Allocation Profilers

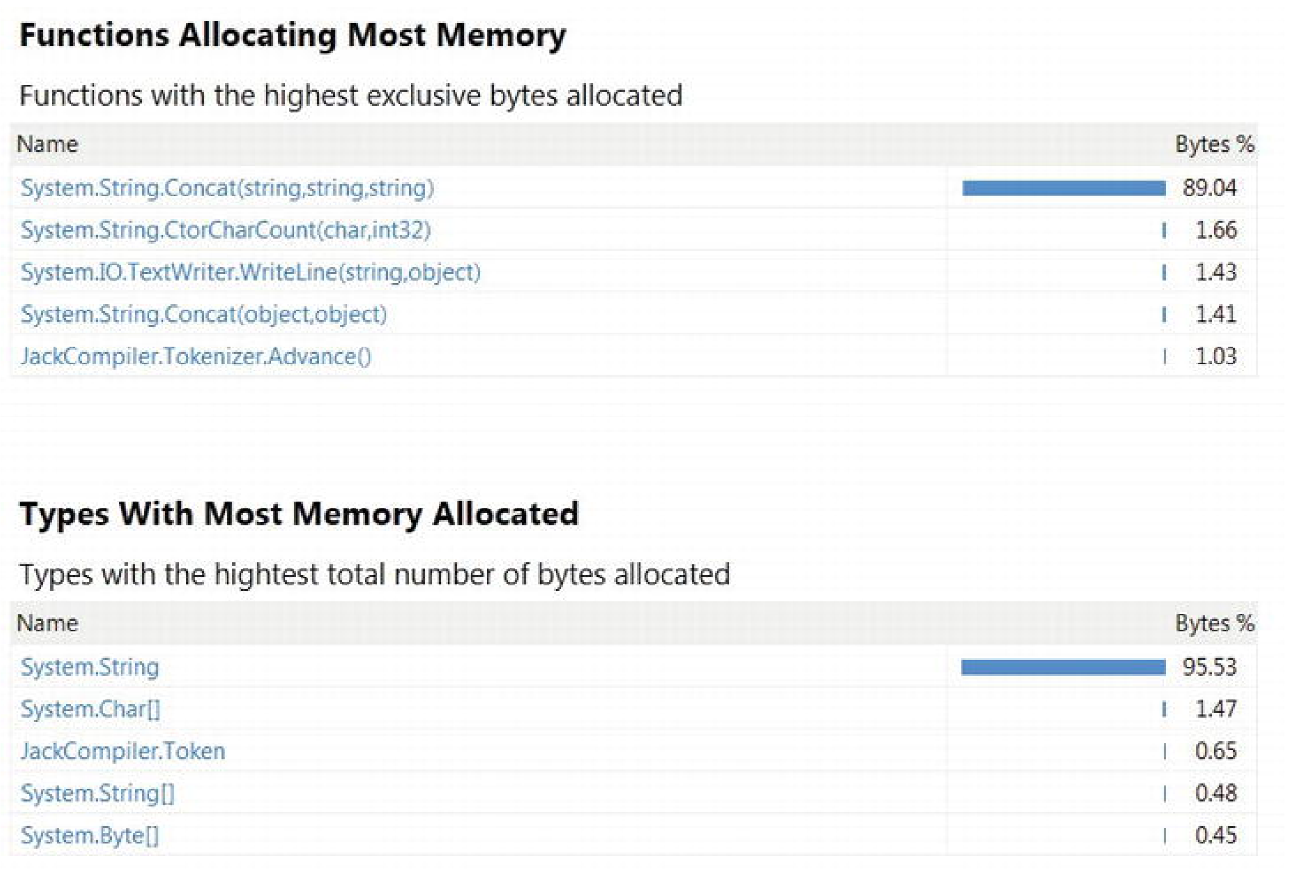
Allocation profilers detect memory allocations performed by an application and can report which methods allocated most memory, which types were allocated by each method, and similar memory-related statistics. Memory-intensive applications can often spend a significant amount of time in the garbage collector, reclaiming memory that was previously allocated. As we will see in Chapter 4, the CLR makes it very easy and inexpensive to allocate memory, but recovering it can be quite costly. Therefore, a group of small methods that allocate lots of memory may not take a considerable amount of CPU time to run—and will be almost invisible in a time profiler's report—but will cause a slowdown by inflicting garbage collections at nondeterministic points in the application's execution. We have seen production applications that were careless with memory allocations, and were able to improve their performance—sometimes by a factor of 10—by tuning their allocations and memory management.

We'll use two tools for profiling memory allocations—the ubiquitous Visual Studio profiler, which offers an allocation profiling mode, and CLR Profiler, which is a free stand-alone tool. Unfortunately, both tools often introduce a significant performance hit to memory-intensive applications, because every memory allocation must go through the profiler for record-keeping. Nonetheless, the results can be so valuable that even a 100× slowdown is worth the wait.

Visual Studio Allocation Profiler

The Visual Studio profiler can collect allocation information and object lifetime data (which objects were reclaimed by the garbage collector) in the sampling and instrumentation modes. When using this feature with sampling, the profiler collects allocation data from the entire process; with instrumentation, the profiler collects only data from instrumented modules.

You can follow along by running the Visual Studio profiler on the JackCompiler.exe sample application from this chapter's source code folder. Make sure to select ".NET memory allocation" in the Visual Studio Performance Wizard. At the end of the profiling process, the Summary view shows the functions allocating most memory and the types with most memory allocated (see Figure 2-16). The Functions view in the report contains for each method the number of objects and number of bytes that method allocated (inclusive and exclusive metrics are provided, as usual) and the Function Details view can provide caller and callee information, as well as color-highlighted source code with allocation information in the margins (see Figure 2-17). More interesting information is in the Allocation view, which shows which call trees are responsible for allocating specific types (see Figure 2-18).


Functions Allocating Most Memory

Functions with the highest exclusive bytes allocated

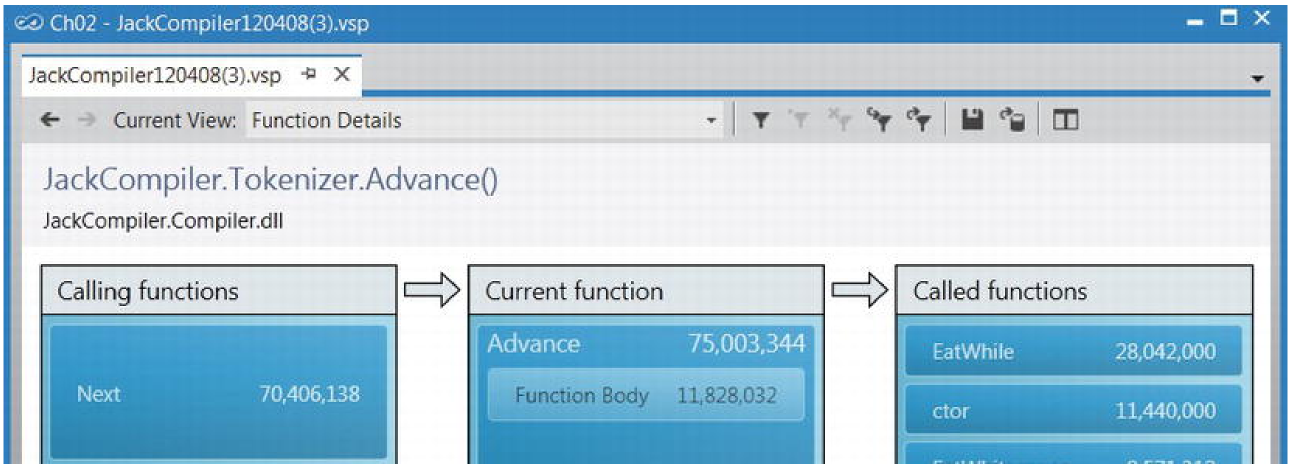
Name	Bytes %
System.String.Concat(string,string,string)	89.04
System.String.CtorCharCount(char,int32)	1.66
System.IO.TextWriter.WriteLine(string,object)	1.43
System.String.Concat(object,object)	1.41
JackCompiler.Tokenizer.Advance()	1.03

Types With Most Memory Allocated

Types with the hightest total number of bytes allocated

Name	Bytes %
System.String	95.53
System.Char[]	1.47
JackCompiler.Token	0.65
System.String[]	0.48
System.Byte[]	0.45

Figure 2-16. Summary view from allocation profiling results


Ch02 - JackCompiler120408(3).vsp

JackCompiler120408(3).vsp

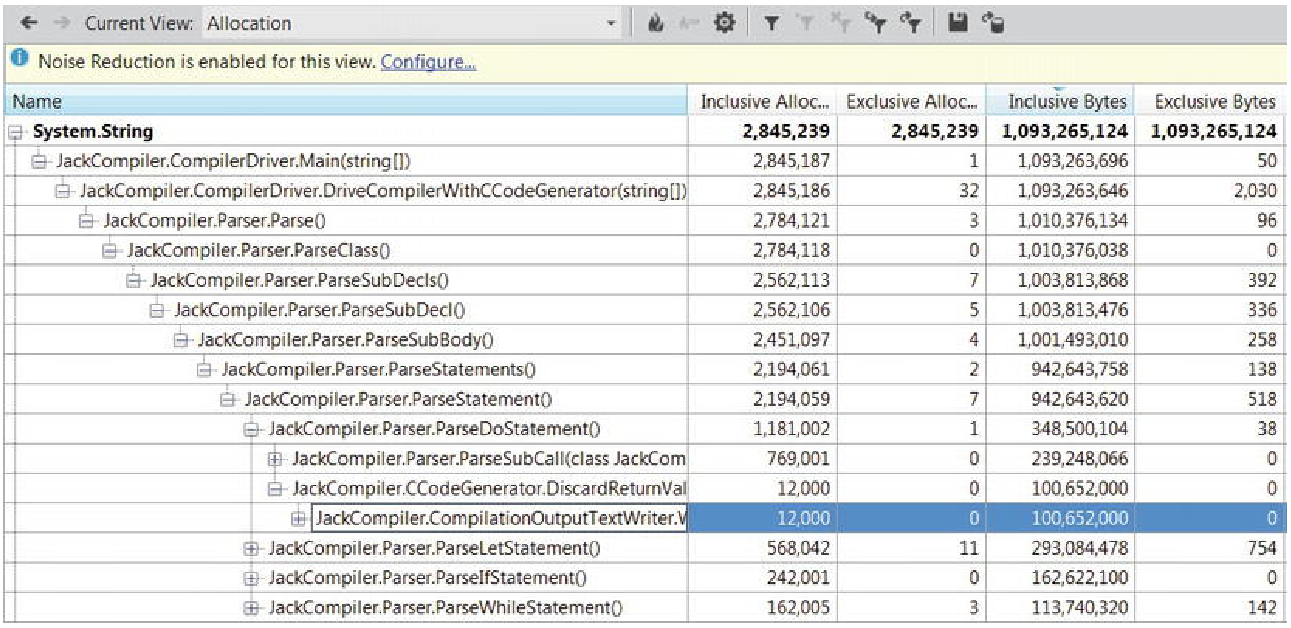
Current View: Function Details

JackCompiler.Tokenizer.Advance()

JackCompiler.Compiler.dll

Calling functions	Current function	Called functions
Next 70,406,138	Advance 75,003,344	EatWhile 28,042,000
	Function Body 11,828,032	ctor 11,440,000

Figure 2-17 . Function Details view for the JackCompiler.Tokenizer.Advance function, showing callers, callees, and the function's source with allocation counts in the margin

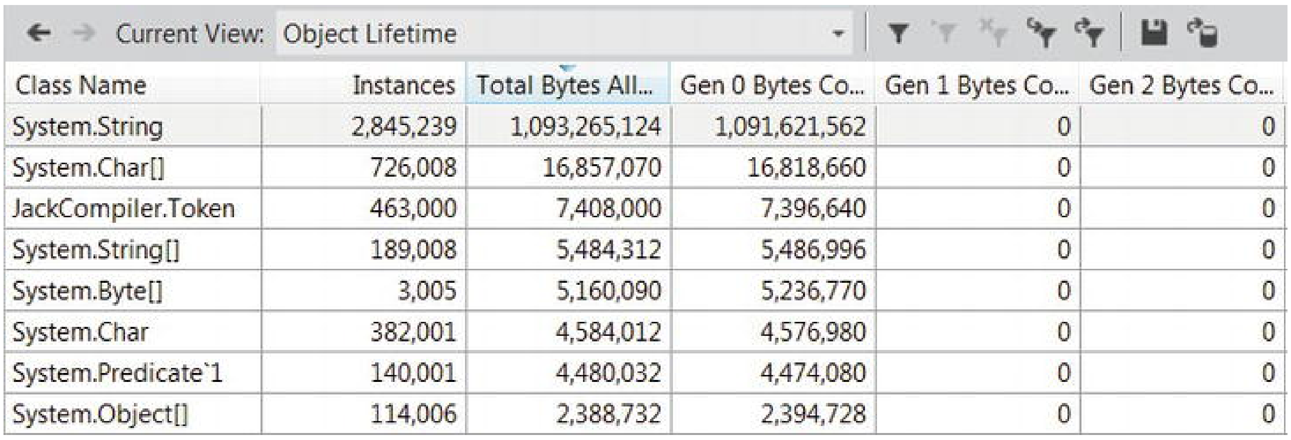

Current View: Allocation

Noise Reduction is enabled for this view. Configure...

Name	Inclusive Alloc...	Exclusive Alloc...	Inclusive Bytes	Exclusive Bytes
System.String	**2,845,239**	**2,845,239**	**1,093,265,124**	**1,093,265,124**
JackCompiler.CompilerDriver.Main(string[])	2,845,187	1	1,093,263,696	50
JackCompiler.CompilerDriver.DriveCompilerWithCCodeGenerator(string[])	2,845,186	32	1,093,263,646	2,030
JackCompiler.Parser.Parse()	2,784,121	3	1,010,376,134	96
JackCompiler.Parser.ParseClass()	2,784,118	0	1,010,376,038	0
JackCompiler.Parser.ParseSubDecls()	2,562,113	7	1,003,813,868	392
JackCompiler.Parser.ParseSubDecl()	2,562,106	5	1,003,813,476	336
JackCompiler.Parser.ParseSubBody()	2,451,097	4	1,001,493,010	258
JackCompiler.Parser.ParseStatements()	2,194,061	2	942,643,758	138
JackCompiler.Parser.ParseStatement()	2,194,059	7	942,643,620	518
JackCompiler.Parser.ParseDoStatement()	1,181,002	1	348,500,104	38
JackCompiler.Parser.ParseSubCall(class JackCom	769,001	0	239,248,066	0
JackCompiler.CCodeGenerator.DiscardReturnVal	12,000	0	100,652,000	0
JackCompiler.CompilationOutputTextWriter.V	12,000	0	100,652,000	0
JackCompiler.Parser.ParseLetStatement()	568,042	11	293,084,478	754
JackCompiler.Parser.ParseIfStatement()	242,001	0	162,622,100	0
JackCompiler.Parser.ParseWhileStatement()	162,005	3	113,740,320	142

Figure 2-18 . Allocation view, showing the call tree responsible for allocating System.String objects

In Chapter 4 we will learn to appreciate the importance of quickly discarding temporary objects, and discuss a critical performance-related phenomenon called *mid-life crisis*, which occurs when temporary objects survive too many garbage collections. To identify this phenomenon in an application, the Object Lifetime view in the profiler's report can indicate in which generation objects are being reclaimed, which helps understand whether they survive too many garbage collections. In Figure 2-19 you can see that all of the strings allocated by the application (more than 1GB of objects!) have been reclaimed in generation 0, which means they didn't survive even a single garbage collection.


Current View: Object Lifetime

Class Name	Instances	Total Bytes All...	Gen 0 Bytes Co...	Gen 1 Bytes Co...	Gen 2 Bytes Co...
System.String	2,845,239	1,093,265,124	1,091,621,562	0	0
System.Char[]	726,008	16,857,070	16,818,660	0	0
JackCompiler.Token	463,000	7,408,000	7,396,640	0	0
System.String[]	189,008	5,484,312	5,486,996	0	0
System.Byte[]	3,005	5,160,090	5,236,770	0	0
System.Char	382,001	4,584,012	4,576,980	0	0
System.Predicate`1	140,001	4,480,032	4,474,080	0	0
System.Object[]	114,006	2,388,732	2,394,728	0	0

Figure 2-19 . The Object Lifetime view helps identify temporary objects that survive many garbage collections. In this view, all objects were reclaimed in generation 0, which is the cheapest kind of garbage collection available. (See Chapter 4 for more details.)

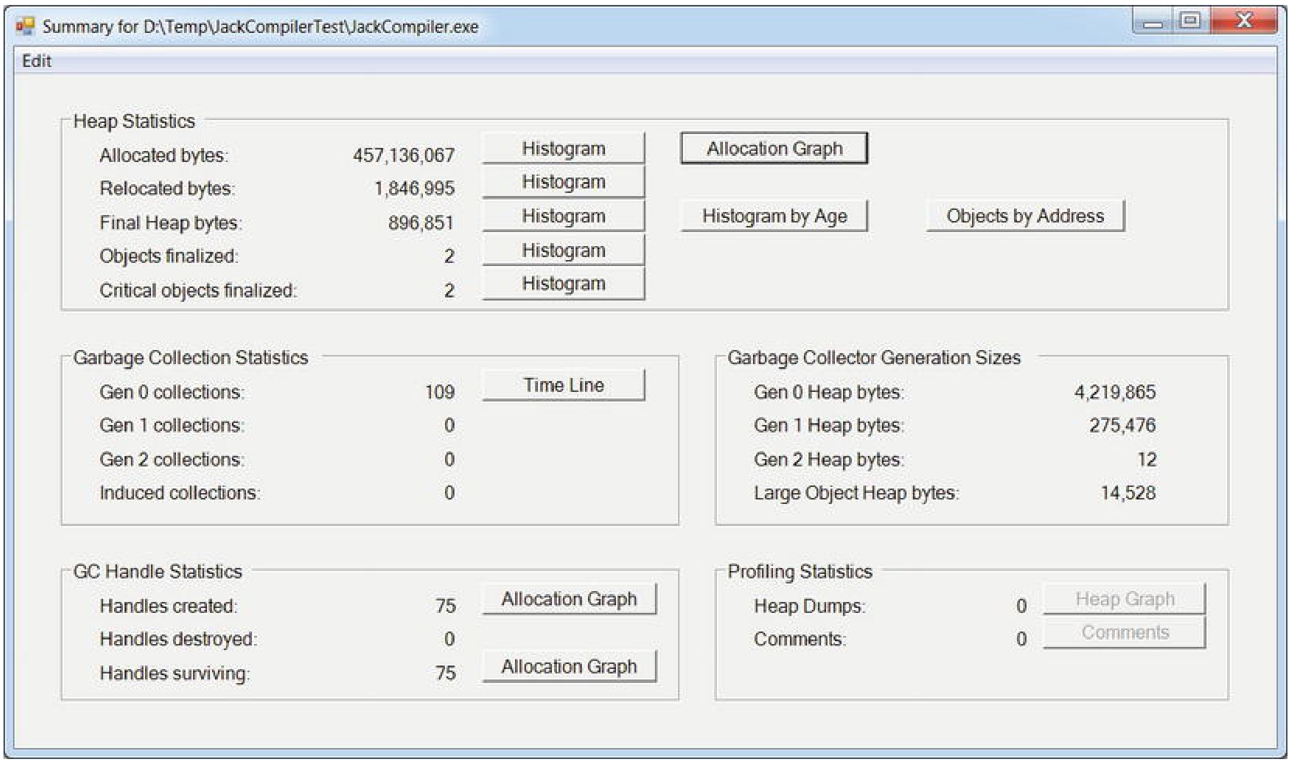
Although the allocation reports generated by the Visual Studio profiler are quite powerful, they are somewhat lacking in visualization. For example, tracing through allocation call stacks for a specific type is quite time-consuming if it is allocated in many places (as strings and byte arrays always are). CLR Profiler offers several visualization features, which make it a valuable alternative to Visual Studio.

CLR Profiler

CLR Profiler is a stand-alone profiling tool that requires no installation and takes less than 1MB of disk space. You can download it from http://www.microsoft.com/download/en/details.aspx?id=16273. As a bonus, it ships with complete sources, making for an interesting read if you're considering developing a custom tool using the CLR Profiling API. It can attach to a running process (as of CLR 4.0) or launch an executable, and record all memory allocations and garbage collection events.

While running CLR Profiler is extremely easy—run the profiler, click Start Application, select your application, and wait for the report to appear—the richness of the report's information can be somewhat overwhelming. We will go over some of the report's views; the complete guide to CLR Profiler is the CLRProfiler.doc document, which is part of the download package. As always, you can follow along by running CLR Profiler on the JackCompiler.exe sample application.

Figure 2-20 shows the main view, generated after the profiled application terminates. It contains basic statistics concerning memory allocations and garbage collections. There are several common directions to take from here. We could focus on investigation memory allocation sources to understand where the application creates most of its objects (this is similar to the Visual Studio profiler's Allocations view). We could focus on the garbage collections to understand which objects are being reclaimed. Finally, we could inspect visually the heap's contents to understand its general structure.


Summary for D:\Temp\JackCompilerTest\JackCompiler.exe

Heap Statistics

Allocated bytes:	457,136,067	Histogram
Relocated bytes:	1,846,995	Histogram
Final Heap bytes:	896,851	Histogram
Objects finalized:	2	Histogram
Critical objects finalized:	2	Histogram

Garbage Collection Statistics

Gen 0 collections:	109
Gen 1 collections:	0
Gen 2 collections:	0
Induced collections:	0

Garbage Collector Generation Sizes

Gen 0 Heap bytes:	4,219,865
Gen 1 Heap bytes:	275,476
Gen 2 Heap bytes:	12
Large Object Heap bytes:	14,528

GC Handle Statistics

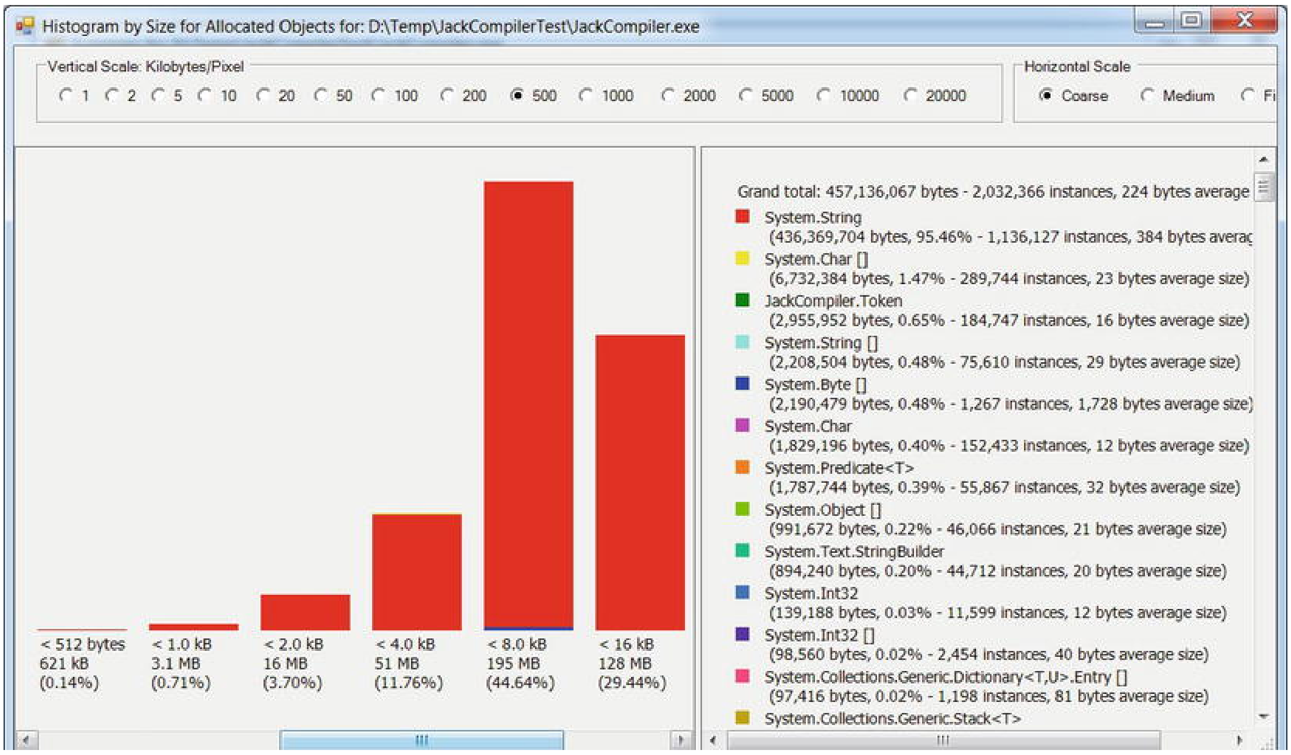
Handles created:	75
Handles destroyed:	0
Handles surviving:	75

Profiling Statistics

Heap Dumps:	0
Comments:	0

Figure 2-20 . CLR Profiler's main report view, showing allocation and garbage collection statistics

The **Histogram** buttons next to "Allocated bytes" and "Final heap bytes" in Figure 2-20 lead to a graph of object types grouped into bins according to their size. These histograms can be used to identify large and small objects, as well as the gist of which types the program is allocating most frequently. Figure 2-21 shows the histogram for all the objects allocated by our sample application during its run.


Histogram by Size for Allocated Objects for: D:\Temp\JackCompilerTest\JackCompiler.exe

Vertical Scale: Kilobytes/Pixel: 1, 2, 5, 10, 20, 50, 100, 200, [✓ 500], 1000, 2000, 5000, 10000, 20000

Horizontal Scale: [✓ Coarse], Medium, Fi

Bin	Size	Percentage
< 512 bytes	621 kB	(0.14%)
< 1.0 kB	3.1 MB	(0.71%)
< 2.0 kB	16 MB	(3.70%)
< 4.0 kB	51 MB	(11.76%)
< 8.0 kB	195 MB	(44.64%)
< 16 kB	128 MB	(29.44%)

Grand total: 457,136,067 bytes - 2,032,366 instances, 224 bytes average

🟥 System.String
(436,369,704 bytes, 95.46% - 1,136,127 instances, 384 bytes averag

🟨 System.Char []
(6,732,384 bytes, 1.47% - 289,744 instances, 23 bytes average size)

🟩 JackCompiler.Token
(2,955,952 bytes, 0.65% - 184,747 instances, 16 bytes average size)

🟦 System.String []
(2,208,504 bytes, 0.48% - 75,610 instances, 29 bytes average size)

🟦 System.Byte []
(2,190,479 bytes, 0.48% - 1,267 instances, 1,728 bytes average size)

🟪 System.Char
(1,829,196 bytes, 0.40% - 152,433 instances, 12 bytes average size)

🟧 System.Predicate<T>
(1,787,744 bytes, 0.39% - 55,867 instances, 32 bytes average size)

🟩 System.Object []
(991,672 bytes, 0.22% - 46,066 instances, 21 bytes average size)

🟩 System.Text.StringBuilder
(894,240 bytes, 0.20% - 44,712 instances, 20 bytes average size)

🟦 System.Int32
(139,188 bytes, 0.03% - 11,599 instances, 12 bytes average size)

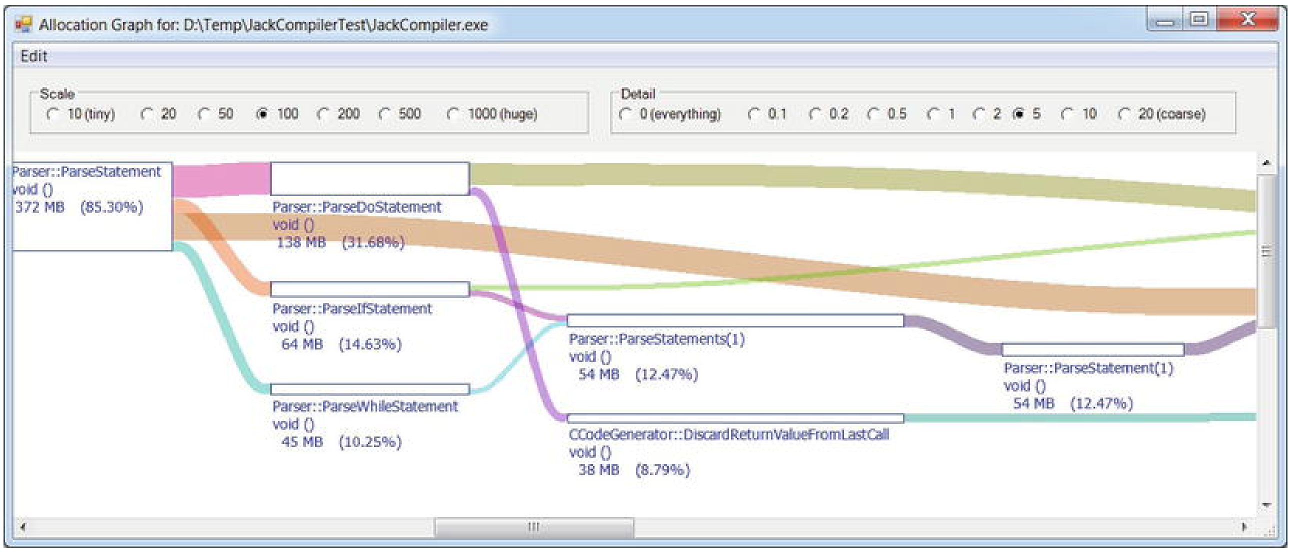
🟪 System.Int32 []
(98,560 bytes, 0.02% - 2,454 instances, 40 bytes average size)

🟥 System.Collections.Generic.Dictionary<T,U>.Entry []
(97,416 bytes, 0.02% - 1,198 instances, 81 bytes average size)

🟨 System.Collections.Generic.Stack<T>

Figure 2-21 . All objects allocated by the profiled application. Each bin represents objects of a particular size. The legend on the left contains the total number of bytes and instances allocated from each type

The **Allocation Graph** button in Figure 2-20 opens a view that shows the allocating call stacks for all the objects in the application in a grouped graph that makes it easy to navigate from the methods allocating most memory to individual types and see which methods allocated their instances. Figure 2-22 shows a small part of the allocation graph, starting from the `Parser.ParseStatement` method that allocated (inclusively) 372MB of memory, and showing the various methods it called in turn. (Additionally, the rest of CLR Profiler's views have a "Show who's allocated" context menu item, which opens an allocation graph for a subset of the application's objects.)


Allocation Graph for: D:\Temp\JackCompilerTest\JackCompiler.exe

Edit

Scale: 10 (tiny), 20, 50, [✓ 100], 200, 500, 1000 (huge)

Detail: 0 (everything), 0.1, 0.2, 0.5, 1, 2, [✓ 5], 10, 20 (coarse)

Parser::ParseStatement void () 372 MB (85.30%)

Parser::ParseDoStatement void () 138 MB (31.68%)

Parser::ParseIfStatement void () 64 MB (14.63%)

Parser::ParseWhileStatement void () 45 MB (10.25%)

Parser::ParseStatements(1) void () 54 MB (12.47%)

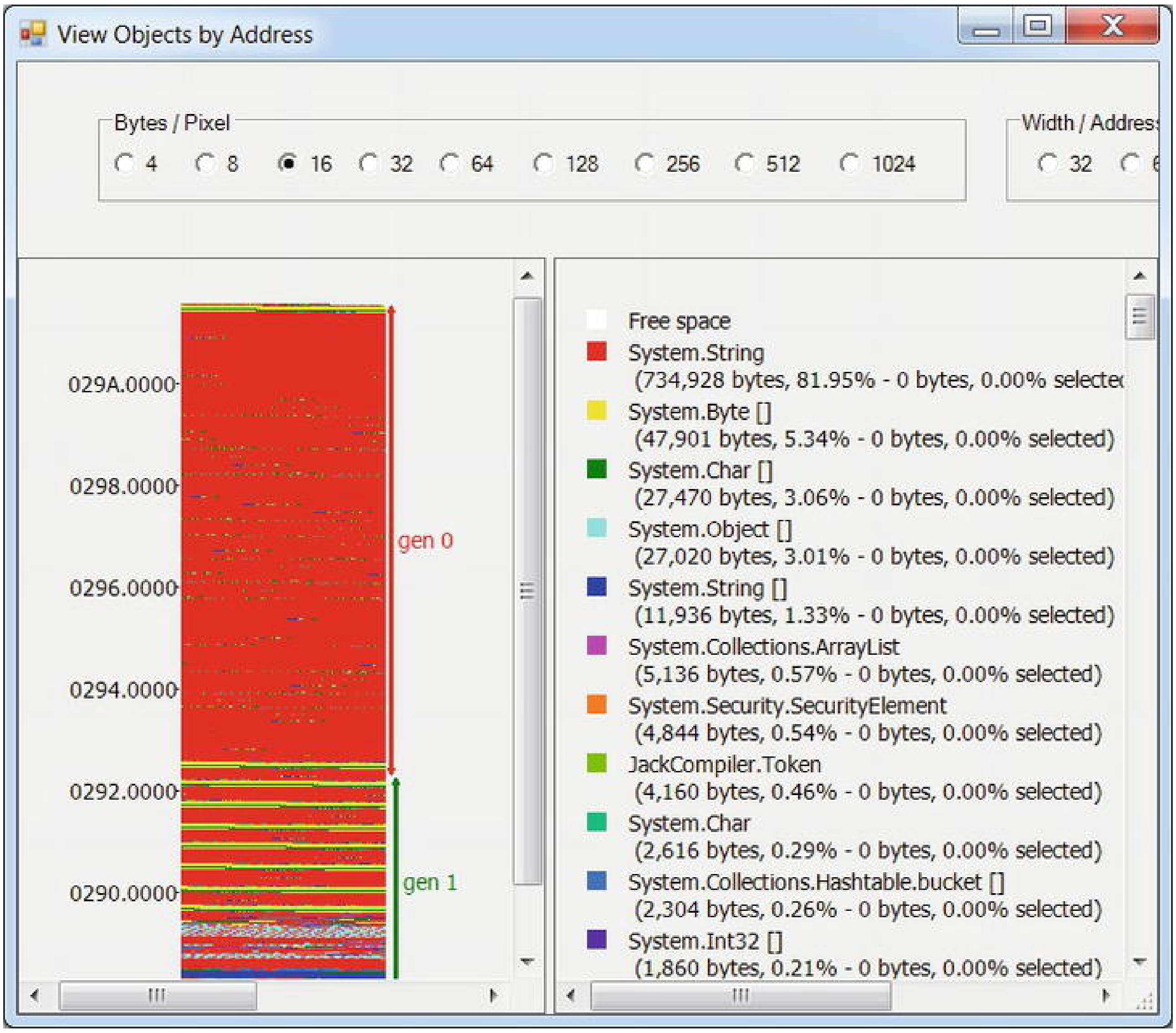
CCodeGenerator::DiscardReturnValueFromLastCall void () 38 MB (8.79%)

Parser::ParseStatement(1) void () 54 MB (12.47%)

Figure 2-22 . Allocation graph for the profiled applications. Only methods are shown here; the actual types allocated are on the far right of the graph

The **Histogram by Age** button in Figure 2-20 displays a graph that groups objects from the final heap to bins according to their age. This enables quick identification of long-lived objects and temporaries, which is important for detecting mid-life crisis situations. (We will discuss these in depth in Chapter 4.)

The **Objects by Address** button in Figure 2-20 visualizes the final managed heap memory regions in layers; the lowest layers are the oldest ones (see Figure 2-23). Like an archaeological expedition, you can dig through the layers and see which objects comprise your application's memory. This view is also useful for diagnosing internal fragmentation in the heap (e.g. due to pinning)—we will discuss these in more detail in Chapter 4.


View Objects by Address

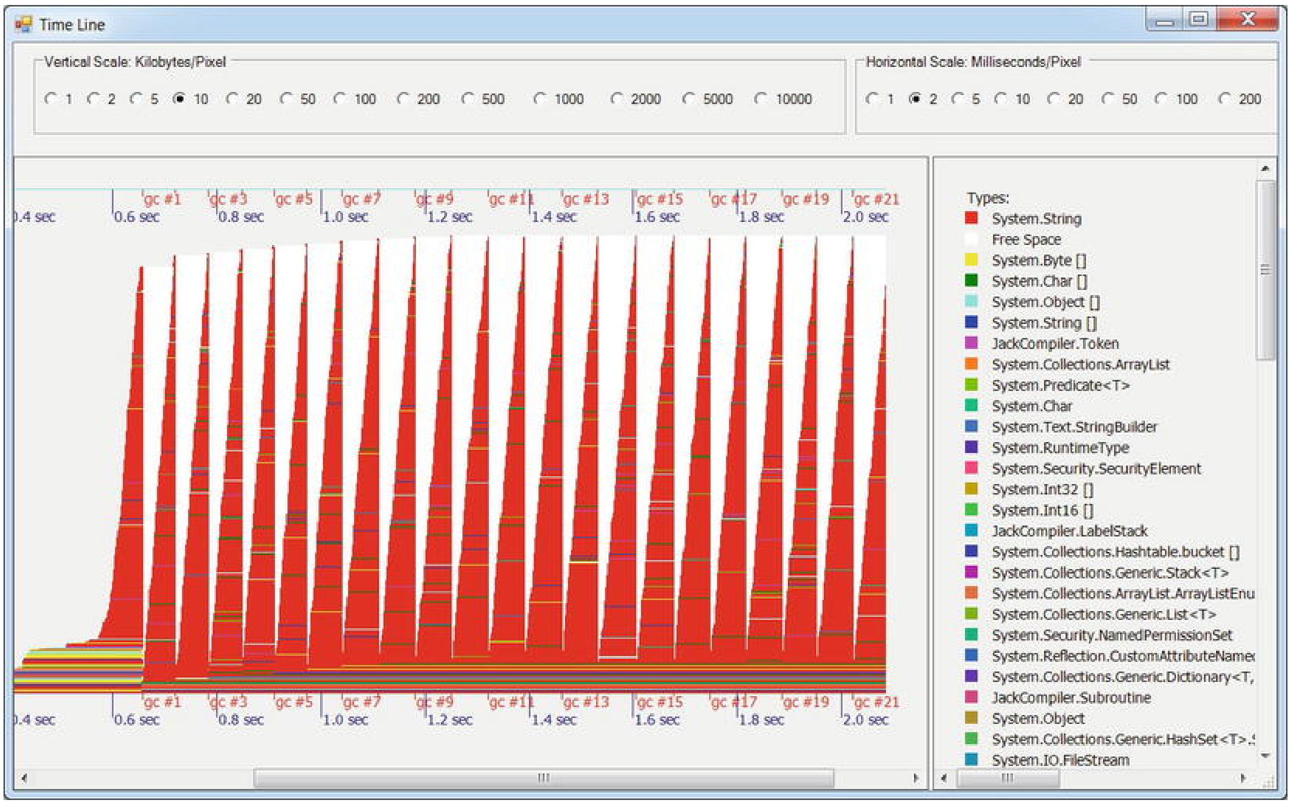
Bytes / Pixel: 4 8 [✓ 16] 32 64 128 256 512 1024

Width / Addres 32 6

⬜	Free space
🟥	System.String (734,928 bytes, 81.95% - 0 bytes, 0.00% selecte
🟨	System.Byte [] (47,901 bytes, 5.34% - 0 bytes, 0.00% selected)
🟩	System.Char [] (27,470 bytes, 3.06% - 0 bytes, 0.00% selected)
🟦	System.Object [] (27,020 bytes, 3.01% - 0 bytes, 0.00% selected)
🟦	System.String [] (11,936 bytes, 1.33% - 0 bytes, 0.00% selected)
🟪	System.Collections.ArrayList (5,136 bytes, 0.57% - 0 bytes, 0.00% selected)
🟧	System.Security.SecurityElement (4,844 bytes, 0.54% - 0 bytes, 0.00% selected)
🟩	JackCompiler.Token (4,160 bytes, 0.46% - 0 bytes, 0.00% selected)
🟩	System.Char (2,616 bytes, 0.29% - 0 bytes, 0.00% selected)
🟦	System.Collections.Hashtable.bucket [] (2,304 bytes, 0.26% - 0 bytes, 0.00% selected)
🟪	System.Int32 [] (1,860 bytes, 0.21% - 0 bytes, 0.00% selected)

Figure 2-23. Visual view of the application's heap. The labels on the left axis are addresses; the "gen 0" and "gen 1" markers are subsections of the heap, discussed in Chapter 4

Finally, the **Time Line** button in the Garbage Collection Statistics section in Figure 2-20 leads to a visualization of individual garbage collections and their effect on the application's heap (see Figure 2-24). This view can be used to identify which types of objects are being reclaimed, and how the heap is changing as garbage collections occur. It can also be instrumental in identifying memory leaks, where garbage collections do not reclaim enough memory because the application holds on to increasing amounts of objects.


Time Line

Vertical Scale: Kilobytes/Pixel
( ) 1 ( ) 2 ( ) 5 [✓ 10] ( ) 20 ( ) 50 ( ) 100 ( ) 200 ( ) 500 ( ) 1000 ( ) 2000 ( ) 5000 ( ) 10000

Horizontal Scale: Milliseconds/Pixel
( ) 1 [✓ 2] ( ) 5 ( ) 10 ( ) 20 ( ) 50 ( ) 100 ( ) 200

gc #1
gc #3
gc #5
gc #7
gc #9
gc #11
gc #13
gc #15
gc #17
gc #19
gc #21
).4 sec
0.6 sec
0.8 sec
1.0 sec
1.2 sec
1.4 sec
1.6 sec
1.8 sec
2.0 sec
Types:
🟥 System.String
Free Space
🟨 System.Byte []
🟩 System.Char []
🟦 System.Object []
🟦 System.String []
🟪 JackCompiler.Token
🟧 System.Collections.ArrayList
🟩 System.Predicate<T>
🟩 System.Char
🟦 System.Text.StringBuilder
🟪 System.RuntimeType
🟥 System.Security.SecurityElement
🟨 System.Int32 []
🟩 System.Int16 []
🟦 JackCompiler.LabelStack
🟦 System.Collections.Hashtable.bucket []
🟪 System.Collections.Generic.Stack<T>
🟧 System.Collections.ArrayList.ArrayListEnu
🟩 System.Collections.Generic.List<T>
🟩 System.Security.NamedPermissionSet
🟦 System.Reflection.CustomAttributeNamed
🟪 System.Collections.Generic.Dictionary<T,
🟥 JackCompiler.Subroutine
🟨 System.Object
🟩 System.Collections.Generic.HashSet<T>.
🟦 System.IO.FileStream

Figure 2-24 . Time line of an application's garbage collections. The ticks on the bottom axis represent individual GC runs, and the area portrayed is the managed heap. As garbage collections occur, memory usage drops significantly and then rises steeply until the next collection occurs. Overall, memory usage (after GC) is constant, so this application is not exhibiting a memory leak

Allocation graphs and histograms are very useful tools, but sometimes it's equally important to identify references between objects and not call stacks of methods. For example, when an application exhibits a managed memory leak, it can be very useful to crawl its heap, detect the largest object categories, and ascertain the object references that are preventing the GC from collecting these objects. While the profiled application is running, clicking the "Show Heap now" button generates a *heap dump*, which can be inspected later to categorize references between objects.

Figure 2-25 shows how three heap dumps are displayed simultaneously in the profiler's report, showing a an increase in the number of `byte[]` objects retained by the f-reachable queue (discussed in Chapter 4), through `Employee` and `Schedule` object references. Figure 2-26 shows the result of selecting "Show New Objects" from the context menu to see only the objects allocated between the second and third heap dumps.

Figure 2-25 . Three heap dumps on top of one another, showing 11MB of byte[] instances being retained

Figure 2-26 . The new objects allocated between the ultimate and penultimate heap dumps, showing that the source of the memory leak is clearly this reference chain from the f-reachable queue

You can use heap dumps in CLR Profiler to diagnose memory leaks in your applications, but the visualization tools are lacking. The commercial tools we discuss next offer richer capabilities, including automatic detectors for common memory leak sources, smart filters, and more sophisticated grouping. Because most of these tools don't record information for each allocated object and don't capture allocation call stacks, they introduce a lower overhead to the profiled application—a big advantage.