8000 == 0) arrList.clear();
```

This line, nestled in main()'s for-loop, removes the list's references, which allows the garbage collector to reclaim the objects.

The outcome of this coding change can be judged by collecting garbage collection details once again, and reviewing them in HPjTune. Figure 6 shows the heap usage after collection against time graph for the revised MemoryLeak class:

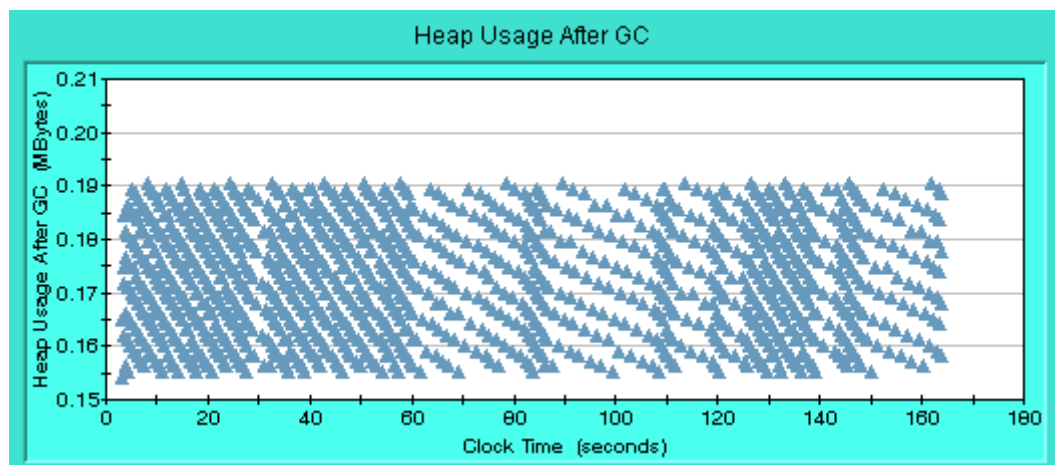


Figure 6. Heap Usage Against Time for Revised MemoryLeak.

The heap usage is flat over time, showing that the memory leak has been removed.

1.5. More Information on HPjTune

The HPjTune site (<http://www.hp.com/products1/unix/java/java2/hpjTune/>) is a good starting point for more documentation about the tool.

The IBM article, "Eye on Performance: Tuning Garbage Collection in the HotSpot JVM" by Jack Shirazi and Kirk Pepperdine (<http://www-128.ibm.com/developerworks/library/j-perf06304/>) describes a real-world use of HPjTune.

The Java Performance Tuning website has a long article on how to use HPjTune, with lots of screenshots (<http://www.javaperformancetuning.com/tools/hpjTune/>).

It's quite straightforward to understand the log format generated by `-Xloggc`. It's explained in a Java technical tip at

<http://java.sun.com/developer/JDCTechTips/2004/tt0420.html>. There are a few additional options that generate more detailed collection data, including `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamps`. How this additional information can be used to identify garbage collection problems is explained in <http://java.sun.com/docs/hotspot/gc1.4.2/example.html>.

1.6. More Information on Profiling and HPjmeter

There are several `-Xrunhprof` options that I haven't described, which are summarized in the online help:

```
java -Xrunhprof:help
```

The `-Xrunhprof` option is an interface to J2SE's HPROF profiling tool, which was much improved in J2SE 5.0 to provide more accurate and comprehensive data. A

good introduction to HPROF can be found in an article by Kelly O'Hair at <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>.

An informative article by Bill Pierce dates from 2001 (http://www.javaworld.com/javaworld/jw-12-2001/jw-1207-hprof_p.html). It looks at how HPROF data can highlight problems such as memory leaks, excessive CPU usage, and thread deadlocks. The version of HPROF he describes is quite old, but his techniques can be employed with the newer version of the tool.

Additional documentation on HPjmeter is available at its home site (<http://www.hp.com/products1/unix/java/hpjmeter/>). The Java Performance Tuning site has a long article on HPjmeter, with lots of screenshots (<http://www.javaperformancetuning.com/tools/hpjmeter/>).

If HPjmeter isn't quite what you're looking for, a good starting point for finding profilers is the Google directory http://www.google.com/Top/Computers/Programming/Languages/Java/Development_Tools/Performance_and_Testing/Profilers/, which currently lists over 20 different ones.

2. The JVM Heap

Before I start fine-tuning the JVM to minimize garbage collection pauses, I need to explain how the JVM structures the heap, and how the heap is garbage collected.

The heap is divided into three 'generations', as shown in Figure 7.

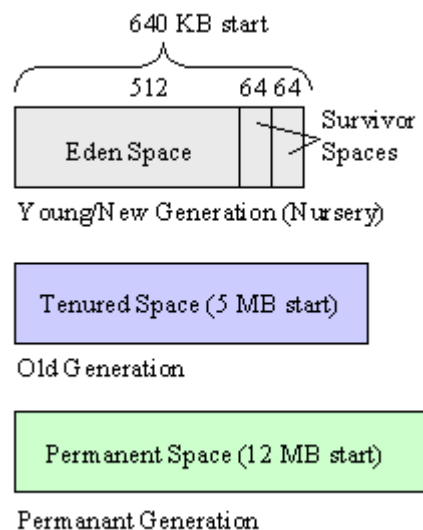


Figure 7. The JVM Heap Layout.

Figure 7 includes sizes for the memory spaces in the heap, which are the default starting values for the standard JVM on a desktop machine running Windows XP and J2SE 5.0. The values vary considerably between different JVMs, platforms, OSes, and J2SE versions.

The heap will typically grow during the application's execution since the JVM attempts to keep the total free space to stored objects ratio in a 40-70% range. If less than 40% of the heap is free, then the heap is expanded. The heap can also shrink, triggered by the free space percentage rising above 70%.

The young generation space (also known as the new space or nursery) is for newly created objects, which are expected to be short-lived. The young generation is divided into three parts: the eden space and two survivor spaces; new objects are initially placed in the eden space.

Garbage collection is applied to the eden space, and one of the survivor spaces, and any objects that aren't deleted are moved to the other survivor space. Each collection will move surviving objects from one survivor space to another, until they've been shuffled backwards and forwards a certain number of times. At that point, the objects are moved to the tenured space, the home for longer-lived objects.

The permanent space is for JVM objects representing classes and methods.

The garbage collection strategies for the young and old generation spaces are different. The young space uses an efficient copy-compaction collector, since it's called frequently due to the short lives of its objects. The old generation uses a mark-sweep-compaction algorithm that defragments the space as it traverses it. It's slower than the copy collector, but is called less frequently due to its objects' longevity, and the larger default size of the space.

If the JVM is called without any heap options, then the heap space can increase to a maximum of 64MB. The young space can grow to at most 5 MB, and the tenured and permanent spaces to at most 64 MB, if there's room.

The sizes of the each space can be adjusted with JVM command line options, as can the size ratios between the eden and survivor spaces, and between the old and young spaces.

3. Reducing Garbage Collection Pause Times

One of the most irritating things while playing a game is for it to suddenly slow down (or even freeze) while the JVM carries out a lengthy garbage collection. Garbage collection is a necessary evil, but it shouldn't impact game play. Collections should be fast enough to be invisible to the player.

The emphasis in this appendix is on techniques for reducing the duration of each garbage collection, but there are other ways of judging the success of a collection strategy.

- *Throughput.* Throughput is the percentage of execution time not spent in a garbage collection over a long period of time. Techniques for shortening the pause time for each collection often lead to an increase in the number of collections. This can raise the total collection time, thereby reducing the throughput.
- *Pause predictability.* Are garbage collection pauses scheduled at times that are convenient for the program (e.g. as the user moves to a new level in the game)?
- *CPU Usage.* What percentage of the total available CPU time is spent in garbage collection?

- *Memory Footprint*. Can the size of the heap used by the program be constrained?
- *Memory Reuse*. How quickly after an object is dereferenced, is its memory available for a new object?

Basic Techniques for Reducing Pause Time

Many performance issues, including collection pause time, may be solved by simply moving to the newest J2SE. JVM performance has improved considerably over the years. The next J2SE (version 6, code-named Mustang) promises further substantial speed gains.

A general technique (which will be shown in action in the following sections) is to reduce the sizes of the heap's spaces. This allows the collectors to finish sooner, so reducing the pause time. However, the total number of collections may increase.

A cardinal rule is to check your work! Any tweaking of JVM options should be followed by garbage collection log analysis (with HPj tune), to see if the pause times really have shrunk, and how other attributes, such as total heap size and throughput, have been affected.

It's important to collect log data that covers a significant period of time (e.g. ten of minutes, or hours, of game play). Longer runs give a more realistic view of the performance issues, and highlight problems, such as memory leaks, that may not be visible when the game first starts. Log data should be gathered from multiple runs, on a variety of JVMs, OSes, and machines.

In the simple examples below, I've analysed log data that only cover 1-2 minutes of execution time, so the results must be viewed with some skepticism. Nevertheless, the techniques I'll be utilizing can reduce pause times in real-size applications.

4. Reducing Pauses in JumpingJack

JumpingJack is a 2D side-scroller developed back in chapter 12. Figure 8 shows a screenshot.



Figure 8. JumpingJack in Action.

I played JumpingJack for 50 seconds, after starting it with garbage collection logging switched on:

```
java -Xloggc:gc.txt JumpingJack
```

The heap usage graph in HPjmeter appears in Figure 9.

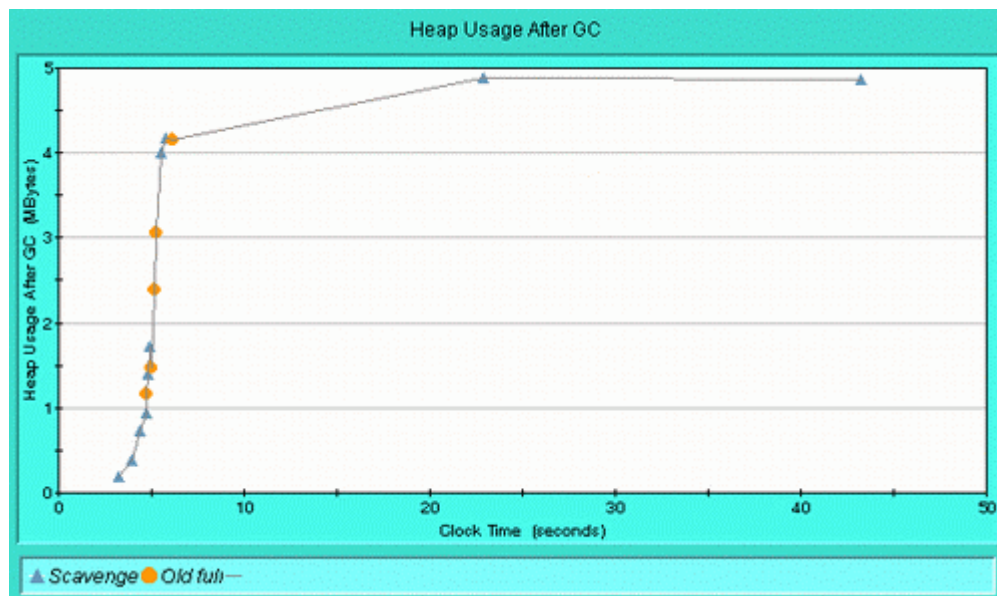


Figure 9. Heap Usage for JumpingJack.

The triangles show when the young generation's copy-compaction collector was executed, and are labeled "Scavenge" in the legend. The circles indicate when the old generation's mark-sweep-compaction collector was in action.

Numerous collections are carried out in the young generation space at start-up time, but the interval between the collections start to lengthen as execution progresses. During that time, heap usage increases steeply, but flattens out at around 5 MB.

This pattern of heap usage and garbage collection is very typical. The start size for the young generation is too small (640 KB), triggering garbage collections and space enlargement, until there's enough room for the required objects.

More details can be gleaned from HPjTune's "Summary" window, which is shown in Figure 10.

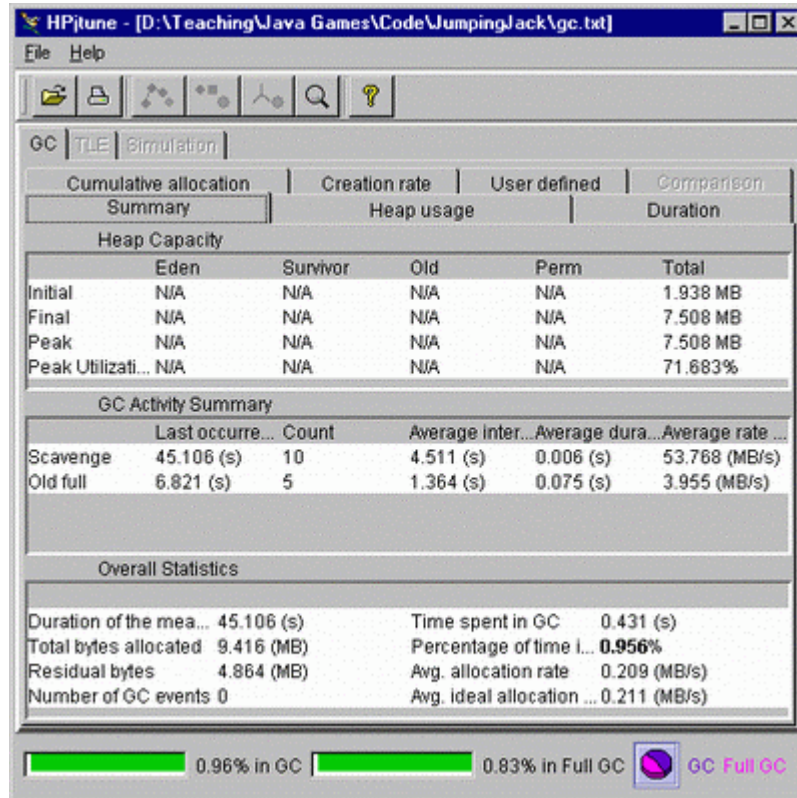


Figure 10. HPjTune Summary for JumpingJack.

The total size of the heap starts at 1.938 MB and increases to 7.508 MB, confirming the growth pattern seen in Figure 9. (Unfortunately, a size breakdown by spaces isn't available.) The starting size contradicts the numbers I gave in Figure 7, but this kind of discrepancy is to be expected between different platforms, OSes, and J2SE versions.

There were 10 young generation (scavenge) collections, taking 6 ms each on average. The five old generation collections took an average 75 ms each, which illustrates that old generation collection is usually a lot slower than young collection (a factor of 10 slower is common).

The percentage of time spent garbage collecting compared to the total running time is a miniscule 1%. This is a strong indicator that I don't need to make any adjustments to the heap. In general, it's only when garbage collection rises to a 10-15% share of execution time that it becomes a concern.

Nevertheless, I'll adjust a few options, to show the kinds of 'tweaking' that are useful in real-sized programs. My primary aim is to reduce garbage collection pause times.

4.1. Adjusting the Young Generation

The (relatively) large number of young generation collections at start-up time suggests that the young generation space is initially too small.

The lack of space will cause objects to be moved to the old space, triggering collections there. Collections of the old generation are slower, and should be avoided.

If I increase the initial size of the young generation, there will be less collections, and less likelihood of object migrating to the old generation. The collection that do occur will employ the copy collector, and so be quicker than the old generation's mark-sweep-compact collector.

I adjust the heap sizes with the following settings:

```
java -Xloggc:gc.txt -Xms30m -Xmx30m
      -XX:NewSize=6m -XX:MaxNewSize=6m
      JumpingJack
```

The `-Xms` and `-Xmx` options set the total size of the heap to start (and remain) at 30 MB. The `-XX:NewSize` and `-XX:MaxNewSize` options fix the young generation at 6 MB. In general, setting a space's start and maximum sizes to be the same helps the collectors, and may improve the JVM's startup time.

The young generation space is set to be 6 MB, a little higher than the maximum heap usage shown in Figure 9 (around 5 MB).

I collected garbage collection statistics using `-Xloggc` once again, and examined them with `HPjtime`. The data showed mixed improvements: during a 45 second run there were only 2 copy collections from the young generation, each taking an average 32 ms, and no garbage collections from the old space. This compares with 10 copy collections, and 5 collections from the old space in the original call. However, the copy collections only took an average of 6 ms each in the original run.

4.2. Space Size Issues for Heap Spaces

There are a few issues to consider when adjusting the young generation and total heap sizes.

The New Ratio

If the young generation space is increased, it shouldn't be made larger than the old generation. The old space must be large enough to hold all the data that might be moved to it from the young space.

In general, a good ratio of old to young is 2:1, which can be specified with the `-XX:NewRatio` option (i.e. `-XX:NewRatio=2`). This means that the old space will be double the size of the young space.

It's a good idea to do a rough calculation of the old:young ratio to ensure that it hasn't dropped below 1:1. In the example above, the 30 MB heap allocation includes room for the permanent, old, and young spaces. Even if the permanent space takes 12 MB (unlikely), it still leaves 18 MB for the old and young. The young generation space is set to be 6 MB, leaving 12 MB for the old generation space, which is a 2:1 ratio.

The Survivor Ratio

The eden to survivor space ratio is set to 8 by default in Windows. For example, Figure 7 shows that the young generation space (640 KB) is divided into 512 KB for eden, and 64 KB each for the two survivor spaces. The eden:survivor ratio is 512:64, or 8. This ratio will be maintained as the young generation grows (and shrinks).

If the survivor space is increased, it'll take longer to fill up, which will delay the movement of objects from the young generation to the old generation. This may reduce the average collection pause time, since copy compaction in the young generation is faster than mark-sweep-compaction in the old generation.

The survivor space is adjusted indirectly by setting the `-XX:SurvivorRatio` option; for instance, `-XX:SurvivorRatio=4` divides the 640 KB of the young generation into approximately 428 KB for the eden space and 107 KB for each survivor space. This has made the survivor spaces bigger (64 to 107 KB), at the expense of the eden space.

Total Heap Size

Reducing the heap size is a good option for reducing the period of each garbage collection, since less space will need to be scanned. However, there will tend to be more collections, and so more pauses. Also, if the total heap is made too small, then the JVM may start utilizing virtual memory, with the associated overheads of paging objects in and out of RAM. Paging affects the efficiency of the collectors, making the pause duration lengthen.

In most real-world applications, the programmer will want to set the total heap size to a much larger value than the default 64 MB. The temptation is to set it equal to the total amount of RAM on the machine, which would be a mistake. RAM is needed by the OS, other applications, and by non-heap parts of the JVM. The result will be a memory-grabbing competition between the JVM, the OS, and other applications, solved by the use of virtual memory and paging, at the expense of performance.

The memory needs of the OS and other applications can be discovered by using Window's Task Manager (accessible via a right-click in the Task Bar in Windows NT/2000/XP/2003). Task Manager will also report the requirements of the running JVM, which will be greater than the total heap space reported by HPjune. Memory is typically needed for the C portions of the JVM, for DLLs (e.g. OpenGL), and for data passed to those DLLs (such as textures in OpenGL).

The maximum heap size should be set to a value which leaves enough memory for the OS, other applications, and for the non-heap needs of the JVM.

There are numerous alternatives to Window's Task Manager; one that I've used is FreeMeter (<http://www.tiler.com/freemeter/>).

5. Reducing Pauses in FPShooter3D

FPShooter3D is a simple 3D FPS, described in chapter 24; figure 11 shows a screenshot.



Figure 11. FPShooter3D in Action.

I started the application with garbage collection logging switched on:

```
java -cp %CLASSPATH%;nca\portfolio.jar -Xloggc:gc.txt FPShooter3D
```

The `-cp` option includes the NCSA Portfolio package for loading the 3D robot model. After a minute, I loaded the contents of `gc.txt` into HPj tune. The heap usage after collections graph is shown in Figure 12.

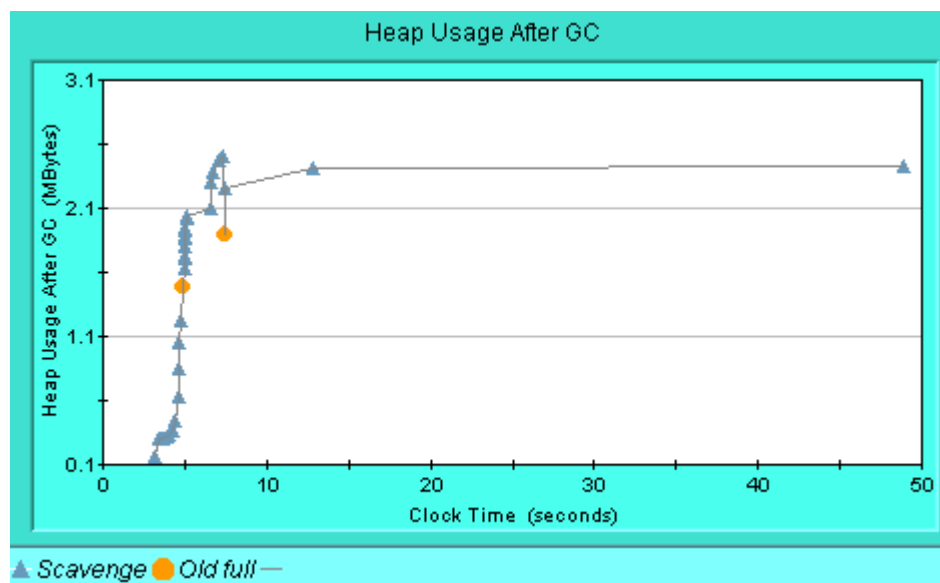


Figure 12. Heap Usage for FPShooter3D.

As with JumpingJack, execution begins with numerous young generation collections, which become less frequent as heap usage reaches around 2.5 MB.

HPjTune's "Summary" window gives more details (Figure 13).

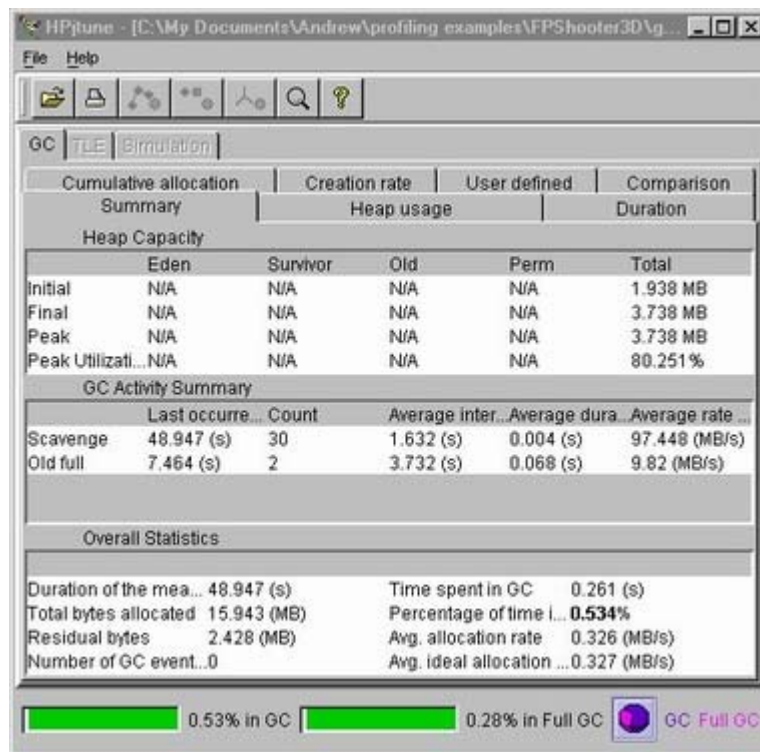


Figure 13. HPjTune Summary for FPSShooter3D.

The heap starts at 1.938 MB and increases to 3.738 MB. There were 30 young generation collections, taking 4 ms each on average. The two old space collections took an average of 68 ms each.

The percentage of time spent garbage collecting was a small 0.5%, which isn't really worth trying to improve. But as with JumpingJack, I'll adjust the total heap space, and the size of the young generation to try to reduce the pauses:

```
java -cp %CLASSPATH%;ncsa\portfolio.jar -Xloggc:gc.txt
-Xms24m -Xmx24m
-XX:NewSize=3m -XX:MaxNewSize=3m
FPSShooter3D
```

The new space is set to be 3 MB, to be a little higher than the maximum heap usage shown in Figure 12 (around 2.5 MB).

The 24 MB heap allocation includes the permanent, old, and young spaces. Even if the permanent space takes 12 MB, it still leaves 12 MB for the old and young. The young space is set to be 3 MB, leaving 9 MB for the old generation, which is a 3:1 ratio.

The garbage collection statistics for this configuration were collected in gc.txt, and examined with HPjTune. The number of pauses went down: during a 68 second run there were only 6 copy collections from the young space, each taking an average 12 ms. This compares with 30 copy collections, and 2 collections from the old space in the original call. However, the average pause time for those copy collections were 4 ms each.

5.1. Using Other Collectors

Aside from the two default collectors (the copy-compaction collector for the young generation, and the mark-sweep-compaction collector for the old generation), there are four other collectors available:

- an incremental collector (also known as the train collector);
- a parallel copying collector (also known as the throughput collector);
- a parallel scavenging collector;
- a concurrent collector.

The incremental, parallel copying, and parallel scavenging collectors are for the young generation, and could replace the default copy-compaction collector.

The incremental collector is sometimes recommended for reducing pause times. It does this by pausing more often, with each pause being short. This usually has an adverse effect on overall execution time. The incremental collector is being phased out, and will probably not be supported in J2SE 6.

A better choice is the parallel copying collector, which has similar characteristics to copy-compaction, but can make use of multiple CPUs. It performs better than copy-compaction, even on a single processor (as you'll see below).

The parallel scavenger is similar to the copy-compaction collector, but designed for machines with plenty of RAM (12 to 80 GB), and 8 or more CPUs. It's intended for use with the server JVM rather than the standard one. (I'll say more about the server JVMs below.)

The concurrent collector can replace the default mark-sweep-compaction collector in the old generation space, but is intended to utilize multiple CPUs.

Using the Parallel Copying Collector

In general, it's not worth changing the default collectors unless you have multiple CPUs on your machine. Nevertheless, let's see what happens when FPSooter3D is garbage collected with the parallel copying collector:

```
java -cp %CLASSPATH%;ncsa\portfolio.jar  
-XX:+UseParallelGC -Xloggc:gc.txt  
FPSooter3D
```

I've reverted to default values for the total heap size and young generation.

The heap usage graph is shown in Figure 14:

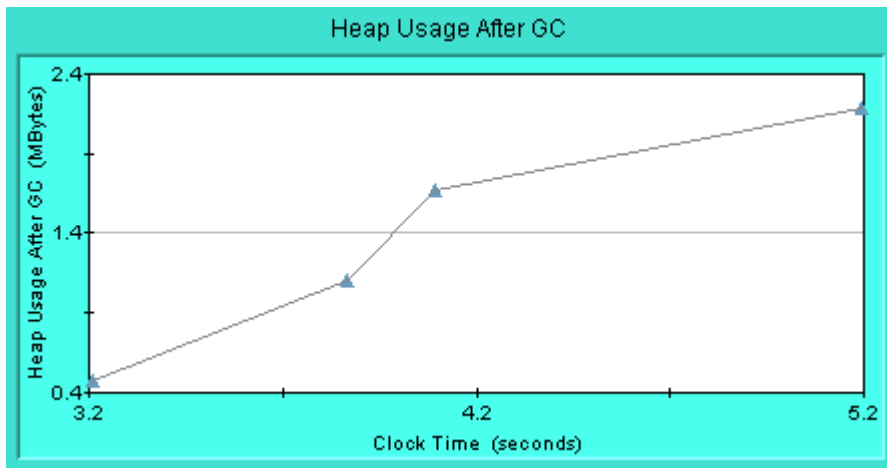


Figure 14. Heap Usage for FPS shooter3D with the Parallel Copy Collector.

The graph is a misleading since the application was executed for around 2 minutes, but there were no garbage collections after the first 5.2 seconds.

The "Summary" window is shown in Figure 15.

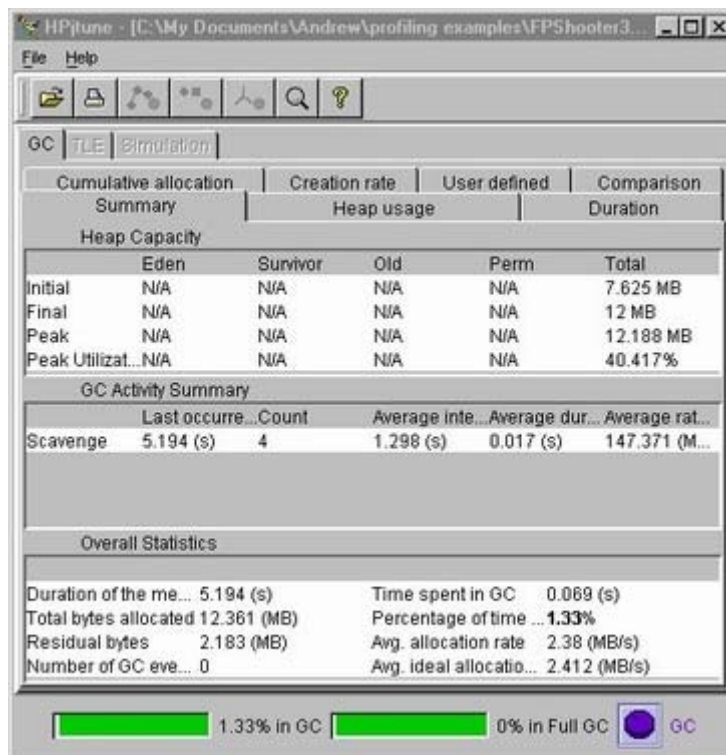


Figure 15. HPjTune Summary for FPS shooter3D with the Parallel Copy Collector.

The heap starts at 7.625 MB and grows to 12.188 MB, compared to 1.938 MB increasing to 3.738 MB with the copy collector. There were only 4 young generation collections, taking an average of 17 ms each. With the copy collector, there were 30 young space collections, taking 4 ms each on average, and two old space collections.

Note that the percentage of time spent garbage collecting (1.33%) is inaccurate since HPjtune bases its calculation on the time of the last garbage collection rather than the running time of the application.

5.2. J2SE 5 Ergonomics

J2SE 5.0 introduced two 'ergonomics', which allow the programmer to suggest preferred pause time and throughput behaviour for the JVM. The `-XX:MaxGCPauseMillis=<time>` option requests that the garbage collector keeps pauses to `<time>` ms or less. This may increase the number of collections, and thereby reduce the application's throughput.

The `-XX:GCTimeRatio=<ratio>` option specifies the maximum amount of time spent garbage collecting as a percentage of the total execution time. The percentage is calculated using $1 / (1 + \text{<ratio>})$. For example, for `-XX:GCTimeRatio=19`, the application will spend at most 5% of its time collecting garbage ($1/(1+19)$).

I tried the parallel copy collector again, and requested that the pause time be 10ms or less:

```
java -cp %CLASSPATH%;ncsa\portfolio.jar
      -XX:+UseParallelGC -XX:MaxGCPauseMillis=10
      -Xloggc:gc.txt
      FPShooter3D
```

The parallel copy collector on its own generates 4 new space collections, taking an average of 17 ms each (see Figure 15). With the additional pause request, this increases to 6 collections, but they only take an average of 2 ms each. This is a very nice improvement.

5.3. Calling the Garbage Collector

C++/C programmers often latch onto the `System.gc()` method as a way of 'fixing' the JVM. In fact, `System.gc()` is only a request, and is likely to trigger a 'full' garbage collection (a collection of both the young and old generations), even when such a detailed scan isn't needed.

A much better way of influencing garbage collection is to dereference objects when they are no longer needed, and to modify JVM settings.

It's possible to disable `System.gc()` by adding the `-XX:+DisableExplicitGC` option to the java command line.

5.4. Using the Server JVM

Sometimes the server JVM is suggested as a way of speeding up code 'for free'; some people claim a 30% boost to an application's performance by switching from the client JVM to the server version.

The server and client JVMs are essentially two different compilers linked to the same JVM. Also, depending on the compiler, the JVM is started with different heap options, and a different garbage collector. The underlying assumptions are that a 'server' machine has multiple CPUs, and at least 2 GB of spare RAM (in addition to

that used by the OS). The parallel scavenging collector is used on the young generation space, and much more heap is allocated at start-up time.

However, the server JVM is probably not suitable for games applications. The client JVM has been optimized for a typical desktop machine, so starts faster, and uses less memory than the server JVM. Generally, the client JVM is better for interactive applications with GUIs.

Another issue is that the server JVM is not part of the standard JRE download for Windows (although it is included with the JDK), so installing it on a user's machine will require a larger download.

The next version of Java (J2SE 6, code-named Mustang) promises significant performance boosts for the client JVM, which will put it on a par with J2SE 5's server JVM. Tests with alpha versions of Mustang (available from <https://mustang.dev.java.net/>) indicate that its client JVM is 58% faster than the one in J2SE 5.0. A lot of Java features important to gaming will be faster, including parts of Java 2D, Swing, and many geometry-related methods.

6. More Information on Garbage Collection

Garbage collection in J2SE 5.0 is explained in "Tuning Garbage Collection with the 5.0 Java Virtual Machine" at http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html. The <http://java.sun.com/docs/hotspot/> site contains links to papers on garbage collecting in J2SE 1.4.2 and 1.3.1, and J2SE 5.0 ergonomics.

One of the best overview of garbage collection and other performance issues in J2SE and J2EE is the Sun Tech Day presentation "J2SE and J2EE Performance: Learn How to Write High Performance Java Applications" by Rima Patel Sriganesh. The PDF version of her talk can be found at <http://www.javapassion.com/j2ee/javaperformance.pdf>.

A book with lots of garbage collection tips, and other performance advice:

- *Java Performance Tuning*
Jack Shirazi, 2nd ed., January 2003
<http://www.oreilly.com/catalog/javapt2/>

The support site for Shirazi's book eventually grew to become <http://www.javaperformancetuning.com/>. Sun's performance website is <http://java.sun.com/blueprints/performance/>, which collects many useful links, including a FAQ and online book.

JavaGaming.org has an active Performance Tuning forum (<http://www.javagaming.org/forums/index.php?board=20.0>), which answered several of my questions when I was writing this appendix.

There are numerous garbage collection options for the JVM that I haven't mentioned; a good list for the Sun and IBM JVM's can be found at <http://www.tagtraum.com/gcviewer-vmflags.html>. The page is part of the GCViewer website, a freeware garbage collection visualizer with similar features to HPj tune.

A very lengthy list of *all* the options for Sun's JVM is given at <http://java.sun.com/docs/hotspot/VMOptions.html>. Another incredibly detailed list by Joseph D. Mocker is at <http://blogs.sun.com/roller/resources/watt/jvm-options-list.html>.

7. Collecting Your Own Statistics

Although the information generated for the garbage collection log (-Xloggc) and the profile log (-Xrunhprof) is very complete, there may come a time when you need more specialized statistics, focussed on particular aspects of your application. For example, you may want to monitor the pattern of method calls in a specific class. It's possible to extract this kind of data from the profile log, but it may be simpler to collect the data with specially written code inside the application.

Two useful methods for obtaining heap sizes are: `Runtime.totalMemory()` and `Runtime.freeMemory()` which return the total amount of heap space and the amount that is currently free. The typical way of using these methods is to obtain a `Runtime` object at start-up, and then call these methods periodically:

```
Runtime runtime = Runtime.getRuntime(); // global

// repeatedly call...
long currHeapSize = runtime.totalMemory();
long currFreeSize = runtime.freeMemory();
```

In the 2D games developed in the first part of *Killer Game Programming in Java*, a good place to put statistics gathering is in the threaded canvas' animation loop; see chapter 2 for an explanation of `GamePanel` and its `run()` method.

If you want to generate graphs like those in the visualization tools, then you may want to employ a good charting library, such as `JFreeChart` (<http://www.jfree.org/jfreechart/>).

7.1. Collecting Statistics in 3D Games

For 3D games, the periodic collection of data should be embedded in a `Behavior` object which is triggered on a regular basis. As an example, I'll describe a behaviour class, `FPSBehaviour`, which collects frame rate numbers, and reports the current and average FPS values. This behaviour can be added to any 3D application, but I'll focus on adding it to the `Checkers3D` example from chapter 15.

The execution of the modified `Checkers3D` produces output like the following:

```
C>java Checkers3D
Frame: 28; 2s; FPS: 14.1; AFPS: 14.1
Frame: 122; 3s; FPS: 92.2; AFPS: 53.1
Frame: 188; 4.1s; FPS: 62.3; AFPS: 56.2
Frame: 263; 5.1s; FPS: 73.2; AFPS: 60.4
Frame: 346; 6.2s; FPS: 78.7; AFPS: 64.1
Frame: 415; 7.2s; FPS: 65.5; AFPS: 64.3
Frame: 489; 8.2s; FPS: 71.6; AFPS: 65.4
Frame: 515; 9.3s; FPS: 25.1; AFPS: 60.3
No frames
```

```

No frames
No frames
Frame: 578; 13.5s; FPS: 59.2; AFPS: 60.2
Frame: 656; 14.5s; FPS: 76.8; AFPS: 61.9
Frame: 715; 15.5s; FPS: 56; AFPS: 61.3
C>

```

The statistics are gathered roughly every second, and the current FPS and the ongoing average FPS (AFPS) are reported.

Java 3D will automatically stop rendering a 3D scene if nothing is changing in it. Therefore, if Checkers3D is left unattended for a while, it will start to output "No Frame" messages to indicate that no new frames have been generated since the last call to FPSBehaviour.

The FPSBehaviour constructor configures the wakeup condition, and initializes the timers.

```

// globals
private static final int DELAY = 1000;    // 1 second

private WakeupCondition wakeUpCond;
private long beforeTime, gameStartTime;
private long prevFrameNo = 0;

private double totalFPS = 0.0;
private int count = 0;

private DecimalFormat df; // for reporting results

public FPSBehaviour()
{
    wakeUpCond = new WakeupOnElapsedTime(DELAY);
    beforeTime = J3DTimer.getValue();
    gameStartTime = beforeTime;

    df = new DecimalFormat("0.#"); // 1 dp
} // end of FPSBehaviour()

```

The wakeup condition is time-based: the behaviour will fire roughly every DELAY milliseconds. I've set DELAY to 1000 ms (1 second), which is a reasonable frequency which doesn't cause the behaviour to be called too often. Overly frequent calls to the behaviour will degrade the performance of the application, reducing the frame rate.

The behavior starts when Java 3D calls initialize():

```

public void initialize()
{ wakeupOn(wakeUpCond); }

```

processStimulus() is called every DELAY ms:

```

public void processStimulus(Enumeration criteria)
{
    long timeNow = J3DTimer.getValue();
    long timeDiff = timeNow - beforeTime;
    double timeSpentInGame =

```

```

        ((double)(timeNow - gameStartTime)/1000000000L); // ns --> secs

    if (timeDiff == 0)
        System.out.println("FPS: 0 time difference");
    else { // timeDiff > 0
        long frameNo = getView().getFrameNumber();
        long numFrames = frameNo - prevFrameNo;

        if (numFrames == 0)
            System.out.println("No frames");
        else {
            double fps = ((double)numFrames * 1000000000L)/timeDiff;
            totalFPS += fps;
            count++;

            System.out.println("Frame: " + frameNo +
                               "; " + df.format(timeSpentInGame) + "s" +
                               "; FPS: " + df.format(fps) +
                               "; AFPS: " + df.format(totalFPS/count) );

            prevFrameNo = frameNo;
        }
    }
    beforeTime = J3DTimer.getValue();
    wakeupOn(wakeUpCond); // make sure we are notified again
} // end of processStimulus()

```

processStimulus() ignores its criteria argument, assuming that it's called when the DELAY interval has passed.

J3DTimer.getValue() is used to calculate the passage of time. I don't rely on the DELAY value since Behaviour uses System.currentTimeMillis() internally, which is somewhat inaccurate on different platforms (see chapter 2 for a discussion). Also, it takes a few milliseconds for a behaviour to start executing after a wakeup condition becomes true, so it's more accurate to measure the time within processStimulus().

An important line is:

```
long frameNo = getView().getFrameNumber();
```

Behaviour.getView() returns the View object attached to the scene graph's view branch, where the scene is rendered. View.getFrameNumber() returns the number of generated frames since the application began.

A common mistake with Behavior code is to forget to re-register the behaviour at the end of processStimulus():

```
wakeupOn(wakeUpCond);
```

This ensures that the behaviour will be called again after a further DELAY ms.

The behaviour can be attached anywhere in the application's scene graph. In WrapCheckers3D, I connect the behaviour to the top-level branch group, sceneBG:

```

private void collectStats()
{
    FPSBehaviour fpsBeh = new FPSBehaviour();
    fpsBeh.setSchedulingBounds( bounds );
    sceneBG.addChild(fpsBeh); // attach to the scene
}

```



```
}
```

Another easy-to-forget coding element is to set the scheduling bounds of the behaviour with `setSchedulingBounds()`. If no bounds are specified then the behaviour will never execute.

`collectStats()` is called when the scene graph is being constructed in `createSceneGraph()` in `WrapCheckers3D`:

```
private void createSceneGraph()
// initialize the scene
{
    sceneBG = new BranchGroup();
    bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

    lightScene();           // add the lights
    addBackground();       // add the sky
    sceneBG.addChild( new CheckerFloor().getBG() ); // add the floor

    collectStats();

    floatingSphere();      // add the floating sphere

    sceneBG.compile();     // fix the scene
} // end of createSceneGraph()
```

Although `FPSBehaviour` only collects and reports FPS data, it can easily be extended to gather heap information by calling `Runtime.totalMemory()` and `Runtime.freeMemory()`.

Other Frame Rate Behaviours

An alternative way of writing a frame rate counter is illustrated by the Java 3D demo, `FPSCounter` (an example in the Java 3D distribution). It uses a frame-based wakeup condition:

```
wakeUpCond = new WakeupOnElapsedFrames(0);
```

This triggers the behaviour in every frame, so is quite resource intensive. It's a good idea to change the integer argument to `WakeupOnElapsedFrames()` to reduce the calling frequency. For instance, `WakeupOnElapsedFrames(10)` will fire the behaviour in every 10th frame.

A potential problem is that if the system stops rendering frames (since the scene is not changing), then `FPSCounter` won't be called.

7.2. Monitoring Applications with JConsole and MBeans

JConsole is a monitoring tool that comes with J2SE 5.0. It uses JMX (Java Management Extensions) to monitor the JVM and the running Java application. The basic configuration is illustrated by Figure 16.

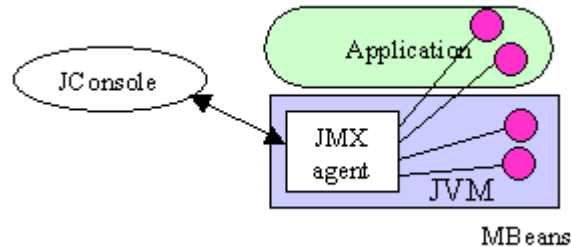


Figure 16. Using JConsole.

JConsole interacts with a JMX agent process inside the JVM, which passes monitoring requests onto MBeans embedded in the JVM and (optionally) in the application. An MBean (Managed Bean) has a Java bean-like interface which allows the JVM (or application) to be examined and controlled from JConsole.

The reason JConsole is mentioned here is because of its ability to monitor applications augmented with MBeans.

Any JMX client can communicate with the JMX agent, not just JConsole. The source for JConsole is available, and part of the aim of JConsole is to act as a medium-sized example of how to write a JMX client.

The JVM-related parts of JConsole give access to the generational spaces, the currently executing threads, the classes that are currently loaded, the compilation system, the garbage collectors, the runtime system, and the OS on which the JVM is running.

Figure 17 shows the MBeans tab of JConsole, with the right-hand window open on the copy collector for the young generation.

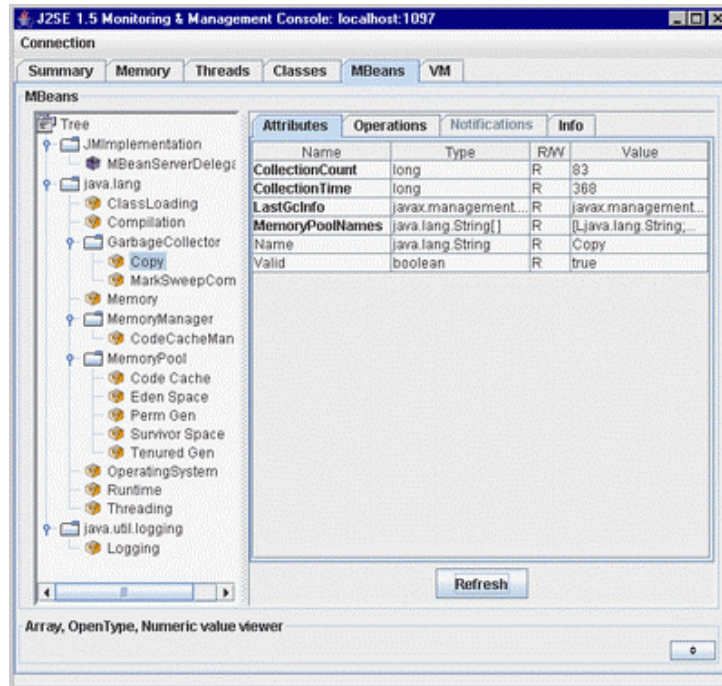


Figure 17. Copy Collector MBean Information.

The data includes the current collections count, and the total collection time.

Figure 18 shows a graph of the total heap usage for the application:

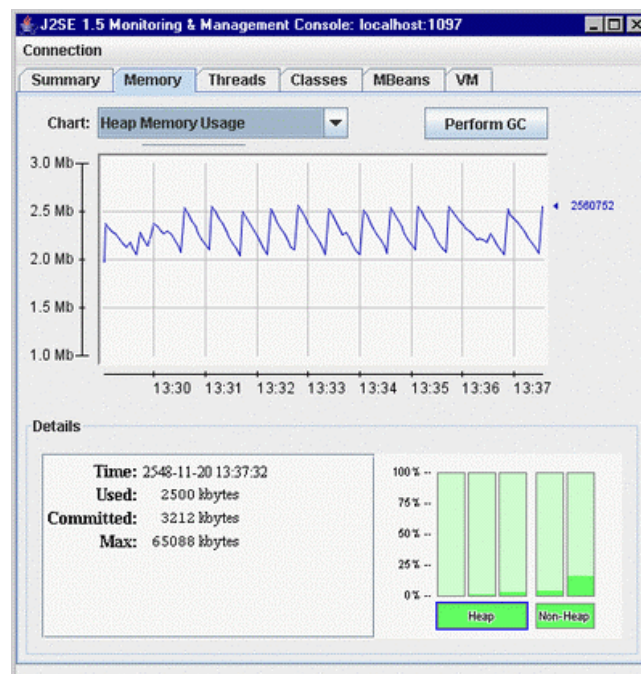


Figure 18. Heap Usage Displayed in JConsole.

Several other charts are available, showing the heap usage in each generation space.

Using JConsole

First the application must be started in a JVM with the JMX agent process switched on. One way of communicating with the agent is via a port address, which automatically comes with password authentication over SSL (Secure Sockets Layer). During testing, authentication and SSL can be switched off, with the command line:

```
java -Dcom.sun.management.jmxremote.port=1097
     -Dcom.sun.management.jmxremote.authenticate=false
     -Dcom.sun.management.jmxremote.ssl=false
     Checkers3D
```

I've set the JMX agent listening at port 1097.

The second stage, is to start JConsole, pointing it to port 1097 on the host:

```
jconsole localhost:1097
```

jconsole.exe is in the <JAVA_HOME>/bin directory.

More Information on JConsole

The J2SE 5.0 documentation for JConsole can be found at <http://java.sun.com/j2se/1.5.0/docs/guide/management/jconsole.html>. It includes a detailed explanation of each of its tabbed screens (with screenshots).

Two good how-to-use JConsole articles are:

- *Using JConsole to Monitor Applications*, Mandy Chung
<http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>
- *Monitoring Local and Remote Applications Using JMX 1.2 and JConsole*, Russell Miles, <http://www.onjava.com/pub/a/onjava/2004/09/29/tigerjmx.html>

These articles concentrate on using JConsole with standard applications. The next step is to augment the application with its own MBeans, which will appear alongside the JVM ones when JConsole is started. This technique is clearly explained in:

- *JConsole, the Essential J2SE Tool*, DJ Walker-Morgan,
<http://www.builder.au.com.au/program/0,39024614,39189623,00.htm>