Chapter 5
Managed Heap and Garbage Collection

Manual memory management is a very common source of errors in applications today. As a matter of fact, several online studies indicate that the most common errors found are all related to manual memory management. Examples of such problems include:

[lb] Dangling pointers

[lb] Double free

[lb] Memory leaks

Automatic memory management serves to remove the tedious and error prone process of managing memory manually. Even though automatic memory management has gained more attention with the advent of Java and the .NET platform, the concept and implementation have been around for quite some time. Invented by John McCarthy in 1959 to solve the problems of manual memory management in LISP, other languages have implemented their own automatic memory management schemes as well. The implementation of an automatic memory management component has become almost universally known as a garbage collector (GC). The .NET platform also works on the basis of automatic memory management and implements its own highly peformant and reliable GC. While using a GC makes life a lot simpler for developers and enables them to focus on more of the business logic, having a solid understanding of how the GC operates is key to avoiding a set of problems that can occur when working in a garbage collected environment. In this chapter we take a look at the internals of the CLR heap manager and the GC and some common pitfalls that can wreak havoc in your application. We utilize the debuggers and a set of other tools to illustrate how we can get to the bottom of the problems.

Windows Memory Architecture Overview

Before we delve into the details of the CLR heap manager and GC it is useful to review the overall memory architecture of Windows. Figure 5-1 shows a high level overview of the various pieces commonly involved in a process.

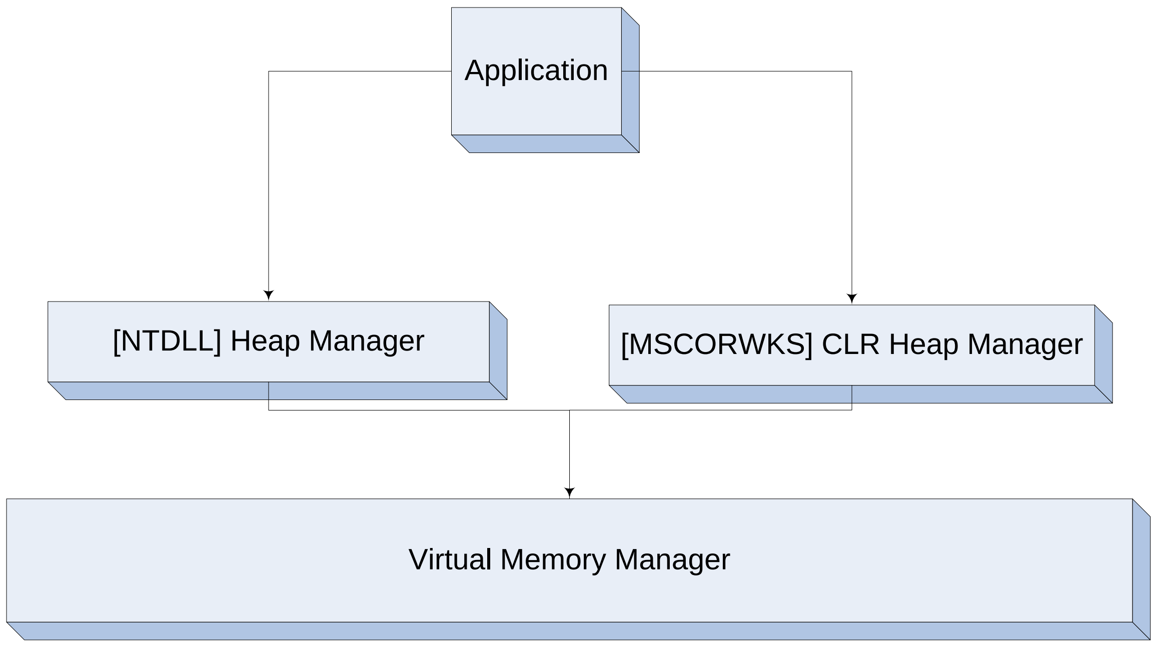


Figure 5-1
High level overview of Windows memory architecture

As you can see from Figure 5-1, processes that run in user mode typically use one or more heap managers. The most common heap managers are the Windows heap manager, which is used extensively in most user mode applications, and the CLR heap manager, which is used exclusively by .NET applications. The Windows heap manager is responsible for satisfying most memory allocation/deallocation requests by allocating memory, in larger chunks known as segments, from the Windows virtual memory manager and maintaining book keeping data (such as look aside and free lists) that enables it to efficiently break up the larger chunks into smaller sized allocations requested by the process. The CLR heap manager takes on similar responsibilities by being the one stop shop for all memory allocations in a managed process. Similar to the Windows heap manager, it also uses the Windows virtual memory manager to allocate larger chunks of memory, also known as segments, and satisfies any memory allocation/deallocation requests from those segments. They key difference between the two heap managers is how the book keeping data is structured in order to maintain the integrity of the heap. Figure 5-2 shows a high level overview of the CLR heap manager.

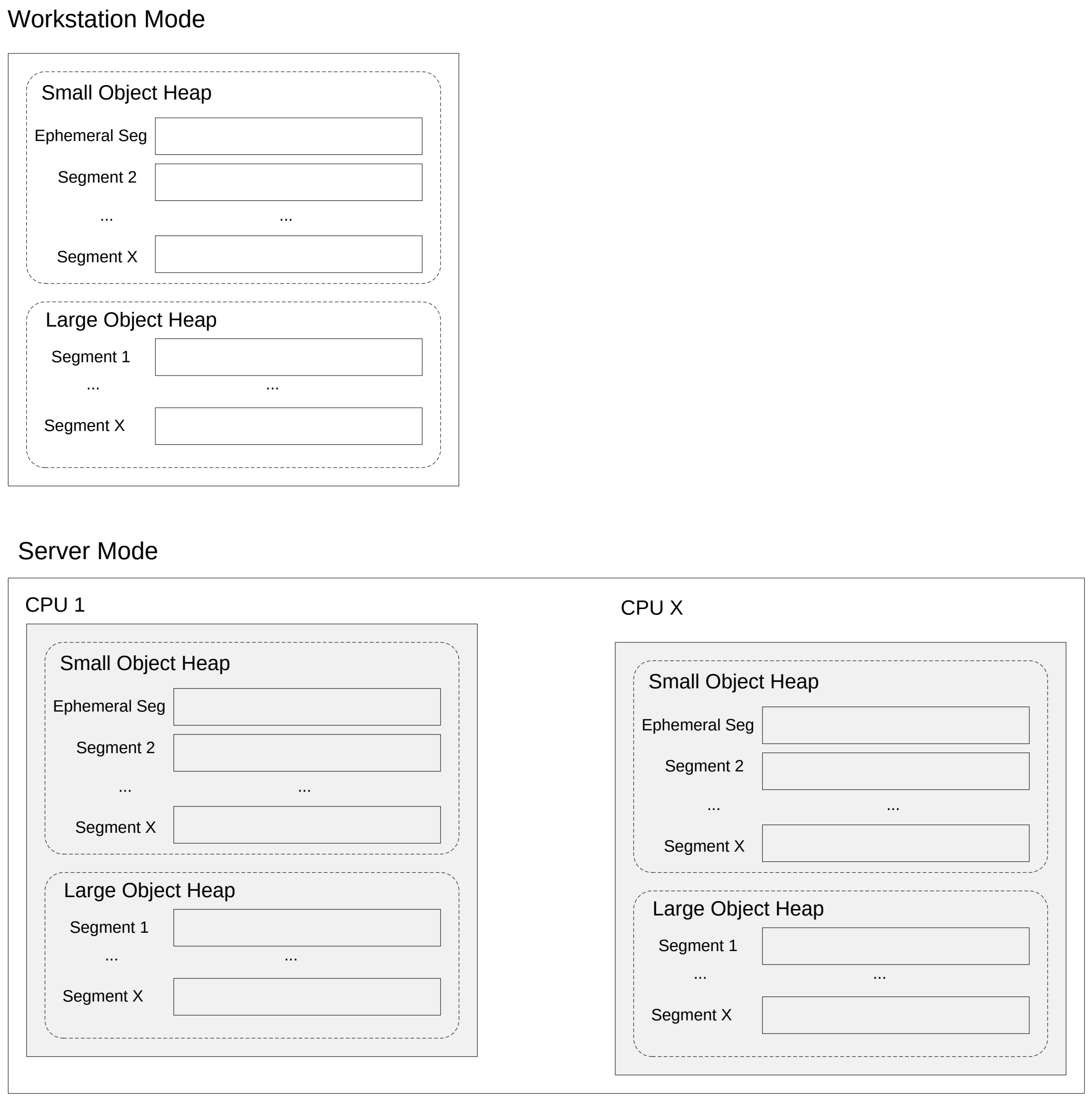


Figure 5-2
High level overview of the CLR heap manager

From Figure 5-2 you can see how the CLR heap manager uses carefully managed larger chunks (segments) to satisfy the memory requests. Also interesting to note from Figure 5-2 is the mode in which the CLR heap manager can operate. There are two modes of operation: workstation and server. The primary difference in as far as the *CLR heap manager* is concerned is that rather than having just one heap, there is now one heap per processor where the size of the heap segments is typically larger than that of the workstation heap (although this is an implementation detail that should not be relied upon). From the GC's perspective, there are other fundamental differences between workstation and server primarily in the area of GC threading models, where the server flavor of the GC has a dedicated thread for all GC activity versus the workstation GC, which runs the GC process on the thread that performed the memory allocation.

Are the implementations for workstation and server in different binaries?

Prior to version 2.0, the workstation GC was implemented in `mscorwks.dll` and the server GC was implemented in `mscorsvr.dll`. In version 2.0 the implementations were folded into one and the same binary (`mscorwks.dll`). Please note that this is purely a merge at the binary level.

Each managed process starts out with two heaps with their own respective segments that are initialized when the CLR is loaded. The first heap is known as the small object heap and has one initial segment of size 16MB on workstations (the server version is bigger). The small object heap is used to hold objects that are less than 85,000 bytes in size. The second heap is known as the Large Object Heap (LOH) and has one initial segment of size 16MB. The LOH is used to hold objects greater than or equal to 85,000 bytes in size. We will see the reason behind dividing the heaps based on object size limits later when we discuss the garbage collector internals. It is important to note that when a segment is created, not all of the memory in that segment is committed; rather, the CLR heap manager reserves the memory space and commits on demand. Whenever a segment is exhausted on the small object heap, the CLR heap manager triggers a GC and expands the heap if space is low. On the large object heap, however, the heap manager creates a new segment that is used to serve up memory. Conversely, as memory is freed by the garbage collector, memory in any given segment can be decommitted as needed, and once a segment is fully decommitted it might be freed all together.

What's in an Address?

Given an address, is there a way to find out the state of that memory? I.e., is the memory reserved? Is the memory committed? Is it writable or just readable? An excellent command to answer those questions is the `address` command. Without any arguments, the `address` command gives a detailed view of the memory activity in the process as well as a summary. If an address is specified, the `address` command attempts to find information about that particular address. For example, assume we have an object on the managed heap at address `0x01d96c58`. If we run the `address` command on this address it shows the following:

```
0:000> !address 0x01d96c58
 ProcessParametrs 004d1668 in range 004d0000 0050b000
 Environment 004d0808 in range 004d0000 0050b000
    01d90000 : 01d90000 - 00012000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageIsVAD
```

The address in question is an allocated and accessible memory location that is read/write enabled and committed.

As briefly mentioned in Chapter 2, "CLR Fundamentals," each object that resides on the managed heap carries with it some additional metadata. More specifically, each object is prefixed by an additional 8 bytes. Figure 5-3 shows an example of a small object heap segment.

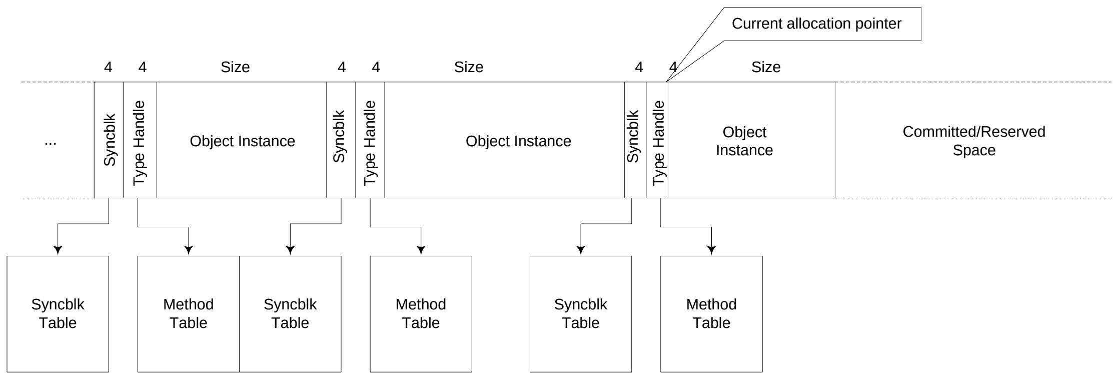


Figure 5-3
Example of a small object heap segment

In Figure 5-3 we can see that the first 4 bytes of any object located on the managed heap is the sync block index followed by an additional 4 bytes that indicate the method table pointer.

Allocating Memory

Now that we understand how the CLR heap manager, at a high level, structures the memory available to applications we can take a look at how allocation requests are satisfied. We already know that the CLR heap manager consists of one or more segments and that memory allocations are allotted from one of the segments and returned to the caller. How is this memory allocation performed? Figure 5-4 illustrates the process that the CLR heap manager goes through when a memory allocation request arrives.

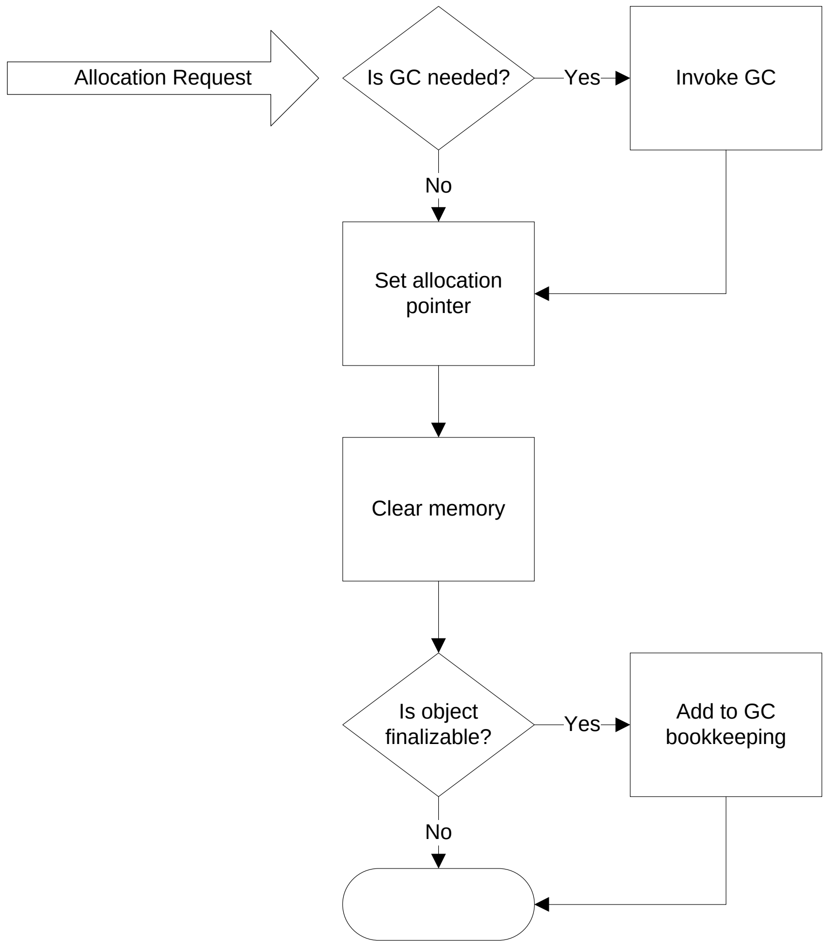


Figure 5-4
Memory allocation process in the CLR heap manager

In the most optimal case, when a GC is not needed for the allocation to succeed, an allocation request is satisfied very efficiently. The two primary tasks performed in that scenario are those of simply advancing a pointer and clearing the memory region. The act of advancing the allocation pointer implies that new allocations are simply tacked on after the last allocated object in the segment. When another allocation request is satisfied the allocation pointer is once again advanced and so forth. Please note that this allocation scheme is quite different than the Windows heap manager in the sense that the Windows heap manager does not guarantee locality of objects on the heap in the same fashion. An allocation request on the Windows heap manager can be satisfied from any given free block anywhere in the segment. The other scenario to consider is what happens when a GC is required due to breaching a memory threshold. In this case, a GC is performed and the allocation attempt is tried again. The last interesting aspect from Figure 5-4 is that of checking to see if the allocated object is finalizable. While not, strictly speaking, a function of the managed heap, it is important to call out as it is part of the allocation process. If an object is finalizable, a record of it is stored in the GC to properly manage the lifetime of the object. We will discuss finalizable objects in more detail later in the chapter.

Before we move on and discuss the garbage collector internals let's take a look at a very simple application that performs a memory allocation. The source code behind the application is shown in Listing 5-1.

Listing 5-1

Simple memory allocation

```
using System;
using System.Text;
using System.Runtime.Remoting;

namespace Advanced.NET.Debugging.Chapter5
{
    class Name
    {
        private string first;
        private string last;

        public string First { get { return first; } }
        public string Last { get { return last; } }

        public Name(string f, string l)
        {
            first = f; last = l;
        }
    }

    class SimpleAlloc
    {
        static void Main(string[] args)
        {
            Name name = null;

            Console.WriteLine("Press any key to allocate memory");
            Console.ReadKey();

            name = new Name("Mario", "Hewardt");

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

The source code and binary for Listing 5-1 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\SimpleAlloc`

[lb] Binary: `C:\ADNDBin\05SimpleAlloc.exe`

The source code in Listing 5-1 is painfully simple, but the more interesting question is how we can find that particular memory allocation on the managed heap using the debuggers. Fortunately, the SOS debugger extension has a few handy commands that enable us to get some insight into the contents of the managed heap. The command we will use in this particular example is the `DumpHeap` command. By default, the `DumpHeap` command will list all the objects that are stored on the managed heap together with their associated address, method table, and size. Let's run our `05SimpleAlloc.exe` application under the debugger and break execution once the `Press any key to allocate memory` prompt is shown. Once execution breaks into the debugger run the `DumpHeap` command. A partial listing of the output of the command is shown in the following:

```
0:004> !DumpHeap
 Address       MT     Size
790d8620 790fd0f0       12
790d862c 790fd8c4       28
790d8648 790fd8c4       32
790d8668 790fd8c4       32
790d8688 790fd8c4       28
790d86a4 790fd8c4       24
790d86bc 790fd8c4       24
…
…
…
total 2379 objects
Statistics:
      MT    Count    TotalSize Class Name
79119954        1           12
System.Security.Permissions.ReflectionPermission
79119834        1           12
System.Security.Permissions.FileDialogPermission
791032a8        1          128 System.Globalization.NumberFormatInfo
79100e38        3          132 System.Security.FrameSecurityDescriptor
791028f4        2          136 System.Globalization.CultureInfo
791050b8        4          144 System.Security.Util.TokenBasedSet
790fe284        2          144 System.Threading.ThreadAbortException
79102290       13          156 System.Int32
790f97c4        3          156 System.Security.Policy.PolicyLevel
790ff734        9          180 System.RuntimeType
790ffb6c        3          192 System.IO.UnmanagedMemoryStream
7912d7c0       11          200 System.Int32[]
790fd0f0       17          204 System.Object
79119364        8          256
System.Collections.ArrayList+SyncArrayList
79101fe4        6          336 System.Collections.Hashtable
79100a18       10          360 System.Security.PermissionSet
79112d68       18          504
System.Collections.ArrayList+ArrayListEnumeratorSimple
79104368       21          504 System.Collections.ArrayList
7912d9bc        6          864 System.Collections.Hashtable+bucket[]
7912dae8        8         1700 System.Byte[]
7912dd40       14         2296 System.Char[]
7912d8f8       23        17604 System.Object[]
790fd8c4     2100       132680 System.String
Total 2379 objects
```

The output of the `DumpHeap` command is divided into two sections. The first section contains the entire list of objects located on the managed heap. The `DumpObject` command can be used on any of the listed objects to get further information about the object. The second section contains a statistical view of the managed heap activity by grouping related objects and displaying the method table, count, total size, and the object's type name. For example, the following item:

```
79100a18       10          360 System.Security.PermissionSet
```

indicates that the object in question is a `PermissionSet` with a method descriptor of `0x79100a18` and that there are `10` instances on the managed heap with a total size of `360` bytes. The statistical view can be very useful when trying to understand an excessively large managed heap and which objects may be causing the heap to grow.

The DumpHeap command produces quite a lot of output and it can be difficult to find a particular allocation in the midst of all of the output. Fortunately, the DumpHeap command has a variety of switches that makes life easier. For example, the –type and –mt switches enable you to search the managed heap for a given type name or a method table address. If we ran the DumpHeap command with the –type switch looking for the allocation our application makes we get the following:

```
0:003> !DumpHeap -type Advanced.NET.Debugging.Chapter5.Name
 Address       MT     Size
total 0 objects
Statistics:
      MT    Count    TotalSize Class Name
Total 0 objects
```

The output clearly indicates that there are no allocations on the managed heap of the given type. This, of course, makes perfect sense since our sample application has not performed its allocation. Resume execution of the application until you see the Press any key to exit prompt. Once again, break execution and run the DumpHeap command again with the –type switch:

```
0:004> !DumpHeap –type Advanced.NET.Debugging.Chapter5.Name
Address       MT     Size
01ca6c7c 002030cc       16
total 1 objects
Statistics:
      MT    Count    TotalSize Class Name
002030cc        1           16 Advanced.NET.Debugging.Chapter5.Name
Total 1 objects
```

This time we can see that we have an instance of our type on the managed heap. The output follows the same structure as the default DumpHeap output by first showing the instance specific data (address, method table and size) followed by the statistical view, which shows the managed heap only having one instance of our type.

The DumpHeap command has several other useful switches depending on the debugging scenario at hand. Table 5-1 details the switches available.

Table 5-1
DumpHeap switches

Switch	Description
-stat	Limits output to managed heap statistics
-strings	Limits output to strings stored on the managed heap
-short	Limits output to just the address of the objects on the managed heap
-min	Filters based on the minimum object size specified
-max	Filters based on the maximum object size specified
-thinlock	Outputs objects with associated thinlocks
-startAtLowerBound	Begin walking the heap at a lower bound
-mt	Limit output to the specified method table
-type	Limit output to the specified type name (substring match)

This concludes our high level discussion of the Windows memory architecture and how the CLR heap manager fits in. We’ve looked at how the CLR heap manager organizes memory to provide an efficient memory management scheme as well as the process that the CLR heap manager goes through when a memory allocation request arrives at its doorstep. The next big question is how the GC itself functions, its relationship to the CLR heap manager, and how memory is actually freed once it has been considered discarded.

Are there other types of CLR heaps?

In addition to the CLR heap, which is the heap typically used during "day-to-day" memory allocations, there are other types of heaps as well. For example, when the JIT compiler translates IL to machine code it uses its own heap. Another example is the CLR loader, which utilizes yet another heap. The internals of these heaps are for the most part undocumented and typically not traversed (outside of SOS) during a debug session.

Garbage Collector Internals

The CLR GC is a highly efficient, scalable, and reliable automatic memory manager. A lot of time and effort went into researching the optimal behavioral characteristics of the GC. Before delving into the details of the CLR GC it is important to state the definition of what the GC is and also what assumptions were made during its design and implementation. Let's begin by looking at some of the key assumptions:

[lb] The CLR GC assumes that everything is garbage unless otherwise told. What this means is that the GC is ready to collect *all* objects on the managed heap unless told otherwise. In essence, it implements a *reference tracking* scheme for all live objects in the system (we will define what live means shortly) where objects without any references to them are considered garbage and can be collected.

[lb] The CLR GC assumes that all objects on the managed heap will be *short lived (or ephemeral)*. In other words, the GC will attempt to collect short lived objects more often than long lived objects operating under the assumption that if an object has been around for a while chances are it will be around for a little longer and there is no need to attempt to collect that object again.

[lb] The CLR GC tracks an object's age via the use of generations. Young objects are placed in generation 0 and older objects in generations 1 and 2. As an object grows older it is promoted from one generation to the next. As such a generation can be said to define the age of an object.

Based upon the assumptions above, we can arrive at a definition of the CLR GC: is a *reference tracking* and *generational* garbage collector.

Let's look at each of the parts of the definition more concretely and begin with how generations define the age of an object.

Generations

The CLR GC defines three generations very innovatively called: generation 0, generation 1 and generation 2. Each of the generations contains objects of a certain age where generation 0 contains newly allocated objects and generation 2 contains the oldest of objects. An object moves from one generation to the next by surviving a garbage collection. By surviving it's implied that the object was still being referenced (or is still rooted) at the time of the garbage collection. Each of the generations can be garbage collected at any time but the frequency of garbage collections depend on the generation. Remember from the previous section that one of the assumptions that the CLR makes it that most objects are going to be short lived (i.e., live in generation 0). Due to that assumption, generation 0 is collected far more frequently than generation 2 in hopes to prune these short lived objects quicker. Figure 5-5 shows the overall algorithm when it comes to how the generations are garbage collected.

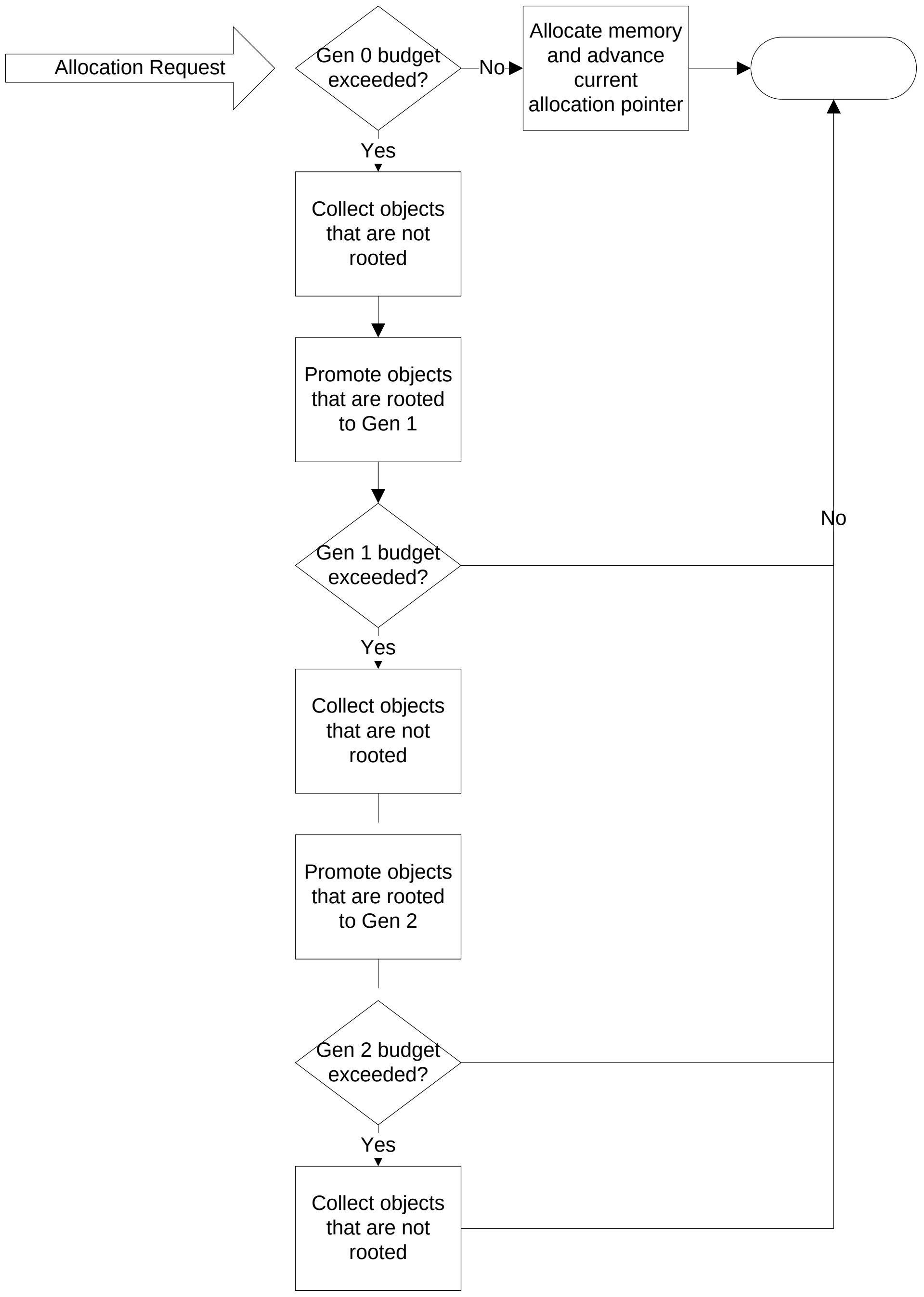

Allocation Request
Gen 0 budget exceeded?
No
Allocate memory and advance current allocation pointer
Yes
Collect objects that are not rooted
Promote objects that are rooted to Gen 1
No
Gen 1 budget exceeded?
Yes
Collect objects that are not rooted
Promote objects that are rooted to Gen 2
Gen 2 budget exceeded?
Yes
Collect objects that are not rooted

Figure 5-5
High level overview of generational garbage collection algorithm

In Figure 5-5 we can see that the triggering of a garbage collection is by new allocation request and when the budget for generation 0 has been exceeded. If so, the garbage collector collects all objects that have no roots associated with them and promotes all objects *with* roots to generation 1. Much in the same way that generation 0 has a budget defined so does generation 1 and if, as part of promoting objects from generation 0 to generation 1, the budget is exceeded, the GC will repeat the process of collecting objects with no roots in generation 1 and promoting objects with roots to generation 2. The process repeats itself for generation 2. If, while promoting to generation 2, the GC is unable to collect any objects and the budget for generation 2 is exceeded, the CLR heap manager will try to allocate another segment that will hold generation 2 objects. If the creation of a new segment fails, an `OutOfMemoryException` is thrown. The CLR heap manager also releases segments if they are not in use anymore and we will discuss that process in more detail later in the chapter.

What else can trigger a garbage collection?

In addition to a garbage collection occurring due to the allocation of memory and exceeding the thresholds for generation 0, 1, and 2 respectively a couple of other scenarios exist that can cause it to happen. First, a garbage collection can be forced via the `GC.Collect` and related API's. Secondly, the garbage collector is very cognizant of memory usage in the system as a whole. Through thorough collaboration with the operating system, the garbage collector can kick start a collection if the system as a whole is found to be under extreme memory pressure.

Let's take a practical look at how an object is collected and promoted. Listing 5-2 shows the source code behind the application we will use to illustrate the generational concepts.

Listing 5-2

Example source code to illustrate generational concepts

```
using System;
using System.Text;
using System.Runtime.Remoting;

namespace Advanced.NET.Debugging.Chapter5
{
    class Name
    {
        private string first;
        private string last;

        public string First { get { return first; } }
        public string Last { get { return last; } }

        public Name(string f, string l)
        {
            first = f; last = l;
        }
    }

    class Gen
    {
        static void Main(string[] args)
        {
            Name n1 = new Name("Mario", "Hewardt");
            Name n2 = new Name("Gemma", "Hewardt");

            Console.WriteLine("Allocated objects");

            Console.WriteLine("Press any key to invoke GC");
            Console.ReadKey();

            n1 = null;
            GC.Collect();
            Console.WriteLine("Press any key to invoke GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

The source code and binary for Listing 5-2 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\Gen`

[lb] Binary: `C:\ADNDBin\05Gen.exe`

In Listing 5-2 we have defined a simple type called `Name`. In the `Main` method we instantiate two instances of the `Name` type which both end up going to generation 0 as new allocations. Once the user has been prompted to `Press any key to invoke GC` we set the `n1` instance to `null`, which indicates that it can be garbage collected since it no longer has any roots. Next, the garbage collection occurs and collects `n1` and promotes `n2` to generation 1. Finally, the last garbage collection promotes `n2` to generation 2 since it is still rooted.

Let's run the application under the debugger and see how we can verify our theories on how `n1` and `n2` are collected and promoted. Once the application is running under the debugger, resume execution until the first `Press any key to invoke GC` prompt. At that point, we need to break execution and find the addresses to the two object instances, which can easily be done via the `ClrStack` command as shown in the following:

```
0:000> !ClrStack -a
OS Thread Id: 0x1c0c (0)
ESP       EIP
0028f3b4 77709a94 [NDirectMethodFrameSlim: 0028f3b4]
Microsoft.Win32.Win32Native.ReadConsoleInput(IntPtr, InputRecord ByRef,
Int32, Int32 ByRef)
0028f3cc 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x0028f3dc = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>

0028f40c 793e8e33 System.Console.ReadKey()
0028f410 003000f3
Advanced.NET.Debugging.Chapter5.Gen.Main(System.String[])
    PARAMETERS:
        args = 0x01c55818
    LOCALS:
        <CLR reg> = 0x01da5938
        <CLR reg> = 0x01da5948

0028f65c 79e7c74b [GCFrame: 0028f65c]
```

The addresses of the two objects on the managed heap are `0x01da5938` and `0x01da5948`. How can we figure out which generation objects on the managed heap belong to? The answer to that lies in understanding the correlation between managed heap segments and generations. As previously discussed, each managed heap consists of one or more segments where the objects reside. Furthermore, part of the segment(s) is dedicated to a given generation. Figure 5-6 shows an example of a hypothetical managed heap segment.

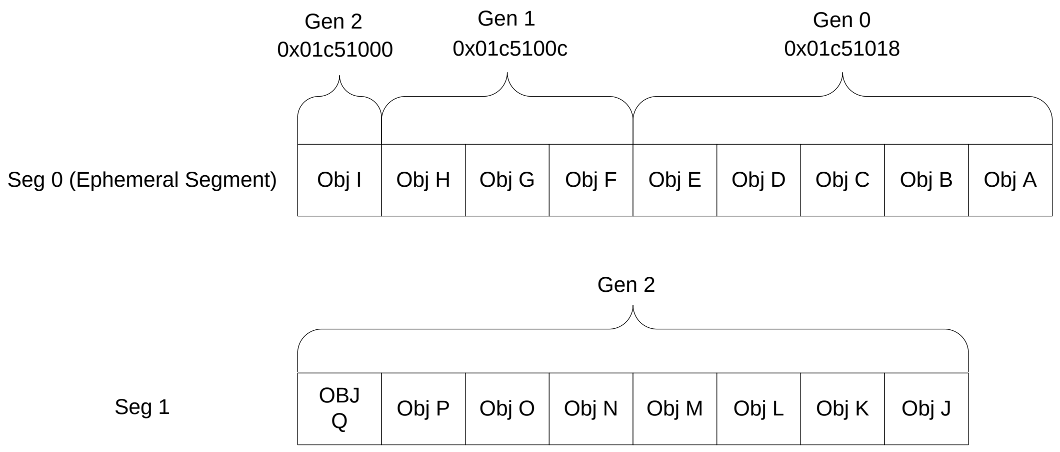


Figure 5-6
Hypothetical managed heap segment

In Figure 5-6, the managed heap segment is divided into three generations each with its own starting address managed by the CLR heap manager. Generations 0 and 1 are part of a single segment known as the ephemeral segment where short lived objects live. Since the GC goes under the assumption that most objects are short lived, most objects are not expected to live past generation 0 or, at a maximum, generation 1. Objects that live in generation 2 are the oldest objects and get collected very infrequently. It is possible that generation 2 can also be part of the ephemeral segment even though generation 2 is not collected as often. By looking at an object's address and knowing the address ranges for each of the generations we can find out which generation an object belongs to. How do we know what the generational starting addresses for the CLR heap manager are? The answer lies in a command called `eeheap`. The `eeheap` command displays various memory statistics of data consumed by internal CLR data structures. By default, `eeheap` displays verbose data meaning that information related to the GC as well as the loader is displayed. To display information only about the GC the `–gc` switch can be used. Let's run the command in our existing debug session and see what we get:

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01da1018
generation 1 starts at 0x01da100c
generation 2 starts at 0x01da1000
ephemeral segment allocation context: none
```

```
 segment    begin allocated     size
002c7db0 790d8620  790f7d8c 0x0001f76c(128876)
01da0000 01da1000  01da8010 0x00007010(28688)
Large object heap starts at 0x02da1000
 segment    begin allocated     size
02da0000 02da1000  02da3250 0x00002250(8784)
Total Size   0x289cc(166348)
------------------------------
GC Heap Size   0x289cc(166348)
```

Part of the output shows clearly the starting addresses of each of the generations. If we look at the object addresses in the debug session of our sample application we can see the following:

```
        <CLR reg> = 0x01da5938
        <CLR reg> = 0x01da5948
```

Both of these addresses corresponding to our objects fall within the address range of generation 0 (starting at `0x01da1018`), hence we can conclude that both of them live within the realm of that generation. This makes perfect sense since we are currently in the code flow where the objects were just allocated and we are pending a garbage collection. If we resume execution of the application and subsequently break execution again the next time we see the `Press any key to invoke GC` we should see some difference in as far as which generation the objects belong to. If we look at the source code, we can see that prior to invoking a garbage collection we set the `n1` reference to `null`, which in essence makes the object rootless and one that should be garbage collected. Furthermore, `n2` is still rooted and as such should be promoted to generation 1 during the garbage collection. Let's take a look by following the same process as earlier: find the object addresses, use the `eeheap` command to find the generational address ranges, and see which generation the object falls into:

```
0:000> !ClrStack -a
OS Thread Id: 0x1910 (0)
ESP       EIP
0021f394 77709a94 [NDirectMethodFrameSlim: 0021f394]
Microsoft.Win32.Win32Native.ReadConsoleInput(IntPtr, InputRecord ByRef,
Int32, Int32 ByRef)
0021f3ac 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x0021f3bc = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>

0021f3ec 793e8e33 System.Console.ReadKey()
0021f3f0 01690111
Advanced.NET.Debugging.Chapter5.Gen.Main(System.String[])
    PARAMETERS:
        args = 0x01da5818
    LOCALS:
        <CLR reg> = 0x00000000
        <CLR reg> = 0x01da5948

0021f644 79e7c74b [GCFrame: 0021f644]
0:000> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01da6c00
generation 1 starts at 0x01da100c
generation 2 starts at 0x01da1000
ephemeral segment allocation context: none
 segment    begin allocated     size
002c7db0 790d8620  790f7d8c 0x0001f76c(128876)
01da0000 01da1000  01da8c0c 0x00007c0c(31756)
Large object heap starts at 0x02da1000
 segment    begin allocated     size
02da0000 02da1000  02da3240 0x00002240(8768)
Total Size   0x295b8(169400)
------------------------------
GC Heap Size   0x295b8(169400)
```

The most interesting part of the output is in the `eeheap` command output. We can see now that the generational address ranges have changed slightly. More specifically, the starting address of generation 0 has changed from `0x01da1018` to `0x01da6c00` which in essence implies that generation 1 has become bigger (since the starting address of generation 1 remains unchanged). If we correlate the address of our `n2` object (`0x01da5948`) with the generational address ranges that the `eeheap` command displayed, we can see that the `n2` object falls into generation 1. Again, this is fully expected since `n2` previously lived in generation 0 and was still rooted at the time of the garbage collection, thereby promoting the object to the next generation. I will leave it as an exercise to the reader to see what happens on the final garbage collection in the sample application.

While the SOS debugger extension provides the means of finding out which generation any given object belongs to it is a somewhat tedious process as it requires that addresses be checked against potentially changing generational addresses within any given managed heap segment. Furthermore, there is no concrete way to list all the objects that fall into any given generation, making it hard to get an overall picture of the per generation utilization. Fortunately, the SOSEX extension comes to the rescue with a command named `dumpgen`. With the dumpgen command you can easily get a list of all objects that belong to the generation specified as an argument to the command. For example, using the same sample application as shown in Listing 5-2, here is the output when running `dumpgen`:

```
0:000> !dumpgen 0
01da6c00             12 **** FREE ****
01da6c0c             68 System.Char[]
2 objects, 80 bytes
0:000> !dumpgen 1
01da100c             12 **** FREE ****
01da1018             12 **** FREE ****
01da1024             72 System.OutOfMemoryException
01da106c             72 System.StackOverflowException
01da10b4             72 System.ExecutionEngineException
01da10fc             72 System.Threading.ThreadAbortException
01da1144             72 System.Threading.ThreadAbortException
01da118c             12 System.Object
01da1198             28 System.SharedStatics
01da11b4            100 System.AppDomain
```

```
...
...
...
01da5948            16 Advanced.NET.Debugging.Chapter5.Name
01da5958            28 Microsoft.Win32.Win32Native+InputRecord
01da5974            12 System.Object
01da5980            20 Microsoft.Win32.SafeHandles.SafeFileHandle
01da5994            36 System.IO.__ConsoleStream
01da59b8            28 System.IO.Stream+NullStream
…
…
…
```

Here we can see that there aren't a whole lot of objects in generation 0 but instead we have a ton of objects in generation 1 including our `n2` instance at address `0x01da5948`. The `dumpgen` command really makes life much easier when looking at generation specific data.

What about GC.Collect()?

As you have seen, the source code in Listing 6-2 (as well as throughout the chapter) contains calls to `GC.Collect()`. The `GC.Collect()` API does pretty much what the name implies. It forces a garbage collection to occur irrespective of whether it is needed. The last part of the previous statement is extremely important: *irrespective of whether it is needed*. The GC continuously fine tunes itself throughout the execution of the application to ensure that it behaves optimally under the application's circumstances. By invoking `GC.Collect()`, and thereby forcing a garbage collection, it can wreak havoc with the GC's fine tuning algorithm. Under normal circumstances it is therefore *highly* recommended not to use the API. The usage of the API in the book is solely to make the examples more deterministic.

So far we have discussed how objects live in managed heap segments divided into generations and how these objects are either garbage collected or promoted to the next generation depending on if they are still referenced (or still rooted). One question that still remains is what it means for an object to be rooted. The next section introduces the notion of roots, which are at the heart of the decision making process the GC uses to determine if an object can be collected.

Roots

One of the most fundamental aspects of a garbage collection is that of being able to determine which objects are still being referenced and which are objects are not and can be considered for garbage collection. Contrary to popular belief, the GC itself does not implement the logic for detecting which objects are still being referenced but rather uses other components in the CLR that have far more knowledge about the lifetimes of the objects. Some of the components that the CLR uses to determine which objects are still referenced are:

[lb] Just in time compiler. The JIT compiler is the component responsible for translating IL to machine code and as such has detailed knowledge of which local variables were considered active at any given point in time. The JIT

compiler maintains this information in a table that it subsequently references when the GC asks for objects that are still considered to be alive.

Retail Versus Debug Builds

Please note that there *can* be a difference between retail and debug builds when it comes to the JIT compiler tracking the aliveness of local variables. In retail builds, the JIT compiler can get rather aggressive and consider a local variable dead even before it goes out of scope (assuming it is not being used). Now, this can present some really interesting challenges when debugging and the decision was therefore made to keep all local variables alive until the end of the scope in debug builds.

[lb] Stack walker comes into play when unmanaged calls are made to the execution engine. During these calls, it is imperative that any managed objects used during the call also be part of the reference tracking system.

[lb] Handle table. The CLR maintains a set of handle tables on a per application domain basis that can contain, for example, pointers to pinned reference types on the managed heap. During a GC inquiry, these handle tables are probed for live references to objects on the managed heap.

[lb] Finalize queue. We will discuss the notion of object finalizers shortly, but for the time being view objects with finalizers as objects that can be considered dead from an applications perspective but still need to be kept alive for cleanup purposes.

[lb] As a member of any of the above categories

During the probing phase above, the GC also marks all the objects according to their state (rooted). Once all components have been probed, the GC goes ahead and starts the garbage collection of all objects by promoting all objects that are still considered rooted. An interesting question in regards to roots is: given an address to an object on the managed heap, is it possible to see if the object is rooted or not and if so, what the reference chain of object is? Again, we turn to the SOS extension and a command named `gcroot`. The `gcroot` command uses a technique similar to the one above utilized by the GC to find the aliveness of the object. Let's take a look at some sample code. Listing 5-3 shows the source code of an application that defines a set of types and references to those types at various scopes.

Listing 5-3

Sample application to illustrate object roots

```
using System;
using System.Text;
using System.Threading;

namespace Advanced.NET.Debugging.Chapter5
{
    class Name
    {
        private string first;
        private string last;

        public string First { get { return first; } }
        public string Last { get { return last; } }
        public Name(string f, string l)
        {
            first = f; last = l;
        }
    }

    class Roots
    {
        public static Name CompleteName = new Name ("First", "Last");

        private Thread thread;
        private bool shouldExit;

        static void Main(string[] args)
        {
            Roots r = new Roots();
            r.Run();
        }

        public void Run()
        {
            shouldExit = false;

            Name n1 = CompleteName;

            thread = new Thread(this.Worker);
            thread.Start(n1);

            Thread.Sleep(1000);

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();

            shouldExit = true;

        }

        public void Worker(Object o)
        {
            Name n1 = (Name)o;
            Console.WriteLine("Thread started {0}, {1}",
                              n1.First,
                              n1.Last);

            while (true)
            {
                // Do work
                Thread.Sleep(500);
                if (shouldExit)
                    break;
            }
        }
    }
}
```

The source code and binary for Listing 5-3 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\Roots`

[lb] Binary: `C:\ADNDBin\05Roots.exe`

The source code in Listing 5-3 declares a static instance of the `Name` type. The main part of the application declares a reference to the static instance in the `Run` method as well as starting up a thread passing the reference to the newly created thread. The method that the new thread executes uses the reference passed to it until the user hits any key, at which point both the worker thread and the application terminate. The object we are interested in tracking for this exercise is the `CompleteName` static field. From the source code we can glean the following characteristics about `CompleteName`:

[lb] We have a static reference to the object instance at the `Roots` class level serving as our first root to the object.

[lb] In the `Run` method, we assign a local variable reference (`n1`) to the object instance serving as our second root. The `n1` local variable is not used once the thread has started and is subject to becoming invalid even before the end of the method scope (in retail builds). In debug builds, the reference is guaranteed to remain valid until the end of the scope is reached.

[lb] In the `Run` method, we pass the local variable reference `n1` to the thread method during thread startup serving as our third root.

Let's run the application under the debugger and manually break execution when the `Press any key to exit` prompt is displayed. The first thing we need to find is the address to the object we are interested in (and dumping the object for good measure) followed by running the `gcroot` command on the address:

```
0:005> ~0s
eax=002cef9c ebx=002cef94 ecx=792274ec edx=79ec9058 esi=002cedf0
edi=00000000
eip=77709a94 esp=002ceda0 ebp=002cedc0 iopl=0         nv up ei pl zr na
pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000246
ntdll!KiFastSystemCallRet:
77709a94 c3              ret
0:000> !ClrStack -a
OS Thread Id: 0x2358 (0)
ESP       EIP
002cef6c 77709a94 [NDirectMethodFrameSlim: 002cef6c]
Microsoft.Win32.Win32Native.ReadConsoleInput(IntPtr, InputRecord ByRef,
Int32, Int32 ByRef)
002cef84 793e8f28 System.Console.ReadKey(Boolean)
    PARAMETERS:
        intercept = 0x00000000
    LOCALS:
        <no data>
        0x002cef94 = 0x00000001
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>
        <no data>

002cefc4 793e8e33 System.Console.ReadKey()
002cefc8 00890212 Advanced.NET.Debugging.Chapter5.Roots.Run()
    PARAMETERS:
        this = 0x01c758e0
    LOCALS:
        <CLR reg> = 0x01c758d0

002cefe8 0089013f
Advanced.NET.Debugging.Chapter5.Roots.Main(System.String[])
    PARAMETERS:
        args = 0x01c75888
    LOCALS:
        <CLR reg> = 0x01c758e0

002cf208 79e7c74b [GCFrame: 002cf208]
0:000> !do 0x01c758d0
Name: Advanced.NET.Debugging.Chapter5.Name
MethodTable: 001b311c
EEClass: 001b13a0
Size: 16(0x10) bytes
 (C:\ADNDBin\05Roots.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value
Name
790fd8c4  4000001        4        System.String  0 instance 01c75898
first
790fd8c4  4000002        8        System.String  0 instance 01c758b4
last
0:000> !gcroot 0x01c758d0
Note: Roots found on stacks may be false positives. Run "!help gcroot"
for
more info.
Scan Thread 0 OSTHread 2358
ESP:2cefbc:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
Scan Thread 1 OSTHread 1630
Scan Thread 3 OSTHread 254c
ESP:47df428:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df42c:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df438:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df4d0:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df4d8:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df4f4:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df500:Root:01c75984(System.Threading.ThreadHelper)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df5c0:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
ESP:47df5c4:Root:01c75998(System.Threading.ParameterizedThreadStart)->
01c75984(System.Threading.ThreadHelper)
ESP:47df754:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)->
01c75984(System.Threading.ThreadHelper)
ESP:47df758:Root:01c75998(System.Threading.ParameterizedThreadStart)->
01c75984(System.Threading.ThreadHelper)
ESP:47df764:Root:01c75998(System.Threading.ParameterizedThreadStart)->
01c75984(System.Threading.ThreadHelper)
ESP:47df76c:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)->
01c75984(System.Threading.ThreadHelper)
DOMAIN(0037FCF8):HANDLE(Pinned):a13fc:Root:02c71010(System.Object[])->
01c758d0(Advanced.NET.Debugging.Chapter5.Name)
```

As you can see from the `gcroot` output, the command scans a number of different sources to find and build the reference chain to the object specified. Regardless of the source, the output of the `GCRoot` command results in the following general format:

```
<root>-><reference 1>-><reference 2>-><reference X>-><object>
```

Depending on the source probed, each of the elements will take on a slightly different format as shown below.

[lb] Local variables on a threads stack. The `root` element will typically look like the following: `<stack register>:<stack pointer>:Root:<object>.` The `stack register` depends on the architecture. For example, on x86 machines it will show as `ESP` and on x64 machines it will show as `RSP`. The `stack pointer` shows the location on the stack where the object is rooted and the `object address` is the address of the object that is holding a reference to the next object in the reference chain. Let's take a look at an example:

```
ESP:47df428:Root:01c758d0(Advanced.NET.Debugging.Chapter5.Name)
```

Here we can see that there is a local variable located on stack (`ESP`) location `0x047df428`. Furthermore, the output tells us that this constitutes a root to the object at address `0x01c758d0` which is a reference to the `Advanced.NET.Debugging.Chapter5.Name` type.

[lb] Handle tables. All handle tables are scanned as part of `GCRoot` execution looking for references to the specified object. If a reference is found, the output of the command takes on the following general syntax: `DOMAIN(<address>):HANDLE(<type>):<handle address>:Root:<object>`. The `domain address` field indicates the address of the application domain to which the handle reference belongs. The `handle type` specifies the type of the handle. The possible handle types are: Weak, WeakTracResurrection, Normal, and Pinned.

Next is the `handle address,` which is the address to the handle itself. Please keep in mind that the handle type is a value type and if you want to dump out the contents you must use the `DumpVC` command rather than `DumpObj`. Finally, the `root object` address is shown. Let's take a look at an example:

```
DOMAIN(002EFCD8):HANDLE(Pinned):2813fc:Root:02c81010(System.Object
[])->01c858d0(Advanced.NET.Debugging.Chapter5.Name)
```

The output above indicates that the object at address `0x01c858d0` is rooted by an object that resides in the handle table corresponding to the application domain with address `0x002efcd8`. Furthermore, the address of the handle value holding the reference is located at address `0x002813fc` and the type of the handle value is pinned. Lastly, the actual object that holds the reference is at address `0x02c81010`, which is of type `System.Object[]`.

[lb] F-reachable queue. The f-reachable queue is scanned to see if there are any references to the specified object. If a root reference to the object is found on the f-reachable queue it will be displayed in the following general format: `Finalizer queue:Root:<object address>(<object type>)`. The first part of the output indicates that the source of the root is the f-reachable queue.

Next, the address of the referenced object is displayed followed by the object type. What follows is an example of the output of `GCRoot` when run against an object that is on the f-reachable queue:

```
Finalizer
queue:Root:01d15750(Advanced.NET.Debugging.Chapter5.Name)
```

In the above output, we can see that the object at address `0x01d15750` of type `Advanced.NET.Debugging.Chapter5.Name` is rooted by the f-reachable queue.

[lb] The last source of output for the GCRoot command is if an object is a member of any of the categories above.

One of the potential problems with `gcroot` and local variables is that it may not always be accurate, thereby producing false positives. In order to convince ourselves that the stack locations listed in the output are accurate we have to manually inspect the stack location and correlate it to source code so that we can see whether or not the local variable is in fact still referencing the object. For example, assume we have the following very simple function:

```
        public void Run()
        {
            Name n1 = new Name("A", "B");

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
     }
```

In the source code above, we have a simple instance of the Name class assigned to the `n1` local variable. If we were to run the `GCRoot` command on the `n1` reference we would expect to only see one reference on the thread stack:

```
0:000> !GCRoot 0x01e9580c
Note: Roots found on stacks may be false positives. Run "!help gcroot"
for
more info.
Scan Thread 0 OSTHread 1638
ESP:1df29c:Root:01e9580c(Advanced.NET.Debugging.Chapter5.Name)
ESP:1df2a0:Root:01e9580c(Advanced.NET.Debugging.Chapter5.Name)
Scan Thread 2 OSTHread 14ac
```

The output clearly shows that thread 0 apparently has two references to the object on the thread stack. How is this possible? The way that the `GCRoot` command works is by assuming that *every* address on the stack is an address to an object. It tries to verify this assumption by utilizing various metadata information. In light of this, objects that are (or were) previously present on the stack will be treated as first class references to those objects and listed in the output of `GCRoot`. If you suspect that the output of `GCRoot`, in as far as thread stacks is concerned, is incorrect the best approach is to use the `U` command to unassemble the stack frames and correlate the stack registers in the `GCRoots` output to the unassembled code to see which objects are truly valid.

Finalization

The garbage collection mechanism described so far assumes that objects that are collected do not require any special clean up code. At times, objects that encapsulate other resources require that these resources be cleaned up as part of object destruction. A great example is an object that wraps an underlying native resource such as a file handle. Without explicit cleanup code, the memory behind the managed object will be cleaned up by the GC but the underlying handle that the object encapsulates will not (since GC has no special knowledge of native handles). The net result is naturally a resource leak. To provide a proper cleanup mechanism, the CLR introduces what is known as finalizers. A finalizer can be compared to destructors in the native C++ world. Whenever an object is freed (or garbage collected) the destructor (or finalizer) is run. In C#, a finalizer is declared very similarly to a C++ destructor by using the `~<class name>()` notation. An example is shown in the following listing:

```
public class MyClass
{
      …
      …
      …
      ~MyClass()
      {
            // Cleanup code
      }
}
```

When the class is compiled into IL, the finalize method gets translated into a function called `Finalize`. The key thing about objects with finalizers is that the garbage collector treats them a little differently than other objects. Since the garbage collector is in fact an *automatic* memory manager it also has the responsibility of executing all finalization code that an object may have during a garbage collection. In order to keep tabs on which objects have finalizers the garbage collector maintains a queue called a finalization queue. Objects that are created on the managed heap and contain finalizers are automatically placed on the finalization queue during creation. Please note that the finalization queue does *not* contain objects that are considered garbage, but rather it contains all objects with finalizers that are alive on the managed heap. Once an object with a finalizer becomes rootless and a garbage collection occurs, the GC places the object on a different queue known as the f-reachable queue. This queue contains all objects with defined finalizers that are considered to be garbage and need to have their finalizers executed. All objects on the f-reachable queue are considered roots to those objects meaning that the object is still alive. It is important to note that the finalizer code for each of the objects on the f-reachable queue is not executed as part of the garbage collection phase. Instead, each .NET process contains a special thread known as the finalization thread. The finalization thread wakes up, on request of the GC, and checks the state of the f-reachable queue. If there are any objects on the f-reachable queue, the finalization thread picks them up one by one and executes the finalize methods.

Why not execute the finalize methods as part of the garbage collection?

Since finalize methods contains managed code and during a GC managed code threads are suspended, the finalizer thread runs outside of the boundary of the GC.

Once the garbage collection finishes, objects with finalizers will be on the f-reachable queue (rooted and alive) until the finalization thread executes the finalize methods. At that point, the object is removed from the f-reachable queue and is now considered rootless and can be truly reclaimed by the garbage collector. The next time a garbage collection is started, the objects will be collected. Figure 5-7 illustrates an example of the finalization process.

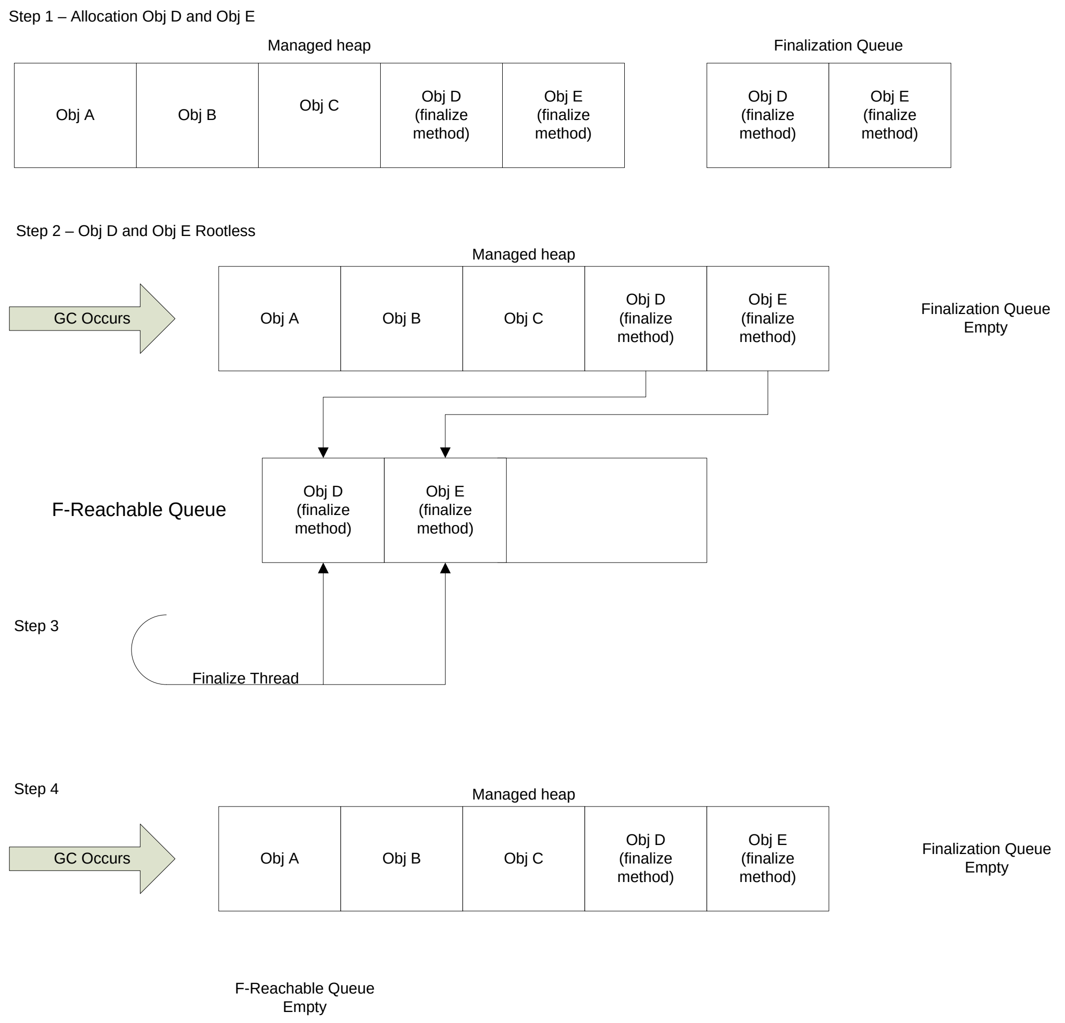


Figure 5-7
Example of Finalization process

Step 1 in Figure 5-7 consists of allocating `Obj D` and `Obj E` both of which contain finalize methods. As part of the allocation, the objects are placed on the managed heap as well as on the finalization queue to indicate that the objects will need to be finalized when no longer in use. In step 2, `Obj D` and `Obj E` have both become rootless when a garbage collection occurs. At that point, both objects are moved from the finalization queue to the f-reachable queue to indicate that the finalize methods are now ready to be run. At some point in the future (non deterministic) step 3 is executed and the finalizer thread wakes up and starts running the finalize methods for both of the objects. Even after the finalizer has finished, both objects are still rooted on the f-reachable queue. Lastly, in step 4, another garbage collection occurs and the objects are removed from the f-reachable queue (no longer rooted) and then collected from the managed heap by the garbage collector.

An interesting aspect of having a dedicated thread executing the finalize methods is that the CLR does not place any guarantees when the thread wakes up and executes. As such it is possible that it will take some time before an object with a finalizer is actually cleaned up. When dealing with objects that aggregate scarce resources it may not always be feasible to wait for a long period of time for the resource to be reclaimed. In such situations, it is best to implement an explicit and deterministic cleanup pattern such as the `IDisposable` and/or `Close` patterns. Finally, having a dedicated thread also means that you have no control over the state of that thread and making assumptions based on state can break your application.

Let's take a look at a concrete example of an object with a finalize method and see if we can track the object during a garbage collection. Listing 5-4 shows the source code of the application we will be utilizing.

Listing 5-4

Simple object with a finalize method

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class NativeEvent
    {
        private IntPtr nativeHandle;

        public IntPtr NativeHandle { get { return nativeHandle; } }

        public NativeEvent(string name)
        {
            nativeHandle = CreateEvent(IntPtr.Zero,
                                       false,
                                       true,
                                       name);
        }

        ~NativeEvent()
        {
            if(nativeHandle!=IntPtr.Zero)
            {
                CloseHandle(nativeHandle);
                nativeHandle=IntPtr.Zero;
            }
        }

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateEvent(IntPtr lpEventAttributes,
                                         bool bManualReset,
                                         bool bInitialState,
                                         string lpName);

        [DllImport("kernel32.dll")]
        static extern IntPtr CloseHandle(IntPtr lpEvent);
    }
    class Finalize
    {
        static void Main(string[] args)
        {
            Finalize f = new Finalize();
            f.Run();
        }

        public void Run()
        {
            NativeEvent nEvent = new NativeEvent("MyNewEvent");

            //
            // Use nEvent
            //

            nEvent = null;

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

    }
}
```

The source code and binary for Listing 5-4 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\Finalize`

[lb] Binary: `C:\ADNDBin\05Finalize.exe`

The source code in Listing 5-4 declares a type called `NativeEvent` that simply wraps the creation of a Windows event using the .NET interoperability services. Since the net result of creating a native event is a handle, the handle must be closed during object destruction to avoid a handle leak in the application. The closing of the handle is implemented in the `NativeEvent` finalize method. The main part of the application is implemented in the `Finalize` class. More specifically, the `Run` method declares an instance of the `NativeEvent` class, sets the local variable reference to `null` (indicating that it can be garbage collected), followed by a couple of forced garbage collections. What do we expect to happen to the `NativeEvent` instance we declared at the point of the first garbage collection? From our previous discussion, we expect that prior to the garbage collection, the object is in the finalization queue. Furthermore, when the garbage collection occurs, the object is deemed rootless and moved to the f-reachable queue where it maintains a reference to the object so that the finalization thread can run the `Finalize` method. It's important to remember that the execution of the finalization thread does not happen during the garbage collection, but rather it happens out of band at any time. Once the `Finalize` method has run, the object can be fully collected during the next garbage collection. Let's see if we can use the debuggers to verify our theory above. Run `05Finalize.exe` under the debugger and break execution once the first `Press any key to GC` prompt appears. Once broken into the debugger, we can use the `FinalizeQueue` command to show the state of the finalizable objects in the process:

```
0:004> !FinalizeQueue
SyncBlocks to be cleaned up: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
----------------------------------
generation 0 has 6 finalizable objects (003d3160->003d3178)
generation 1 has 0 finalizable objects (003d3160->003d3160)
generation 2 has 0 finalizable objects (003d3160->003d3160)
Ready for finalization 0 objects (003d3178->003d3178)
Statistics:
      MT    Count    TotalSize Class Name
00123128        1           12
Advanced.NET.Debugging.Chapter5.NativeEvent
7911c9c8        1           20
Microsoft.Win32.SafeHandles.SafePEFileHandle
791037c0        1           20
Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20
Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
79101444        1           20
Microsoft.Win32.SafeHandles.SafeFileHandle
790fe704        1           56 System.Threading.Thread
Total 6 objects
```

There are several pieces of useful information in the output. First, the finalization queues for each generation are shown. In this particular case, generation 0 has 6 finalizable objects and generations 1 and 2 have none. For each of the finalization queues, the `FinalizeQueue` command also shows the address range of the queue itself for that particular generation. For example, generation 0's finalization queue starts at address `0x003d3160` and ends at address `0x003d3178`. We can use the `dd` command to dump the queue as shown below:

```
0:004> dd 003d3160 l6
003d3160  01fc1df0 01fc5090 01fc5964 01fc5998
003d3170  01fc683c 01fc6850
```

The elements in the queue can be looked at further by using the `do` command. If we wanted to look at the object at address `0x01fc5964` in more detail we would use the command shown below:

```
0:004> !do 01fc5964
Name: Advanced.NET.Debugging.Chapter5.NativeEvent
MethodTable: 00123128
EEClass: 00121804
Size: 12(0xc) bytes
 (C:\ADNDBin\05Finalize.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value
Name
791016bc  4000001        4        System.IntPtr  1 instance      1f0
nativeHandle
```

The next piece of useful information from the `FinalizeQueue` command is the f-reachable queue which is shown in the following output:

```
Ready for finalization 0 objects (000c3178->000c3178)
```

The output indicates that at this point there are no objects that are ready to be finalized. This makes perfect sense since a garbage collection has not yet occurred.

The final piece of output in the `FinalizeQueue` command is the statistics section which shows a summarized list of all objects in either the finalization queue or the f-reachable queue.

Before we resume execution we need to discuss the magic finalization thread that exists in all managed processes. What does the stack trace of this thread look like? To find the answer, use the `~*kn` command to display the stack traces of all the threads in the process including frame numbers. In the output, one thread in particular looks interesting:

```
      2  Id: 1a10.c10 Suspend: 1 Teb: 7ffdd000 Unfrozen
 # ChildEBP RetAddr
00 011cf604 77709254 ntdll!KiFastSystemCallRet
01 011cf608 7618c244 ntdll!ZwWaitForSingleObject+0xc
02 011cf678 79e789c6 KERNEL32!WaitForSingleObjectEx+0xbe
03 011cf6bc 79e7898f mscorwks!PEImage::LoadImage+0x1af
04 011cf70c 79e78944 mscorwks!CLREvent::WaitEx+0x117
05 011cf720 79ef2220 mscorwks!CLREvent::Wait+0x17
06 011cf73c 79fb997b mscorwks!WKS::WaitForFinalizerEvent+0x4a
07 011cf750 79ef3207 mscorwks!WKS::GCHeap::FinalizerThreadWorker+0x79
08 011cf764 79ef31a3 mscorwks!Thread::DoADCallBack+0x32a
09 011cf7f8 79ef30c3 mscorwks!Thread::ShouldChangeAbortToUnload+0xe3
0a 011cf834 79fb9643 mscorwks!Thread::ShouldChangeAbortToUnload+0x30a
0b 011cf85c 79fb960d mscorwks!ManagedThreadBase_NoADTransition+0x32
0c 011cf86c 79fba09b mscorwks!ManagedThreadBase::FinalizerBase+0xd
0d 011cf8a4 79f95a2e mscorwks!WKS::GCHeap::FinalizerThreadStart+0xbb
0e 011cf93c 76184911 mscorwks!Thread::intermediateThreadProc+0x49
0f 011cf948 776ee4b6 KERNEL32!BaseThreadInitThunk+0xe
10 011cf988 776ee489 ntdll!__RtlUserThreadStart+0x23
11 011cf9a0 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Frames 6 and 7 in the stack trace indicate that this is in fact the finalizer thread for the process. Frame 6 in particular shows that the thread is currently waiting for finalizer events (or objects that need to be finalized). Let's set a breakpoint on the return address of frame 6 (`0x79fb997b`) which will trigger anytime the finalizer thread is woken up to perform work:

```
bp 79fb997b
```

Once the breakpoint is set, resume execution and press any key to trigger the first garbage collection. You'll notice that a breakpoint is hit as shown in the following:

```
0:003> g
 Breakpoint 0 hit
eax=00000001 ebx=00000001 ecx=7618c42d edx=77709a94 esi=00000000
edi=00493a48
eip=79fb997b esp=00b7f768 ebp=00b7f770 iopl=0         nv up ei pl nz na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000202
mscorwks!WKS::GCHeap::FinalizerThreadWorker+0x79:
```

```
79fb997b 3bde            cmp     ebx,esi
```

The breakpoint corresponds to the finalizer thread breakpoint set earlier and indicates that the finalizer is ready to execute the Finalize methods on the objects in the f-reachable queue. How do we find out what objects are in the f-reachable queue? You guessed it; by using the `FinalizeQueue` command:

```
0:002> !FinalizeQueue
SyncBlocks to be cleaned up: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
----------------------------------
generation 0 has 0 finalizable objects (003d3170->003d3170)
generation 1 has 4 finalizable objects (003d3160->003d3170)
generation 2 has 0 finalizable objects (003d3160->003d3160)
Ready for finalization 2 objects (003d3170->003d3178)
Statistics:
      MT    Count    TotalSize Class Name
00123128        1           12
Advanced.NET.Debugging.Chapter5.NativeEvent
7911c9c8        1           20
Microsoft.Win32.SafeHandles.SafePEFileHandle
791037c0        1           20
Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20
Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
79101444        1           20
Microsoft.Win32.SafeHandles.SafeFileHandle
790fe704        1           56 System.Threading.Thread
```

This time the output states that there are two objects in the f-reachable queue, starting at address `0x003d3160,` that the finalization thread is about to execute. If we dump out the contents of the f-reachable queue and each of the objects we can see the following:

```
0:002> dd 003d3170 l2
003d3170  01fc5090 01fc5964
0:002> !do 01fc5090
Name: Microsoft.Win32.SafeHandles.SafePEFileHandle
MethodTable: 7911c9c8
EEClass: 791fb61c
Size: 20(0x14) bytes

(C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.
dll)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value
Name
791016bc  40005c1        4        System.IntPtr  1 instance   3eab28
handle
79102290  40005c2        8         System.Int32  1 instance        4
_state
7910be50  40005c3        c       System.Boolean  1 instance        1
_ownsHandle
7910be50  40005c4        d       System.Boolean  1 instance        1
_fullyInitialized
0:002> !do 01fc5964
Name: Advanced.NET.Debugging.Chapter5.NativeEvent
```

```
MethodTable: 00123128
EEClass: 00121804
Size: 12(0xc) bytes
 (C:\ADNDBin\05Finalize.exe)
Fields:
      MT    Field   Offset                 Type VT     Attr    Value
Name
791016bc  4000001        4        System.IntPtr  1 instance      1f0
nativeHandle
```

The first object is of type `SafePEFileHandle` and the second object is of type `NativeEvent`, which happens to be the object we are interested in. If we resume execution, the finalizer thread will execute the Finalize method of our `NativeEvent` class. What happens to the objects on the f-reachable queue once finalization has completed? Well, the objects are removed from the f-reachable queue which renders them rootless and they will be collected during the next garbage collection.

This concludes our discussion of finalization. As you can see there is a lot of work being done under the hood whenever a finalizable type comes into play. Not only does the CLR need additional data structures (such as the finalization queue and f-reachable queue), but it also spins up a dedicated thread to run the `Finalize` methods for each object that is being collected. Furthermore, an object with a `Finalize` does not get collected in just one garbage collection, but rather two, which in essence means that the objects with `Finalize` methods always get promoted to generation 1 before they are truly dead, making it a far more expensive object to work with.

Reclaiming GC Memory

So far we have discussed the GC in quite a bit of detail. We now know exactly what the GC does when an object is considered garbage. The one missing piece of information is what the GC does with the memory that becomes available once an object is garbage collected. Does the memory get put on some sort of free list and then re-used once another allocation request arrives? Does the memory get freed? Is fragmentation ever a problem on the managed heap? The answer is a combination of all three. If a collection that occurs in generations 0 and 1 leaves gap in on the managed heap, the garbage collector compacts all live objects so that they reside next to each other and coalesces any free blocks on the managed heap into a larger block that is located after the last live object (starting at the current allocation pointer). Figure 5-8 shows an example of the compaction and coalescing.

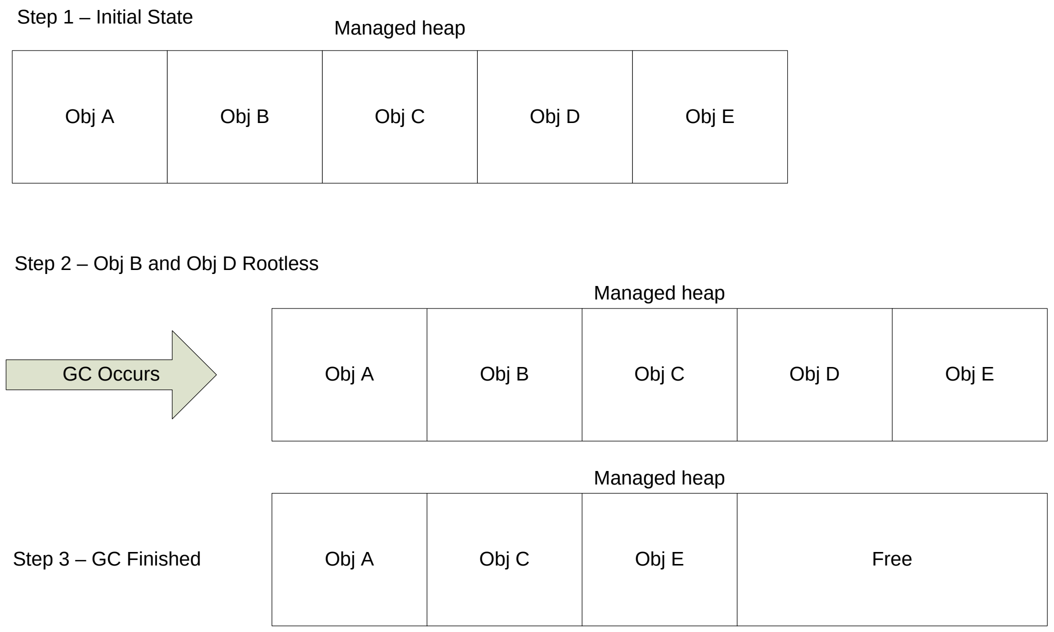


Figure 5-8
Garbage collection compaction and coalescing phase

In Figure 5-8, the initial state of the managed heap contains five rooted objects (A-E). At some point during execution, objects B and D become rootless and are candidates to be reclaimed during a garbage collection. Once the garbage collection occurs, the memory occupied by objects B and D is reclaimed, which leads to gaps on the managed heap. To remove these gaps, the garbage collector compacts the remaining live objects (Obj A, C and E) and coalesces the two free blocks (used to hold Obj B and D) into one free block. Lastly, the current allocation pointer is updated as a result of the compaction and coalescing.

The ephemeral segment contains both generation 0 and generation 1 (and also part of generation 2), but generation 2 can consist of multiple managed heap segments. As more and more objects make it to generation 2 the need to grow generation 2 also increases. The way that the CLR heap manager grows generation 2 is by allocating more segments. Once objects in generation 2 are collected, the CLR heap manager decommits memory in the segments, and when a segment is no longer needed, it is entirely freed. In certain situations and allocation patterns, generation 2 grows and shrinks quite frequently, leading to a large number of calls to allocate and free virtual memory (`VirtualAlloc` and `VirtualFree` APIs). These calls can be expensive as a transition to kernel mode is required as well as the potential to fragment the VM address space. As such, CLR 2.0 introduces a feature called VM Hoarding, which essentially does not free segments but rather keeps the segments on a standby list that can be utilized when more memory is required. In order to utilize the VM hoarding feature, the CLR host itself must specify that it wishes to use the feature.

Full Verus Partial Garbage Collection

A garbage collection that collects all three generations due to breaching all three generational thresholds is known as a full garbage collection. In contrast, garbage collection in only generation 0 or generation 0 and 1 is simply known as a garbage collection.

Since the cost of a compaction is directly proportional to the size of the object (the bigger the object, the costlier the compaction), the garbage collector introduces another type of heap called the Large Object Heap (LOH). Objects that are large enough to severely hurt the performance of a compaction are placed on the LOH which we will discuss next.

Large Object Heap

The Large Object Heap (LOH) consists of objects that are greater than or equal to 85,000 bytes in size. The decision to separate objects of that size into its own heap is related to the fact that during the compacting phase of a garbage collection the cost of compacting an object is directly proportional to the size of the object being compacted. Rather than having large objects on the standard heap eating up garbage collection time during compaction, the LOH was created. The LOH is best viewed as an extension of generation 2, and a collection of the LOH can only be done after a generation 2 collection has occurred, implying that a collection of the LOH is only done during a full garbage collection. Since compacting large objects is very expensive, the GC avoids compacting the LOH all together and instead uses a process known as sweeping that keeps a free list that is used to keep track of available memory in the LOH segment(s). Figure 5-9 shows an example of a LOH with two segments.

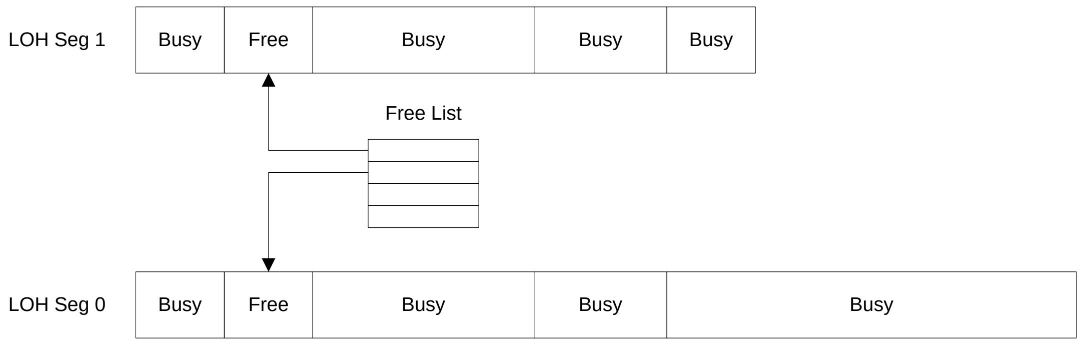


Figure 5-9
LOH Example

Please note that while the LOH does not perform any compaction it does do coalescing of adjacent free blocks. That is, if you ever end up with two free adjacent blocks, the GC coalesces those blocks into a larger block and adds it to the free list (while also removing the two smaller blocks).

To find out the current state of the LOH in the debugger we can once again use the `eeheap –gc` command, which include details on the LOH:

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x01fc6c18
generation 1 starts at 0x01fc100c
generation 2 starts at 0x01fc1000
ephemeral segment allocation context: none
 segment    begin allocated     size
00308030 790d8620  790f7d8c 0x0001f76c(128876)
01fc0000 01fc1000  01fc8c24 0x00007c24(31780)
Large object heap starts at 0x02fc1000
 segment    begin allocated     size
02fc0000 02fc1000  02fc3240 0x00002240(8768)
Total Size   0x295d0(169424)
------------------------------
```

```
GC Heap Size   0x295d0(169424)
```

The LOH section in the command output shows the starting point of the LOH as well as per segment information such as the segment, start, and end address of the segment and total size of the segment. In the example above we can see that the LOH has one segment (`0x02fc000`) starting at address `0x02fc1000` and ending at `0x02fc3240` with a total size of `0x00002240`. The last piece of information is the total size of all segments in the LOH. One interesting question related to the LOH is how the contents of the LOH can be dumped. There are a couple of options which both revolve around using `DumpHeap` command switches. The first switch of interest is the `–min` switch which tells the `DumpHeap` command that you are only interested in objects of the specified size. Since we know that LOH objects are greater than or equal to 85,000 in size we can use the following command:

```
0:004> !DumpHeap -min 85000
 Address       MT     Size
02c53250 7912dae8   100016
total 1 objects
Statistics:
      MT    Count    TotalSize Class Name
7912dae8        1       100016 System.Byte[]
```

Here we can see that there is one object of size `100016` on the LOH. You can verify or convince yourself that the object is in fact on the LOH by looking at the address. If the address of the object falls within the LOH segments addresses it must be located on the LOH (with the exception of free objects which can reside both in the SOH as well as the LOH).

The next option we have is to specify a starting address for the `DumpHeap` command. If we specify the starting address of the LOH we can ask the command to dump out all objects on the LOH. The switch to use is the `–startAtLowerBound` switch which takes the address as a parameter. Using the same LOH as above, the following command can be used:

```
0:004> !DumpHeap -startAtLowerBound 02c51000
 Address       MT     Size
02c51000 002a6360       16 Free
02c51010 7912d8f8     4096
02c52010 002a6360       16 Free
02c52020 7912d8f8     4096
02c53020 002a6360       16 Free
02c53030 7912d8f8      528
02c53240 002a6360       16 Free
02c53250 7912dae8   100016
02c6b900 002a6360       16 Free
total 9 objects
Statistics:
      MT    Count    TotalSize Class Name
002a6360        5           80      Free
7912d8f8        3         8720 System.Object[]
7912dae8        1       100016 System.Byte[]
Total 9 objects
```

Here we once again see the object of size `100016`, but what is even more interesting is that we see objects that are *smaller than 85,000 bytes* on the LOH. What are these objects and how did they end up on the LOH? The answer is that these very, very small objects are placed there by the CLR heap manager which uses them for its own purposes. Generally speaking you will always see a select few objects with a size less than 85,000 bytes exclusively used by the GC.

Let's take a look at a small sample application that allocates a single large object of size 10000 bytes. We will then use the debuggers to see if we can locate the object on the LOH and see what happens when the object is collected.

Listing 5-5

Sample application demonstrating LOH

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
     class LOH
    {
        static void Main(string[] args)
        {
            LOH l = new LOH();
            l.Run();
        }

        public void Run()
        {
            byte[] b = null;
            Console.WriteLine("Press any key to allocate on LOH");
            Console.ReadKey();

            b = new byte[100000];

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            b = null;
            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

    }
}
```

The source code and binary for Listing 5-5 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\LOH`

[lb] Binary: `C:\ADNDBin\05LOH.exe`

Let's run the application in the debugger and break execution when the `Press any key to allocate on LOH` is displayed. At this point, we haven't yet created our big allocation but it never hurts to take a look at the LOH heap to see what, if anything, is already on it:

```
0:004> !eeheap -gc
Number of GC Heaps: 1
```

```
generation 0 starts at 0x01f01018
generation 1 starts at 0x01f0100c
generation 2 starts at 0x01f01000
ephemeral segment allocation context: none
 segment    begin allocated     size
004a8008 790d8620  790f7d8c 0x0001f76c(128876)
01f00000 01f01000  01f5c334 0x0005b334(373556)
Large object heap starts at 0x02f01000
 segment    begin allocated     size
02f00000 02f01000  02f03250 0x00002250(8784)
Total Size   0x7ccf0(511216)
------------------------------
GC Heap Size   0x7ccf0(511216)
0:004> !dumpheap -startatlowerbound 02f01000
 Address       MT     Size
02f01000 00496360       16 Free
02f01010 7912d8f8     4096
02f02010 00496360       16 Free
02f02020 7912d8f8     4096
02f03020 00496360       16 Free
02f03030 7912d8f8      528
02f03240 00496360       16 Free
total 7 objects
Statistics:
      MT    Count    TotalSize Class Name
00496360        4           64      Free
7912d8f8        3         8720 System.Object[]
Total 7 objects
```

We start by finding the starting point of the LOH by using the `eeheap` command. The starting point in this case is `0x02f01000`. We then feed the starting address to the `dumpheap` command using the `–startatlowerbound` switch to output all objects on the LOH. In the output we can see that the only objects that are on the LOH are the mysterious object arrays that are smaller than 85,000 bytes. Other than that, we have no other objects present. Next, resume execution and once again manually break execution once the `Press any key to GC` is shown. We issue the same dumpheap command as before to see if we can spot our 100KB allocation:

```
0:003> !dumpheap -startatlowerbound 02f01000
 Address       MT     Size
02f01000 00496360       16 Free
02f01010 7912d8f8     4096
02f02010 00496360       16 Free
02f02020 7912d8f8     4096
02f03020 00496360       16 Free
02f03030 7912d8f8      528
02f03240 00496360       16 Free
02f03250 7912dae8   100016
02f1b900 00496360       16 Free
total 9 objects
Statistics:
      MT    Count    TotalSize Class Name
00496360        5           80      Free
7912d8f8        3         8720 System.Object[]
7912dae8        1       100016 System.Byte[]
Total 9 objects
```

Here we can see that our allocation is stored at address `0x02f03250` on the LOH. Next, we once again resume execution until we see the `Press any key to exit` prompt. At this point a garbage collection has occurred so let's see what the LOH looks like by using the same `dumpheap` command again:

```
0:003> !dumpheap -startatlowerbound 02f01000
 Address       MT     Size
02f01000 00496360       16 Free
02f01010 7912d8f8     4096
02f02010 00496360       16 Free
02f02020 7912d8f8     4096
02f03020 00496360       16 Free
02f03030 7912d8f8      528
total 6 objects
Statistics:
      MT    Count    TotalSize Class Name
00496360        3           48      Free
7912d8f8        3         8720 System.Object[]
```

This time we can see how the object has been removed from the LOH and the free blocks available as a result of the collection.

Pinning

As we saw in the Releasing GC Memory section, the garbage collector employs a technique known as compaction in order to reduce fragmentation on the GC heap. When a compaction occurs, objects may end up moving around on the heap so that they can be placed together, thereby avoiding gaps. As part of the object move, since the address of the object changes, all references to the object are also updated. This works very well assuming all references to the object are contained within the CLR, but quite often it is necessary for .NET applications to work outside of the boundary of the CLR by using the interoperability services (such as platform invocation or COM interoperability). If a reference to a managed object is passed to an underlying native API, the object might be moved while the native API is reading and/or writing to the memory causing serious problems since the CLR clearly cannot notify the native API of the address change. Figure 5-10 illustrates the problem.

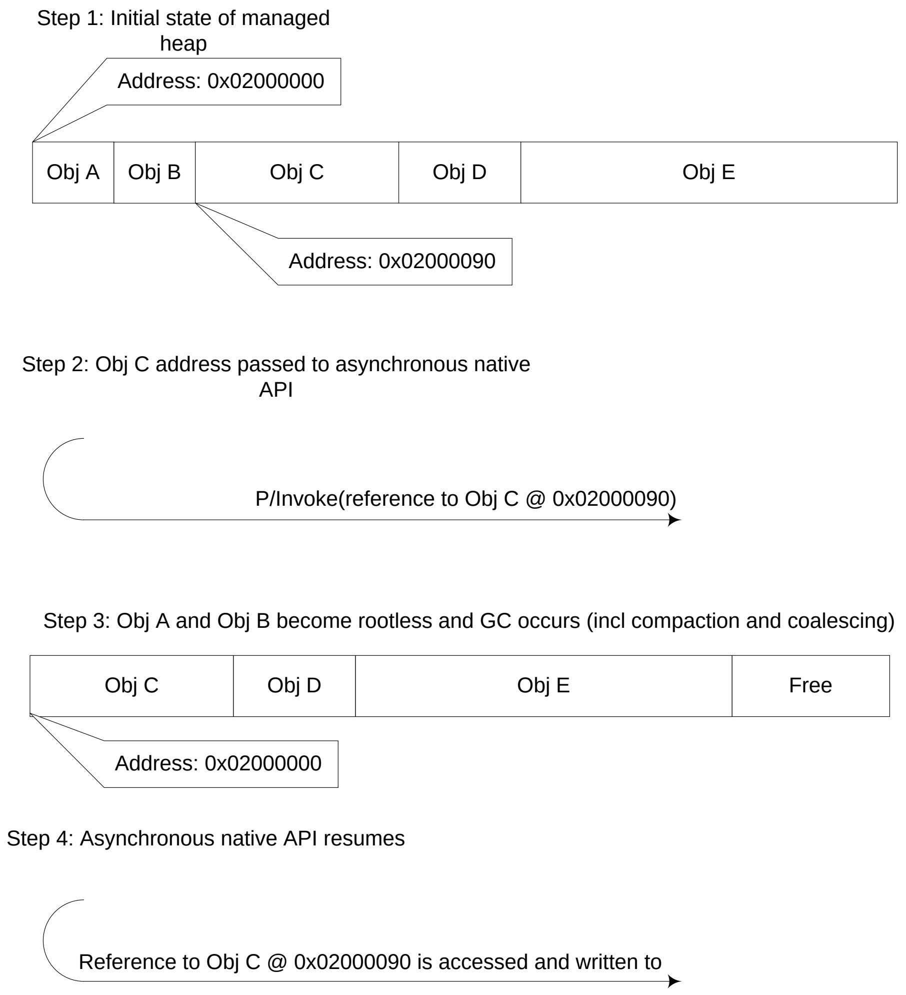

Step 1: Initial state of managed heap
Address: 0x02000000
Obj A
Obj B
Obj C
Obj D
Obj E
Address: 0x02000090
Step 2: Obj C address passed to asynchronous native API
P/Invoke(reference to Obj C @ 0x02000090)
Step 3: Obj A and Obj B become rootless and GC occurs (incl compaction and coalescing)
Obj C
Obj D
Obj E
Free
Address: 0x02000000
Step 4: Asynchronous native API resumes
Reference to Obj C @ 0x02000090 is accessed and written to

Figure 5-10
Interoperability services and GC compaction problem

From the flow in Figure 5-10 we can see that the initial state of the managed heap includes five objects starting with `Obj A` at address `0x02000000`. At a certain point, a platform invocation call to an asynchronous native API is required. Furthermore, the address of `Obj C` (`0x02000090`) needs to be passed to the API. Upon successfully calling the asynchronous native API a garbage collection occurs causing `Obj A` and `Obj B` to be collected. This leaves a gap of two free objects on the managed heap and the garbage collector dutifully rectifies the problem by compacting the managed heap and therefore moving `Obj C` to address `0x02000000`. It also coalesces the two free blocks and places them at the end of the heap. After the garbage collection has finished, the asynchronous API call we made earlier decides to write to the address initially passed to it (`0x02000090`) which originally held `Obj C`. As you can see, with the asynchronous API writing to that address we will experience a managed heap corruption as the memory is no longer occupied by `Obj C`.

Since the invocation of native code is such a common task, a solution had to be devised that allowed for safe invocation in light of a compacting garbage collector. The solution is called pinning and refers to the ability to pin specific objects on the managed heap. Once an object is pinned the garbage collector will not move the object for any reason until the object is unpinned. If `Obj C` in Figure 5-10 was pinned prior to invoking the asynchronous native API the managed heap corruption would not have occurred due to the garbage collector not moving `Obj C` during the compaction phase.

Let’s take a look at an example of a simple application that performs pinning and see what it looks like in the debugger. Listing 5-6 shows the source code of the application.

Listing 5-6

Sample application using pinning

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class Pinning
    {
        static void Main(string[] args)
        {
            Pinning p = new Pinning();
            p.Run();
        }

        public void Run()
        {
            SByte[] b1 = null;
            SByte[] b2 = null;
            SByte[] b3 = null;

            Console.WriteLine("Press any key to alloc");
            Console.ReadKey();

            b1 = new SByte[100];
            b2 = new SByte[200];
            b3 = new SByte[300];

            GCHandle h1 = GCHandle.Alloc(b1, GCHandleType.Pinned);
            GCHandle h2 = GCHandle.Alloc(b2, GCHandleType.Pinned);
            GCHandle h3 = GCHandle.Alloc(b3, GCHandleType.Pinned);

            Console.WriteLine("Press any key to GC");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();

            h1.Free(); h2.Free(); h3.Free();
        }

    }
}
```

The source code and binary for Listing 5-6 can be found in the following folders:

[lb]   Source code: `C:\ADND\Chapter5\Pinning`

[lb]	Binary: `C:\ADNDBin\05Pinning.exe`

The sample application shown in Listing 5-6 illustrates how to use the `GCHandle` type to pin objects. The `Run` method declares three arrays of the `SByte` type and creates `GCHandles` for each of the allocations specifying that the objects be pinned. The application then forces a garbage collection and exits. Let's run the application under the debugger and see if we can track the allocated memory and how it gets pinned. Resume execution of the application until you see the `Press any key to GC prompt`. At that point, we manually break execution and use a command called `GCHandles`. The `GCHandles` command displays a list of all the handles available in the process:

```
0:004> !GCHandles
GC Handle Statistics:
Strong Handles: 15
Pinned Handles: 7
Async Pinned Handles: 0
Ref Count Handles: 0
Weak Long Handles: 0
Weak Short Handles: 1
Other Handles: 0
Statistics:
      MT    Count    TotalSize Class Name
790fd0f0        1           12 System.Object
790feba4        1           28 System.SharedStatics
790fcc48        2           48 System.Reflection.Assembly
790fe17c        1           72 System.ExecutionEngineException
790fe0e0        1           72 System.StackOverflowException
790fe044        1           72 System.OutOfMemoryException
790fed00        1          100 System.AppDomain
790fe704        2          112 System.Threading.Thread
79100a18        4          144 System.Security.PermissionSet
790fe284        2          144 System.Threading.ThreadAbortException
7912ee44        3          636 System.SByte[]
7912d8f8        4         8736 System.Object[]
Total 23 objects
```

The `GCHandles` command walks the handle tables and looks for all types of different handles (strong, weak, pinned etc) and displays a summary of the results as well as a statistical section with detailed information on each type found. In the output above, we can see that we have 15 strong handles, 7 pinned handles and 1 weak short handle. In addition, in the `Statistics` section, we can see the three `SByte` arrays that we allocated and pinned. The `GCHandles` command provides a good overview of the handle activity in any given process, but if further information is required, such as the type of handle for each of the types listed in the `Statistics` section, we have to use an additional command called `objsize`. One of the functions of the `objsize` command is to output the size of the object passed in as an argument. If no arguments are specified it scans all the referenced objects in the process and outputs the size as well as other useful information:

```
0:004> !objsize
Scan Thread 0 OSTHread 2558
ESP:2fed54: sizeof(01d9599c) =          20 (        0x14) bytes
(Microsoft.Win32.SafeHandles.SafeFileHandle)
ESP:2fee18: sizeof(01d96d9c) =         312 (       0x138) bytes
(System.SByte[])
```

```
ESP:2fee20: sizeof(01d96c58) =          112 (        0x70) bytes
(System.SByte[])
ESP:2fee24: sizeof(01d96cc8) =          212 (        0xd4) bytes
(System.SByte[])
ESP:2fee30: sizeof(01d958b4) =           12 (         0xc) bytes
(Advanced.NET.Debugging.Chapter5.Pinning)
…
…
…
Scan Thread 2 OSTHread 2c80
DOMAIN(004DFD10):HANDLE(Strong):1c119c: sizeof(01d958a4) =          16
(        0x10) bytes (System.Object[])
…
…
…
DOMAIN(004DFD10):HANDLE(WeakSh):1c12fc: sizeof(01d91de8) =          56
(        0x38) bytes (System.Threading.Thread)
DOMAIN(004DFD10):HANDLE(Pinned):1c13e4: sizeof(01d96d9c) =         312
(       0x138) bytes (System.SByte[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13e8: sizeof(01d96cc8) =         212
(        0xd4) bytes (System.SByte[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13ec: sizeof(01d96c58) =         112
(        0x70) bytes (System.SByte[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13f0: sizeof(02d93030) =         708
(       0x2c4) bytes (System.Object[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13f4: sizeof(02d92020) =        4276
(      0x10b4) bytes (System.Object[])
DOMAIN(004DFD10):HANDLE(Pinned):1c13f8: sizeof(01d9118c) =          12
(         0xc) bytes (System.Object)
DOMAIN(004DFD10):HANDLE(Pinned):1c13fc: sizeof(02d91010) =       19332
(      0x4b84) bytes (System.Object[])
```

The output above has been abbreviated but clearly shows the that our `SByte` arrays have been pinned as shown by `HANDLE(Pinned)`.

While the notion of pinning objects solves the problem of movable objects during native code invocations it does present a problem to the garbage collector. and the problem is that of fragmentation (one of the problems that compaction is meant to solve to begin with). If there are a lot of interleaved pinned objects on the managed heap situations may occur where there isn't enough contiguous free space available. Figure 5-11 shows a hypothetical example of a fragmented managed heap due to excessive pinning.

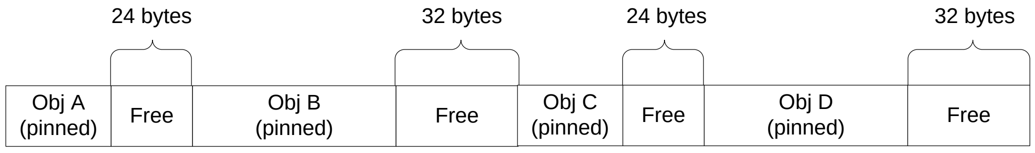


Figure 5-11
Hypothetical example of a fragmented managed heap

In the layout illustrated in Figure 5-11 we can see that we have several free smaller blocks intertwined with live objects (Obj A-D). If a garbage collection should occur, the layout of the managed heap will remain unchanged and the reason for that is simple. The garbage collector is unable to perform a compaction due to all live objects being pinned and hence not movable. Since the free blocks are not adjacent it also cannot perform coalescing. Even though we have free blocks available, memory allocation requests may in fact fail if the size of the requested allocation is greater than 32 bytes. We will take a look at a real world managed heap fragmentation problem in detailer later in the chapter.

What about the LOH?

Earlier we discussed the LOH and how the LOH is swept rather than compacted. This essentially means that objects on the LOH will never move. Does that mean that we can skip pinning objects on the LOH? The answer is a resounding no! If you don't pin objects on the LOH you are making a very dangerous implementation assumption: That the LOH will *never ever* utilize compaction. That is an implementation detail that can change between CLR versions. It is therefore imperative that objects on the LOH always be pinned in case the implementation changes.

Garbage Collection Modes

The last topic we will discuss is the notion of the mode that the garbage collector runs in. There are three primary modes of operation:

[lb] Non concurrent workstation.

[lb] Concurrent workstation

[lb] Server

We've already discussed the difference between server and workstation in general, and it boils down to the server mode creating one heap and one GC thread per processor. All garbage collection related activities are performed by the dedicated GC thread on the processor it is assigned to. What we haven't discussed is the notion of concurrent and non-concurrent garbage collections. In the non concurrent workstation mode, the garbage collector suspends all managed threads for the *entire* duration of the garbage collection. Only when the garbage collection is finished does it resume all the managed threads in the process. This may work just fine if there isn't a need for super fast responsiveness, but in cases such as GUI applications quick response times are very critical. Hence, the introduction of the concurrent workstation mode where, during a garbage collection, the managed threads are not suspended for the entire duration of the garbage collection but rather are allowed to wake up periodically and do work before being put back to sleep again for the garbage collector to do some more work. This increases the responsiveness of the application but can make garbage collection slightly slower.

Debugging Managed Heap Corruptions

A heap corruption is best defined as a bug that violates the integrity of the heap and causes strange behaviors to occur in an application. The symptoms of a heap corruption are vast and can range from subtle and random behaviors or a flat out crash that stops an application in its tracks. For example, assume an application that had an object whose state controlled the frequency with which work items were pulled from a queue. If a thread inadvertently changed the frequency due to corrupting the memory of the object, work items may be pulled of much quicker than the system can handle or, conversely, work items may not be pulled out at all, causing processing delays. In a situation like this, tracking down the culprit can be very difficult since the behavior is exhibited after the corruption has already taken place. In fact, when working with heap corruptions the best case scenario is a crash that happens as close to the source of the corruption as possible, eliminating the need for a lot of painful historic back tracking of how the heap ended up being corrupted in the first place. Due to the subtle nature of heap corruption symptoms it is also one of the trickiest categories of bugs to debug. What causes a heap corruption to occur to begin with? Generally speaking, there are probably as many different causes for heap corruptions as there are symptoms, but one very common cause is that of not properly managing the memory that the application owns. Problems such as re-use after free, dangling pointers, buffer overruns, etc. can all be possible heap corruption culprits. The good news is that the CLR eliminates many of these problems by effectively managing the memory on the applications' behalf. For example, re-use after free is no longer possible since an object won't be collected while rooted, buffer overruns are trapped and surfaced as an exception, and dangling pointers are not easily achieved. While the CLR very effectively eliminates a lot of the heap corruption culprits, it does so only when the code runs within the confines of the managed execution environment. Quite often it is necessary for a managed code application to call into native code and pass data to the native API. The second that the code transitions into the native world, the data that resides on the managed heap and which is passed to the native code is no longer under the protection of the CLR and can cause all sorts of problems unless carefully managed before making the transition. For example, buffer overruns are no longer trapped and the compacting nature of the GC can cause pointers to become stale. The managed to native code interaction is one of the biggest heap corruption culprits in the managed world.

Can there be managed heap corruptions without native code involvement?

While it is possible for a managed heap to become corrupted without any native code interactions, it is a very, very rare occurrence and usually indicates a bug in the CLR itself.

In this part of the chapter we will take a look at an example of an application that suffers from a heap corruption. Listing 5-7 illustrates the application's source code.

Listing 5-7

Example of an application that suffers from a heap corruption

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class Heap
    {
        static void Main(string[] args)
        {
            Heap h = new Heap();
            h.Run();
        }

        public void Run()
        {
            byte[] b = new byte[50];
            for (int i = 0; i < 50; i++)
```

```
                b[i] = 15;


            Console.WriteLine("Press any key to invoke native method");
            Console.ReadKey();

            InitBuffer(b, 50);

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

        [DllImport("05Native.dll")]
        static extern void InitBuffer(byte[] buffer, int size);

    }
}
```

The source code and binary for Listing 5-7 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\Heap`

[lb] Binary: `C:\ADNDBin\05Heap.exe and C:\ADNDBin\05Native.dll`

Note that the native source code is not shown so as to better illustrate the debug session.

The application in Listing 5-6 allocates a byte array (50 elements) and calls into a native API to initialize the memory by passing in the byte array as well as the size of the array. If we run the application under the debugger we can very quickly see that an access violation occurs:

```
…
…
…
Press any key to invoke native method
 ModLoad: 71190000 711ab000   C:\ADNDBin\05Native.dll
ModLoad: 63f70000 64093000
C:\Windows\WinSxS\x86_microsoft.vc90.debugcrt_1fc8b3b9a1e18e3b_9.0.21022
.8_none_96748342450f6aa2\MSVCR90D.dll
(1b00.26e4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=77767574 ebx=00000001 ecx=01c659a4 edx=01c66ad8 esi=01c66868
edi=00000017
eip=7936ab16 esp=0031edac ebp=00000017 iopl=0         nv up ei pl nz na
pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010206
*** WARNING: Unable to verify checksum for
C:\Windows\assembly\NativeImages_v2.0.50727_32\mscorlib\5b3e3b0551bcaa72
2c27dbb089c431e4\mscorlib.ni.dll
mscorlib_ni+0x2aab16:
7936ab16 ff90a4000000    call    dword ptr [eax+0A4h]
ds:0023:77767618=????????
0:000> !ClrStack
OS Thread Id: 0x26e4 (0)
ESP       EIP
0031edac 7936ab16 System.IO.StreamWriter.Flush(Boolean, Boolean)
0031edcc 7936b287 System.IO.StreamWriter.Write(Char[], Int32, Int32)
0031edec 7936b121 System.IO.TextWriter.WriteLine(System.String)
```

```
0031ee04 7936b036
System.IO.TextWriter+SyncTextWriter.WriteLine(System.String)
0031ee10 793e9d86 System.Console.WriteLine(System.String)
0031ee1c 00810171 Advanced.NET.Debugging.Chapter5.Heap.Run()
0031ee48 008100a7
Advanced.NET.Debugging.Chapter5.Heap.Main(System.String[])
0031f068 79e7c74b [GCFrame: 0031f068]
```

What is interesting about the access violation is the stack trace of the offending thread. It looks like the access violation occurred while making our second call to the `Console.WriteLine` method (right after our call to the native `InitBuffer` API). Even if we assume that a heap corruption is taking place, why is it failing in some seemingly random place in the code base? Again, it is important to remember that a heap corruption rarely breaks at the point of the corruption, but rather it breaks at some seemingly random place later in the execution flow. This would certainly qualify as random since we certainly do not expect a call to `Console.WriteLine` to ever fail with an access violation. Armed with the knowledge that an access violation has occurred and that the access violation occurred in a rather strange part of the execution flow we can now *theorize* that we have a possible heap corruption on our hands. The big question is how do we verify our theory? Remember our earlier definition of a heap corruption: a violation of the integrity of the heap. If we can walk all objects on the heap, and verify the validity of each object we can say for sure whether or not the integrity has been violated. While it's possible to walk the entire managed heap by hand it is a very time consuming process to say the least. Fortunately, the SOS `VerifyHeap` command automates this process for us. The `VerifyHeap` command walks the entire managed heap, validating each object along the way and reports the results of the validation. If we run the command in our debug session we can see the following:

```
0:000> !VerifyHeap
-verify will only produce output if there are errors in the heap
object 01c65968: does not have valid MT
curr_object : 01c65968
Last good object: 01c65928
----------------
object 02c61010: bad member 01c65968 at 02c61084
object 02c61010: bad member 01c65984 at 02c6109c
object 02c61010: bad member 01c659fc at 02c61444
object 02c61010: bad member 01c659e4 at 02c61448
object 02c61010: bad member 01c659f0 at 02c6144c
object 02c61010: bad member 01c659c8 at 02c6158c
curr_object : 02c61010
Last good object: 02c61000
----------------
```

In the output above we can see that there seems to be a number of problems with our managed heap. More specifically, the first error encountered seems to be with the object located at address `0x01c65968` not having a valid MT (method table). We can easily verify this by hand by dumping out the contents of that address using the `dd` command:

```
0:000> dd 01c65968 l1
01c65968  3b3a3938
0:000> dd 3b3a3938 l1
3b3a3938  ????????
```

The method table of the object located at address `0x01c65968` seems to be `0x3b3a3938`, which furthermore is shown to be an invalid address. At this point, we know we are working with a corrupted heap starting with an object at address `0x01c65968`, but what we don't know yet is how it got corrupted. A useful technique in situations like this is to investigate objects surrounding the corrupted memory area. For example, what does the previous object look like? The output of `VerifyHeap` shows the address of the last good object to be `0x01c65928`. If we dump out the contents of that object we can see the following:

```
0:000> !do 01c65928
Name: System.Byte[]
MethodTable: 7912dae8
EEClass: 7912dba0
Size: 62(0x3e) bytes
Array: Rank 1, Number of elements 50, Type Byte
Element Type: System.Byte
Fields:
None
0:000> !objsize 01c65928
sizeof(01c65928) =          64 (        0x40) bytes (System.Byte[])
```

The object in question appears to be a byte array with 50 elements, which also looks very similar to the byte array that we created in our application. Furthermore, since the `do` command was able to display details of the object, the object's metadata seems to be structurally intact. Please note that the `objsize` command was used to get the total size (including members of the object) of the object (`64`). The next interesting piece of information to look at is the contents of the array itself. We can use the `dd` command to display the entire object in raw memory form:

```
0:000> dd 01c65928
01c65928  7912dae8 00000032 03020100 07060504
01c65938  0b0a0908 0f0e0d0c 13121110 17161514
01c65948  1b1a1918 1f1e1d1c 23222120 27262524
01c65958  2b2a2928 2f2e2d2c 33323130 37363534
01c65968  3b3a3938 3f3e3d3c 43424140 47464544
01c65978  4b4a4948 4f4e4d4c 53525150 57565554
01c65988  5b5a5958 5f5e5d5c 63626160 67666564
01c65998  6b6a6968 6f6e6d6c 73727170 77767574
```

In the output we can see that the 64 bytes that the object occupies begin with the method table indicating the type of the array followed by the number of elements in the array followed by the array contents itself. The next object begins at address `0x01c65928` ((starting address of object)+`0x40`(total size of object)=`0x01c65968`). If we look at the contents of the last good object (`0x01c65928`) we can see that the array contains incremental integer values. Furthermore, once the end of the last good object is reached we still see a progression of the incremental integer values spilling over to what is considered the next object on the heap (`0x01c65968`). This observation yields a very important clue as to what may potentially be happening. If the object at address `0x01c65928` was incorrectly written to and allowed to write past the end of the object boundary we would corrupt the next object in the heap. Figure 5-12 illustrates the scenario.

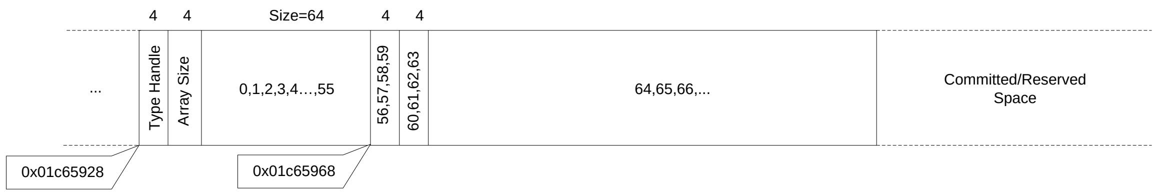


Figure 5-12
Managed heap corruption

At this point we have a pretty good understanding of the data shown to us in the debugger. By code reviewing the parts of the application that manipulate our byte array we can see that when we pass the byte array to the native `InitBuffer` API the function does not respect the boundaries of the object and writes past the end of the object causing the subsequent object on the heap to become corrupted (as output by the `VerifyHeap` command).

There is one additional piece of information that was displayed by the `VerifyHeap` command earlier:

```
object 02c61010: bad member 01c65968 at 02c61084
object 02c61010: bad member 01c65984 at 02c6109c
object 02c61010: bad member 01c659fc at 02c61444
object 02c61010: bad member 01c659e4 at 02c61448
object 02c61010: bad member 01c659f0 at 02c6144c
object 02c61010: bad member 01c659c8 at 02c6158c
curr_object : 02c61010
Last good object: 02c61000
```

What `VerifyHeap` is telling us is that there exists an object located at address `0x02c61010` that contains a member that references the corrupted object starting at address `0x01c65968`. As a matter of fact, there are multiple lines stating that the same object is referencing a number of different members of the corrupted object at various addresses (`0x01c65968, 0x01c65984, 0x01c659fc` etc). In essence, not only does `VerifyHeap` tell us which object is corrupted, but any other object on any of the heaps that references the corrupt object will also be displayed.

VerifyHeap and GC Interference

We have seen how the `VerifyHeap` command can make troubleshooting managed heap corruptions more efficient by walking the heap and reporting inconsistencies that can be a result of a heap corruption. There are times, however, when `VerifyHeap` can yield results that may not be as a result of a heap corruption. An example of that is if the CLR is in *the middle* of doing a garbage collection. During a garbage collection, the GC may end up compacting the heap, which involves moving objects around. If, for example, a move was currently in progress, the `VerifyHeap` command may very well fail or give inaccurate information due to the heap being re-organized.

One of the built in diagnostic aids that the garbage collector includes is the ability to perform heap verification before and after garbage collection occurs. To enable this diagnostics, set the environment variable `COMPLUS_HeapVerify=1`.

The sample application we used above to demonstrate how the managed heap can become corrupted was based on using the interoperability services to invoke native code. Depending on how the heap is corrupted by the native code as well as the timing of garbage collections, there may not be any signs of a heap corruption being present until much later after the native code has already done the damage, making it difficult to backtrack to the source of the problem. To aid in this troubleshooting process an MDA was added called the gcUnmanagedToManaged MDA. The MDA essentially aims at reducing the time gap between when the corruption actually occurs in native code and when the next GC occurs. The way this is accomplished is by forcing a garbage collection when the interoperability call transitions back from unmanaged to managed code thereby pinpointing the problem much earlier in the process. Let’s enable the MDA (please see Chapter 1, “Introduction to the Tools” on how to enable MDAs) and re-run our sample application under the debugger to see if we can trap the heap corruption earlier:

```
…
…
…
Press any key to invoke native method
 ModLoad: 71190000 711ab000   C:\ADNDBin\05Native.dll
ModLoad: 63f70000 64093000
C:\Windows\WinSxS\x86_microsoft.vc90.debugcrt_1fc8b3b9a1e18e3b_9.0.21022
.8_none_96748342450f6aa2\MSVCR90D.dll
(19d8.258c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=3b3a3938 ebx=02d81010 ecx=00960184 edx=01d8598c esi=00020000
edi=00001000
eip=79f66846 esp=0025ec54 ebp=0025ec74 iopl=0         nv up ei pl nz na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010202
mscorwks!WKS::gc_heap::mark_object_simple+0x16c:
79f66846 0fb708          movzx   ecx,word ptr [eax]
ds:0023:3b3a3938=????
0:000> k
ChildEBP RetAddr
0025ec74 79f66932 mscorwks!WKS::gc_heap::mark_object_simple+0x16c
0025ec88 79fbc552 mscorwks!WKS::GCHeap::Promote+0x8d
0025eca0 79fbc3c9 mscorwks!PinObject+0x10
0025ecc4 79fc37b9 mscorwks!ScanConsecutiveHandlesWithoutUserData+0x26
0025ece4 79fba942 mscorwks!BlockScanBlocksWithoutUserData+0x26
0025ed08 79fba917 mscorwks!SegmentScanByTypeMap+0x55
0025ed60 79fba807 mscorwks!TableScanHandles+0x65
0025edc8 79fbb9a2 mscorwks!HndScanHandlesForGC+0x10d
0025ee0c 79fbaaf8 mscorwks!Ref_TracePinningRoots+0x6c
0025ee30 79f669f6 mscorwks!CNameSpace::GcScanHandles+0x60
0025ee70 79f65d57 mscorwks!WKS::gc_heap::mark_phase+0xae
0025ee94 79f6614c mscorwks!WKS::gc_heap::gc1+0x62
0025eea8 79f65f5d mscorwks!WKS::gc_heap::garbage_collect+0x261
0025eed4 79f6dfa1 mscorwks!WKS::GCHeap::GarbageCollectGeneration+0x1a9
0025eee4 79f6df4b mscorwks!WKS::GCHeap::GarbageCollectTry+0x2d
0025ef04 7a0aea3d mscorwks!WKS::GCHeap::GarbageCollect+0x67
0025ef8c 7a12addd mscorwks!MdaGcUnmanagedToManaged::TriggerGC+0xa7
0025f020 79e7c74b mscorwks!FireMdaGcUnmanagedToManaged+0x3b
0025f030 79e7c6cc mscorwks!CallDescrWorker+0x33
```

```
0025f0b0 79e7c8e1 mscorwks!CallDescrWorkerWithHandler+0xa3
0:000> !ClrStack
OS Thread Id: 0x258c (0)
ESP       EIP
0025efdc 79f66846 [NDirectMethodFrameStandalone: 0025efdc]
Advanced.NET.Debugging.Chapter5.Heap.InitBuffer(Byte[], Int32)
0025efec 00a80165 Advanced.NET.Debugging.Chapter5.Heap.Run()
0025f018 00a800a7
Advanced.NET.Debugging.Chapter5.Heap.Main(System.String[])
0025f240 79e7c74b [GCFrame: 0025f240]
```

Here we can see that the native stack trace that caused the access violation looks a lot different than our earlier stack trace. It now looks like we are hitting the problem during a garbage collection. Where in our managed code flow did the garbage collection occur? If we look at the managed code stack trace we can see that we now get the access violation during our call to the native `InitBuffer` API.

If you ever suspect that a heap corruption might be taking place due to a native API invocation, enabling the gcUnmanagedtoManaged MDA can save a ton of debugging time.

Debugging Managed Heap Fragmentation

Earlier in the chapter we described a phenomenon known as heap fragmentation where free and busy blocks are arranged and interleaved on the managed heap in such a way that they can cause problems in applications that surface as OutOfMemory exceptions when, in reality, enough memory is actually free, just not in a contiguous fashion. The CLR heap manager utilizes a technique known as compaction and coalescing to reduce the risk of heap fragmentation. In this section we will take a look at an example that can cause heap fragmentation to occur and how we can use the debuggers to identify that a heap fragmentation is in fact occurring and the reasons behind it. The example is shown in Listing 5-8.

Listing 5-8

Heap fragmentation example

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace Advanced.NET.Debugging.Chapter5
{
    class Fragment
    {
        static void Main(string[] args)
        {
            Fragment f = new Fragment();
            f.Run(args);
        }

        public void Run(string[] args)
        {
            if (args.Length < 2)
            {
                Console.WriteLine("05Fragment.exe <alloc. size> <max mem
in MB>");
                return;
            }

            int size = Int32.Parse(args[0]);
            int maxmem = Int32.Parse(args[1]);
            byte[][] nonPinned = null;
            byte[][] pinned = null;
            GCHandle[] pinnedHandles = null;

            int numAllocs=maxmem*1000000/size;

            pinnedHandles = new GCHandle[numAllocs];

            pinned = new byte[numAllocs / 2][];
            nonPinned = new byte[numAllocs / 2][];

            for (int i = 0; i < numAllocs / 2; i++)
            {
                nonPinned[i] = new byte[size];
                pinned[i] = new byte[size];
    pinnedHandles[i] =
GCHandle.Alloc(pinned[i], GCHandleType.Pinned);
            }

            Console.WriteLine("Press any key to GC & promo to gen1");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to GC  & promo to gen2");
            Console.ReadKey();

            GC.Collect();

            Console.WriteLine("Press any key to GC(free non pinned");
            Console.ReadKey();

            for (int i = 0; i < numAllocs / 2; i++)
            {
                nonPinned[i] = null;
            }

            GC.Collect();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

The source code and binary for Listing 5-8 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\Fragment`

[lb] Binary: `C:\ADNDBin\05Fragment.exe`

The application enables the user to specify an allocation size and the maximum amount of memory that the application should consume. For example, if we wanted the allocation size to be

50000 bytes and the overall memory consumption limit to be 100Mb we would run the application as following:

```
C:\ADNDBIN\05Fragment 50000 100
```

The application proceeds to allocate memory, in chunks of the specified allocation size, until the limit is reached. Once the allocations have been made, the application performs a couple of garbage collections to promote the surviving objects to generation 2 and then makes the non pinned objects rootless, followed by another garbage collection that subsequently releases the non pinned allocations. Let's take a look by running the application under the debugger with an allocation size of `50000` and a max memory threshold of 1Gb.

Once the `Press any key to GC and promo to Gen1` prompt is displayed the application has finished allocating all the memory and we can take a look at the managed heap using the `DumpHeap –stat` command:

```
0:004> !DumpHeap -stat
total 22812 objects
Statistics:
      MT    Count    TotalSize Class Name
79119954        1           12
System.Security.Permissions.ReflectionPermission
79119834        1           12
System.Security.Permissions.FileDialogPermission
791197b0        1           12 System.Security.PolicyManager
…
…
…
791032a8        2          256 System.Globalization.NumberFormatInfo
79101fe4        6          336 System.Collections.Hashtable
7912d9bc        6          864 System.Collections.Hashtable+bucket[]
7912dd40       10         2084 System.Char[]
00395f68      564        13120      Free
7912d8f8       14        17348 System.Object[]
791379e8        1        80012 System.Runtime.InteropServices.GCHandle[]
79141f50        2        80032 System.Byte[][]
790fd8c4     2108       132148 System.String
7912dae8    20002   1000240284 System.Byte[]
Total 22812 objects
```

The output of the command shows a few interesting fields. Since we are looking specifically for heap fragmentation symptoms, any listed `Free` blocks should be carefully investigated. In our case, we seem to have `564` free blocks occupying a total size of `13120`. Should we be worried about these free blocks causing heap fragmentation? Generally speaking it is useful to look at the total size of the free blocks in comparison to the overall size of the managed heap. If the size of the free blocks is large in comparison to the overall heap size, heap fragmentation may be an issue and should be investigated further. Another important consideration to be made is that of which generation the possible heap fragmentation is occurring in. In generation 0, fragmentation is typically not a problem since the CLR heap manager is able to allocate using any free blocks that may be available. In generation 1 and 2 however, the only way for the free blocks to be used is by promoting objects to each respective generation. Since generation 1 is part of the ephemeral segment which there can only be one of, most commonly, generation 2 is the generation of interest when looking at heap fragmentation problems. Let's take a look at what our heap looks like by using the `eeheap –gc` command:

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x56192a54
generation 1 starts at 0x55d91000
generation 2 starts at 0x01c21000
ephemeral segment allocation context: none
 segment    begin allocated     size
003a80e0 790d8620  790f7d8c 0x0001f76c(128876)
01c20000 01c21000  0282db84 0x00c0cb84(12635012)
04800000 04801000  05405ee4 0x00c04ee4(12603108)
05800000 05801000  06405ee4 0x00c04ee4(12603108)
06a50000 06a51000  07655ee4 0x00c04ee4(12603108)
07a50000 07a51000  08655ee4 0x00c04ee4(12603108)
…
…
…
4fd90000 4fd91000  50995ee4 0x00c04ee4(12603108)
50d90000 50d91000  51995ee4 0x00c04ee4(12603108)
51d90000 51d91000  52995ee4 0x00c04ee4(12603108)
52d90000 52d91000  53995ee4 0x00c04ee4(12603108)
53d90000 53d91000  54995ee4 0x00c04ee4(12603108)
54d90000 54d91000  55995ee4 0x00c04ee4(12603108)
55d90000 55d91000  5621afd8 0x00489fd8(4759512)
Large object heap starts at 0x02c21000
 segment    begin allocated     size
02c20000 02c21000  02c23250 0x00002250(8784)
Total Size  0x3ba38e90(1000574608)
------------------------------
GC Heap Size  0x3ba38e90(1000574608)
```

The last line of the output tells us that the total GC Heap Size is right around 1GB. You will also notice that there is a rather large list of segments. Since we are allocating a rather large amount of memory, the ephemeral segment gets filled up pretty quickly and new generation 2 segments get created. We can verify this by looking at the starting address of generation 2 in the output above (`0x01c21000`) and correlate the start addresses of each segment in the segment list. Let's get back to the free blocks we saw earlier. In which generations are they located? We can find out by using the `dumpheap –type Free` command. An abbreviated out is shown below.

???Mario—In the highlighted text above (and below), Gb (gigabits) or GB (gigabytes)? Just checking. Thanks, Chris

```
0:004> !DumpHeap -type Free
 Address       MT     Size
01c21000 00395f68       12 Free
01c2100c 00395f68       24 Free
01c24c44 00395f68       12 Free
01c24c50 00395f68       12 Free
01c24c5c 00395f68     6336 Free
01e299d0 00395f68       12 Free
0202a6f4 00395f68       12 Free
0222b418 00395f68       12 Free
0242c13c 00395f68       12 Free
0262ce60 00395f68       12 Free
04801000 00395f68       12 Free
0480100c 00395f68       12 Free
04a01d30 00395f68       12 Free
04c02a54 00395f68       12 Free
```

```
04e03778 00395f68       12 Free
0500449c 00395f68       12 Free
052051c0 00395f68       12 Free
05801000 00395f68       12 Free
0580100c 00395f68       12 Free
05a01d30 00395f68       12 Free
05c02a54 00395f68       12 Free
05e03778 00395f68       12 Free
0600449c 00395f68       12 Free
062051c0 00395f68       12 Free
06a51000 00395f68       12 Free
06a5100c 00395f68       12 Free
06c51d30 00395f68       12 Free
06e52a54 00395f68       12 Free
07053778 00395f68       12 Free
0725449c 00395f68       12 Free
074551c0 00395f68       12 Free
07a51000 00395f68       12 Free
07a5100c 00395f68       12 Free
07c51d30 00395f68       12 Free
07e52a54 00395f68       12 Free
08053778 00395f68       12 Free
0825449c 00395f68       12 Free
084551c0 00395f68       12 Free
08a51000 00395f68       12 Free
08a5100c 00395f68       12 Free
08c51d30 00395f68       12 Free
08e52a54 00395f68       12 Free
09053778 00395f68       12 Free
0925449c 00395f68       12 Free
094551c0 00395f68       12 Free
09a51000 00395f68       12 Free
09a5100c 00395f68       12 Free
09c51d30 00395f68       12 Free
09e52a54 00395f68       12 Free
0a053778 00395f68       12 Free
0a25449c 00395f68       12 Free
0a4551c0 00395f68       12 Free
0aee1000 00395f68       12 Free
0aee100c 00395f68       12 Free
0b0e1d30 00395f68       12 Free
0b2e2a54 00395f68       12 Free
0b4e3778 00395f68       12 Free
…
…
…
55192a54 00395f68       12 Free
55393778 00395f68       12 Free
5559449c 00395f68       12 Free
557951c0 00395f68       12 Free
55d91000 00395f68       12 Free
55d9100c 00395f68       12 Free
55f91d30 00395f68       12 Free
56192a54 00395f68       12 Free
02c21000 00395f68       16 Free
02c22010 00395f68       16 Free
02c23020 00395f68       16 Free
02c23240 00395f68       16 Free
total 564 objects
Statistics:
      MT    Count    TotalSize Class Name
```

```
00395f68      564         13120      Free
Total 564 objects
```

By looking at the address of each of the free blocks and correlating the address to the segments from the `eeheap` command we can see that a great majority of the free objects reside in generation 2. Now, with a total free size of 13120 in a heap that is right around 1GB in size, the fragmentation is only a small fraction of one percent. Nothing to worry about (yet). Let's resume the application and keep pressing any key when prompted until you see the `Press any key to exit` prompt. At that point, break into the debugger and once again run the `DumpHeap –stat` command to get another view of the heap:

```
0:004> !DumpHeap -stat
total 22233 objects
Statistics:
      MT    Count    TotalSize Class Name
79119954        1           12
System.Security.Permissions.ReflectionPermission
79119834        1           12
System.Security.Permissions.FileDialogPermission
791197b0        1           12 System.Security.PolicyManager
00113038        1           12 Advanced.NET.Debugging.Chapter5.Fragment
791052a8        1           16 System.Security.Permissions.UIPermission
79117480        1           20
System.Security.Permissions.EnvironmentPermission
791037c0        1           20
Microsoft.Win32.SafeHandles.SafeFileMappingHandle
79103764        1           20
Microsoft.Win32.SafeHandles.SafeViewOfFileHandle
…
…
…
7912d8f8       12        17256 System.Object[]
791379e8        1        80012 System.Runtime.InteropServices.GCHandle[]
79141f50        2        80032 System.Byte[][]
790fd8c4     2101       131812 System.String
00395f68    10006    496172124      Free
7912dae8    10002    500120284 System.Byte[]
Total 22233 objects
```

This time we can see that the amount of free space has grown considerably. From the output, there are `10006` instances of free blocks occupying a total of `496172124` bytes of memory. To find out how that total amount correlates to our overall heap size we once again use the `eeheap –gc` command:

```
0:004> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x55d9100c
generation 1 starts at 0x55d91000
generation 2 starts at 0x01c21000
ephemeral segment allocation context: none
 segment    begin allocated     size
003a80e0 790d8620  790f7d8c 0x0001f76c(128876)
01c20000 01c21000  02821828 0x00c00828(12585000)
04800000 04801000  053f9b88 0x00bf8b88(12553096)
…
…
```

```
…
54d90000 54d91000  55989b88 0x00bf8b88(12553096)
55d90000 55d91000  562190b0 0x004880b0(4751536)
Large object heap starts at 0x02c21000
 segment    begin allocated     size
02c20000 02c21000  02c23240 0x00002240(8768)
Total Size  0x3b6725f4(996615668)
------------------------------
GC Heap Size  0x3b6725f4(996615668)
```

The total GC heap size is reported as `996615668` bytes. Overall, we can say that the heap is approximately 50% fragmented. This can easily be verified by looking at the verbose output of the `DumpHeap` command:

```
0:004> !DumpHeap
 Address       MT     Size
…
…
…
55ff381c 7912dae8    50012
55fffb78 00395f68    50012 Free
5600bed4 7912dae8    50012
56018230 00395f68    50012 Free
5602458c 7912dae8    50012
560308e8 00395f68    50012 Free
5603cc44 7912dae8    50012
56048fa0 00395f68    50012 Free
560552fc 7912dae8    50012
56061658 00395f68    50012 Free
5606d9b4 7912dae8    50012
56079d10 00395f68    50012 Free
5608606c 7912dae8    50012
560923c8 00395f68    50012 Free
5609e724 7912dae8    50012
560aaa80 00395f68    50012 Free
560b6ddc 7912dae8    50012
560c3138 00395f68    50012 Free
560cf494 7912dae8    50012
560db7f0 00395f68    50012 Free
560e7b4c 7912dae8    50012
560f3ea8 00395f68    50012 Free
56100204 7912dae8    50012
5610c560 00395f68    50012 Free
…
…
…
```

From the output, we can see that a pattern has emerged. We have a block of size 50012 that is allocated and in use followed by a free block of the same size that is considered free. We can use the `DumpObj` command on the allocated object to find out more details:

```
0:004> !DumpObj 5606d9b4
Name: System.Byte[]
MethodTable: 7912dae8
EEClass: 7912dba0
Size: 50012(0xc35c) bytes
Array: Rank 1, Number of elements 50000, Type Byte
```

```
Element Type: System.Byte
Fields:
None
```

This object is a byte array which corresponds to the allocations that our application is creating. How did we end up with such an allocation pattern (allocated, free, allocated, free) to begin with? We know that the garbage collector should perform compaction and coalescing to avoid this scenario. One of the situations that can cause the garbage collector not to compact and coalesce is if there are objects on the heap that are pinned (i.e., non-moveable). To find out if that is indeed the case in our application we need to see if there are any pinned handles in the process. We can utilize the `GCHandles` command to get an overview of handle usage in the process:

```
0:004> !GCHandles
GC Handle Statistics:
Strong Handles: 15
Pinned Handles: 10004
Async Pinned Handles: 0
Ref Count Handles: 0
Weak Long Handles: 0
Weak Short Handles: 1
Other Handles: 0
Statistics:
      MT    Count    TotalSize Class Name
790fd0f0        1           12 System.Object
790feba4        1           28 System.SharedStatics
790fcc48        2           48 System.Reflection.Assembly
790fe17c        1           72 System.ExecutionEngineException
790fe0e0        1           72 System.StackOverflowException
790fe044        1           72 System.OutOfMemoryException
790fed00        1          100 System.AppDomain
790fe704        2          112 System.Threading.Thread
79100a18        4          144 System.Security.PermissionSet
790fe284        2          144 System.Threading.ThreadAbortException
7912d8f8        4         8744 System.Object[]
7912dae8    10000    500120000 System.Byte[]
Total 10020 objects
```

The output of GCHandles tells us that we have 10004 pinned handles. Furthermore, in the `statistics` section, we can see that 10000 of those handles are used to pin byte arrays. At this point we are almost there and can do a quick code review that shows that half of the byte array allocations made in the application are explicitly pinned, causing the heap to get fragmented.

Excessive or prolonged pinning is probably one of the most common reasons behind fragmentation of the managed heap. If pinning is necessary, the developer must ensure that pinning is short-lived in order not to interfere too much with the garbage collector.

How Much Is Too Much?

In our example above, initially, the heap fragmentation was a fraction of one percent. At that point, we said that we really don't have to pay too much attention to it as it was small enough not to concern us. Later on, we noticed that the fragmentation grew to 50% which caused an in-depth investigation to figure out the reason for it. Is there a magical percentage of when one should start worrying? There doesn't exist a hard number, but generally speaking if the heap is between 10%-30%fragmented, due diligence should be exercised to ensure that it is not a long running problem.

In the example above, we looked at fragmentation as it relates to the managed heap. It is also possible to encounter situations where the virtual memory managed by the Windows Virtual Memory manager gets fragmented. In those cases, the CLR heap manager may not be able to grow its heap (i.e., allocate new segments) to accommodate allocation requests. The `address` command can be used to get in-depth information on the systems virtual memory state.

Debugging Out of Memory Exceptions

Even though the CLR heap manager and the garbage collector work hard to ensure that memory is automatically managed and used in the most efficient way possible, bad programming can still cause serious issues in .NET applications. In this part of the chapter we will take a look at how a .NET application can exhaust enough memory to fail with an `OutOfMemoryException` and how we can use the debuggers to figure out the source of the problem. It is important to note that the example we will use illustrates how memory can be exhausted in the managed world and does not cover the various ways in which how resources can be leaked in native code when invoked via the interoperability services layer. In Chapter 7, "Interoperability" we will look at an example of a native resource leak caused by improper invocations from managed code.

The application we will use to illustrate the problem is shown in Listing 5-9.

Listing 5-9

Example of an application that causes an eventual OutOfMemoryException.

```
using System;
using System.IO;
using System.Xml.Serialization;

namespace Advanced.NET.Debugging.Chapter5
{

    public class Person
    {
        private string name;
        private string social;
        private int age;

        public string Name
        {
            get { return name; }
            set { this.name=value;}
        }

        public string SocialSecurity
        {
            get { return social; }
            set { this.social= value; }
        }

        public int Age
        {
            get { return age; }
            set { this.age = value; }
        }

        public Person() {}
        public Person(string name, string ss, int age)
        {
            this.name = name; this.social = ss; this.age = age;
        }
    }

    class OOM
    {
        static void Main(string[] args)
        {
            OOM o = new OOM();
            o.Run();
        }

        public void Run()
        {
            XmlRootAttribute root = new XmlRootAttribute();
            root.ElementName = "MyPersonRoot";
            root.Namespace = "http://www.contoso.com";
            root.IsNullable = true;

            while (true)
            {
                Person p = new Person();
                p.Name = "Mario Hewardt";
                p.SocialSecurity = "xxx-xx-xxxx";
                p.Age = 99;

                XmlSerializer ser = new
                    XmlSerializer(typeof(Person), root);
                Stream s = new
                    FileStream("c:\\ser.txt", FileMode.Create);

                ser.Serialize(s, p);
                s.Close();
            }
        }
    }
}
```

The source code and binary for Listing 5-9 can be found in the following folders:

[lb] Source code: `C:\ADND\Chapter5\OOM`

[lb] Binary: `C:\ADNDBin\05OOM.exe`

The application is pretty straightforward and consists of a `Person` class and an `OOM` class. The `OOM` class contains a `Run` method that sits in a tight loop creating instances of the `Person` class and serializes the instance into XML stored in a file on the local drive. When we run this application we would like to monitor the memory consumption to see if it steadily increases over time which could eventually lead to an `OutOfMemoryException` being thrown. What tools do we have at our disposal to monitor the memory consumption of a process? Well, we have several options. The most basic option is to simply use task manager (shortcut `SHIFT-CTRL-ESC`). Task manager can display per process memory information such as the working set, commit size and paged/non-paged pool. By default, only the Memory (Private Working Set) is enabled. In order to enable other process information, the Select columns menu choice on the View menu can be used. The Windows Task Manager has several different tabs and the tab of most interest when looking at per process details is the Processes tab. The Processes tab shows a number of rows where each row represents a running process. Each of the columns in turn shows a specific piece of information about the process. Figure 5-13 shows an example of Windows Task Manager with a number of different memory details enabled in the Processes tab.

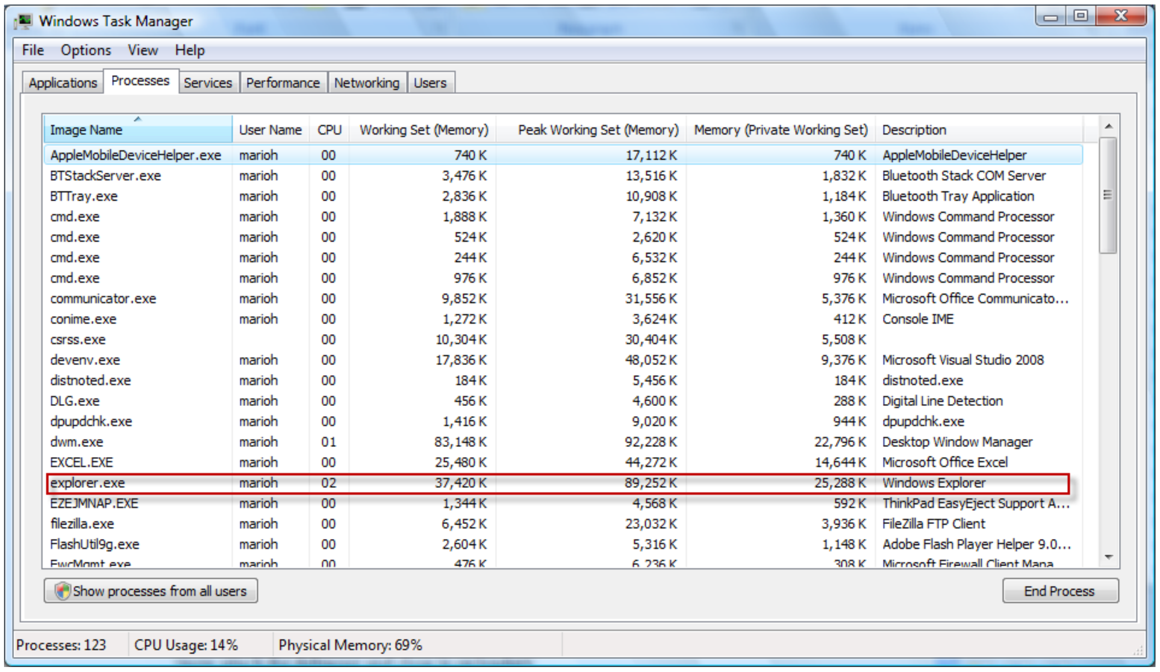

Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Show processes from all users
End Process

Image Name	User Name	CPU	Working Set (Memory)	Peak Working Set (Memory)	Memory (Private Working Set)	Description
AppleMobileDeviceHelper.exe	marioh	00	740 K	17,112 K	740 K	AppleMobileDeviceHelper
BTStackServer.exe	marioh	00	3,476 K	13,516 K	1,832 K	Bluetooth Stack COM Server
BTTray.exe	marioh	00	2,836 K	10,908 K	1,184 K	Bluetooth Tray Application
cmd.exe	marioh	00	1,888 K	7,132 K	1,360 K	Windows Command Processor
cmd.exe	marioh	00	524 K	2,620 K	524 K	Windows Command Processor
cmd.exe	marioh	00	244 K	6,532 K	244 K	Windows Command Processor
cmd.exe	marioh	00	976 K	6,852 K	976 K	Windows Command Processor
communicator.exe	marioh	00	9,852 K	31,556 K	5,376 K	Microsoft Office Communicato...
conime.exe	marioh	00	1,272 K	3,624 K	412 K	Console IME
csrss.exe		00	10,304 K	30,404 K	5,508 K	
devenv.exe	marioh	00	17,836 K	48,052 K	9,376 K	Microsoft Visual Studio 2008
distnoted.exe	marioh	00	184 K	5,456 K	184 K	distnoted.exe
DLG.exe	marioh	00	456 K	4,600 K	288 K	Digital Line Detection
dpupdchk.exe	marioh	00	1,416 K	9,020 K	944 K	dpupdchk.exe
dwm.exe	marioh	01	83,148 K	92,228 K	22,796 K	Desktop Window Manager
EXCEL.EXE	marioh	00	25,480 K	44,272 K	14,644 K	Microsoft Office Excel
explorer.exe	marioh	02	37,420 K	89,252 K	25,288 K	Windows Explorer
EZEJMNAP.EXE	marioh	00	1,344 K	4,568 K	592 K	ThinkPad EasyEject Support A...
filezilla.exe	marioh	00	6,452 K	23,032 K	3,936 K	FileZilla FTP Client
FlashUtil9g.exe	marioh	00	2,604 K	5,316 K	1,148 K	Adobe Flash Player Helper 9.0...
FwcMgmt.exe	marioh	00	476 K	6,236 K	308 K	Microsoft Firewall Client Mana

Processes: 123 | CPU Usage: 14% | Physical Memory: 69%

Figure 5-13
Example of Windows Task Manager process view

In Figure 5-13 we can see, for example, that `explorer.exe` working set size is 37,420K. Before we can move forward and effectively utilize Windows Task Manager for memory related investigations we have to have a clear understanding of what each of the possible memory related columns mean. Table 5-2 details the most commonly used columns and their description.

Table 5-2
Windows Task Manager Memory related columns

Column	Description
Memory – Working Set	Amount of memory in the private working set as well as the shared memory
Memory – Peak Working Set	Maximum amount of working set used by the process
Memory – Working Set Delta	Amount of change in the working set
Memory – Private Working Set	Amount of memory the process is using minus shared memory
Memory – Commit Size	Amount of virtual memory committed by the process

Pre Windows Vista Task Manager

Some much needed changes were made in Windows Vista and higher versions to better capture the memory related process information. Prior to Windows Vista, Windows Task Manager had a column named VM size which, contrary to popular belief, indicated the amount of private bytes a process was consuming. Similarly, the Mem Usage column corresponds to the working set (including shared memory) of the process. Finally, a feature we will utilize in Chapter 8, “Postmortem Debugging,” is the capability to create dump files simply by right clicking on the process and choosing the Create Dump File item.

Let’s run `05OOM.exe` and watch the Memory – Working Set, Memory – Private Working Set, and Memory – Commit Size columns. Table 5-3 shows the results taken at periodic (approx 60 sec) intervals.

Table 5-3
Memory Usage of 005OOM.exe

Interval	Working Set (K)	Private Working Set (K)	Commit Size (K)
1	18,000	7,000	16,000
2	24,000	11,000	19,000
3	28,000	13,000	22,000
4	33,000	16,000	25,000
5	38,000	19,000	28,000
6	42,000	22,000	30,000

From Table 5-3 we can see that we have a steady increase across the board. Both the working set sizes as well as the commit size are continuously growing. If this application is allowed to run indefinitely chances are high that it could eventually run out of memory and an `OutOfMemoryException` would be thrown. While using the Windows Task Manager is useful to get an overview of the memory consumed, what information does it present to us as far as figuring out the source of the excessive memory consumption? Is the memory located on the native heap or the managed heap? Is it located on the heap period or elsewhere?

To find the answers to those questions, we need a more granular tool to aid us and that tool is called the Windows Reliability and Performance Monitor. The Windows Reliability and Performance Monitor tool is a powerful and extensible tool that can be used to investigate the state of the system as a whole or on a per process basis. The tool uses several different data sources such as performance counters, trace logs and configuration information. During .NET debug sessions, performance counters is the most commonly used data source. A performance counter is an entity that is responsible for publishing a specific performance characteristic of an application or service at regular time intervals or under specific conditions. For example, a web service servicing credit card transactions can publish a performance counter that shows how many failed transactions that have occurred over time. The Windows Reliability and Performance tool knows where to gather the performance counter data and displays the results in a nice graphical and historical view. To run the tool, click the Windows Start button and type `perfmon.exe` in the search tool (prior to Windows Vista select run and then type `perfmon.exe`). Figure 5-14 shows an example of the start screen of the tool.

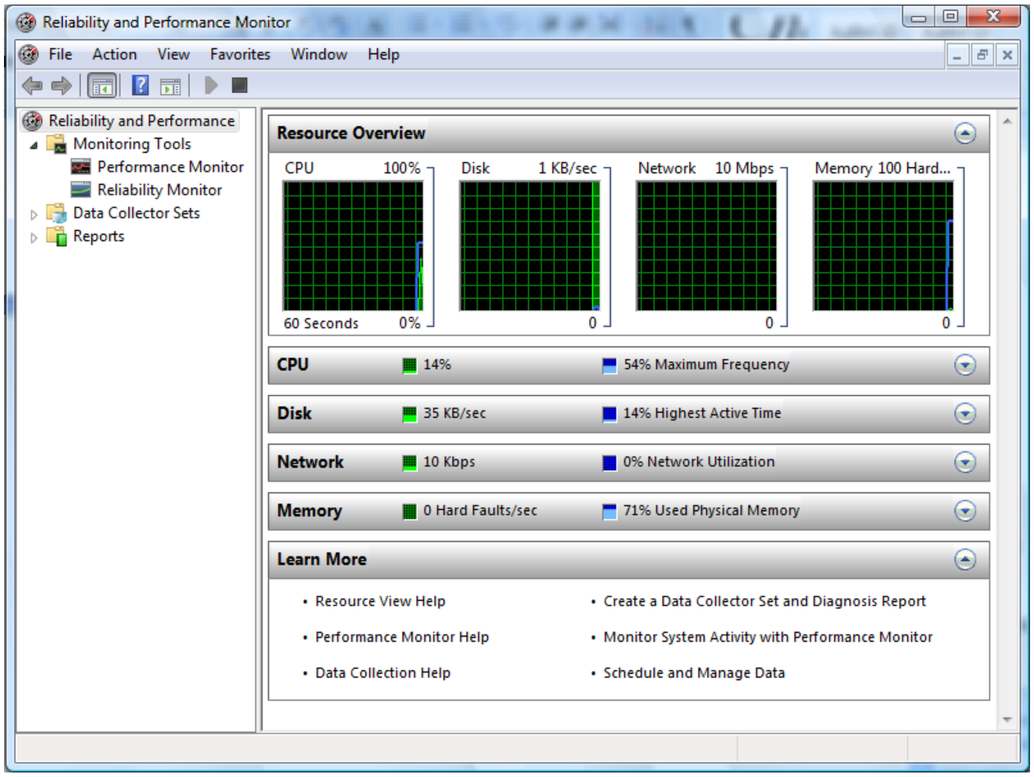
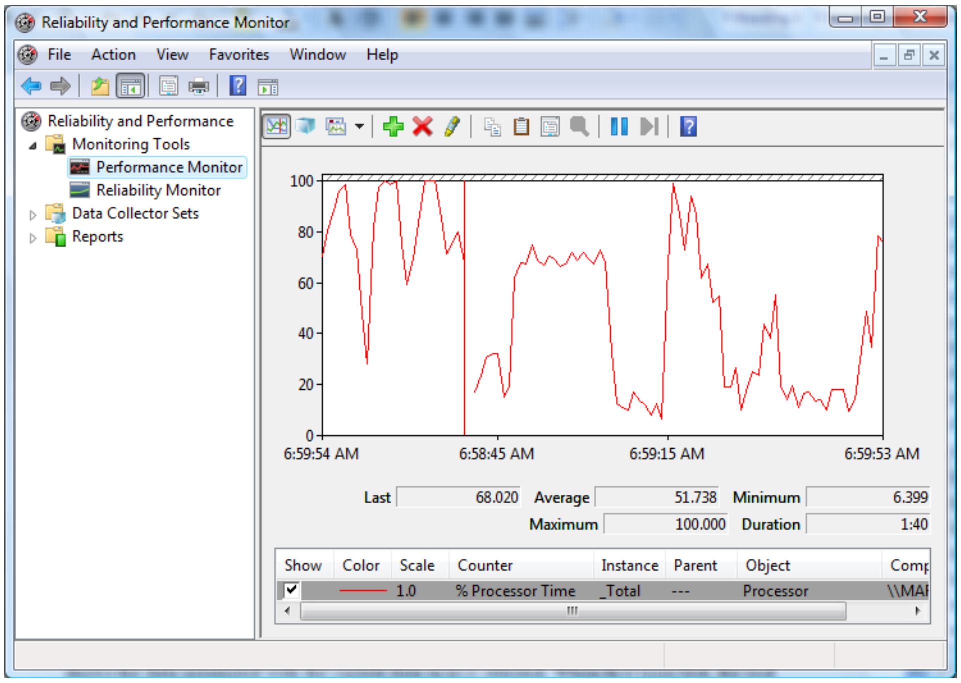

Reliability and Performance Monitor
File Action View Favorites Window Help
Reliability and Performance
Monitoring Tools
Performance Monitor
Reliability Monitor
Data Collector Sets
Reports
Resource Overview
CPU 100%
Disk 1 KB/sec
Network 10 Mbps
Memory 100 Hard...
60 Seconds 0%
0
0
0
CPU 14% 54% Maximum Frequency
Disk 35 KB/sec 14% Highest Active Time
Network 10 Kbps 0% Network Utilization
Memory 0 Hard Faults/sec 71% Used Physical Memory
Learn More
Resource View Help
Performance Monitor Help
Data Collection Help
Create a Data Collector Set and Diagnosis Report
Monitor System Activity with Performance Monitor
Schedule and Manage Data

Figure 5-14
Windows Reliability and Performance Monitor

The left hand pane shows the different data sources available to the tool. As mentioned earlier, performance counters are used quite heavily when diagnosing .NET applications and are located under the Monitoring Tools node under Performance Monitor. The right hand pane shows the data associated with the current data source selected. When first launched, the tool shows an overview of the system state including CPU, Disk, Memory and Network utilization. Figure 5-15 shows the tool once the Performance Monitor item is selected.



Last	68.020	Average	51.738	Minimum	6.399
		Maximum	100.000	Duration	1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Comp
✓	🟥	1.0	% Processor Time	_Total	---	Processor	\\MAI

Figure 5-15
Performance Monitor

The right hand pane now displays a visual representation of the selected performance counters over time. By default, the Processor Time counter is always selected when the tool is first launched. In order to add additional counters, right click in the right pane and select Add Counters, which brings up the Add Counters dialog shown in Figure 5-16.

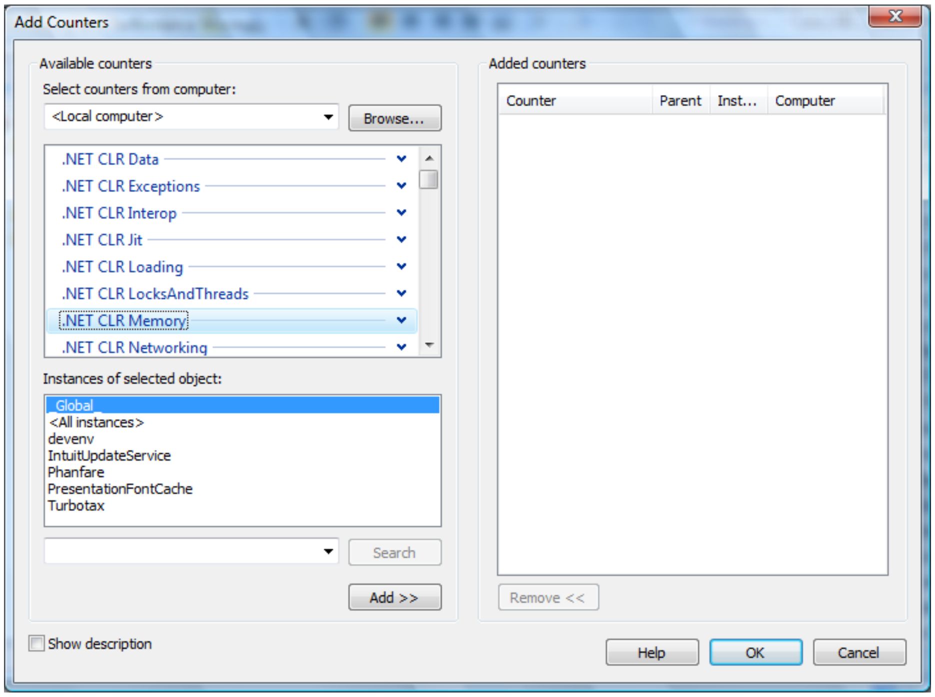


Figure 5-16
Add Counters dialog

The Add Counters dialog has two parts. The first part is the left hand side Add Counters options which shows a drop down list of all available counter categories as well as the instances of the available objects that the performance counters can collect and display data on. For example, if the .NET CLR Memory performance counter category is selected the list of available instances shows the processes that are available. The right hand pane simply shows all the performance counters that have been added.

Now that we know how to add and display performance counters in the tool let's try it out on our sample application. The first question we have to answer before blindly adding random performance counters is which CLR counters are we specifically interested in based on the symptoms we are seeing? Table 5-4 shows the available CLR performance counter categories as well as their associated descriptions.

Table 5-4
CLR specific performance counters categories

Category	**Description**
.NET CLR Data	Runtime statistics on data (such as SQL) performance
.NET CLR Exceptions	Runtime statistics on CLR exception handling such as number of exceptions thrown
.NET CLR Interop	Runtime statistics on the interoperability services such as number of marshalling operations
.NET CLR Jit	Runtime statistics on the just in time compiler such as number of methods jitted.
.NET CLR Loading	Runtime statistics on the CLR class/assembly loader such as total number of bytes in the loader heap
.NET CLR LocksAndThreads	Runtime statistics on locks and threads such as the contention rate of a lock
.NET CLR Memory	Runtime statistics on the managed heap and garbage collector such as the number of collections in each generation.

.NET CLR Networking	Runtime statistics on networking such as datagrams sent and received.
.NET CLR Remoting	Runtime statistics on remoting such as remote calls per second
.NET CLR Security	Runtime statistics on security the total number of runtime checks.

???Mario—Should there be a description of .NET CLR Data? Thanks, Chris

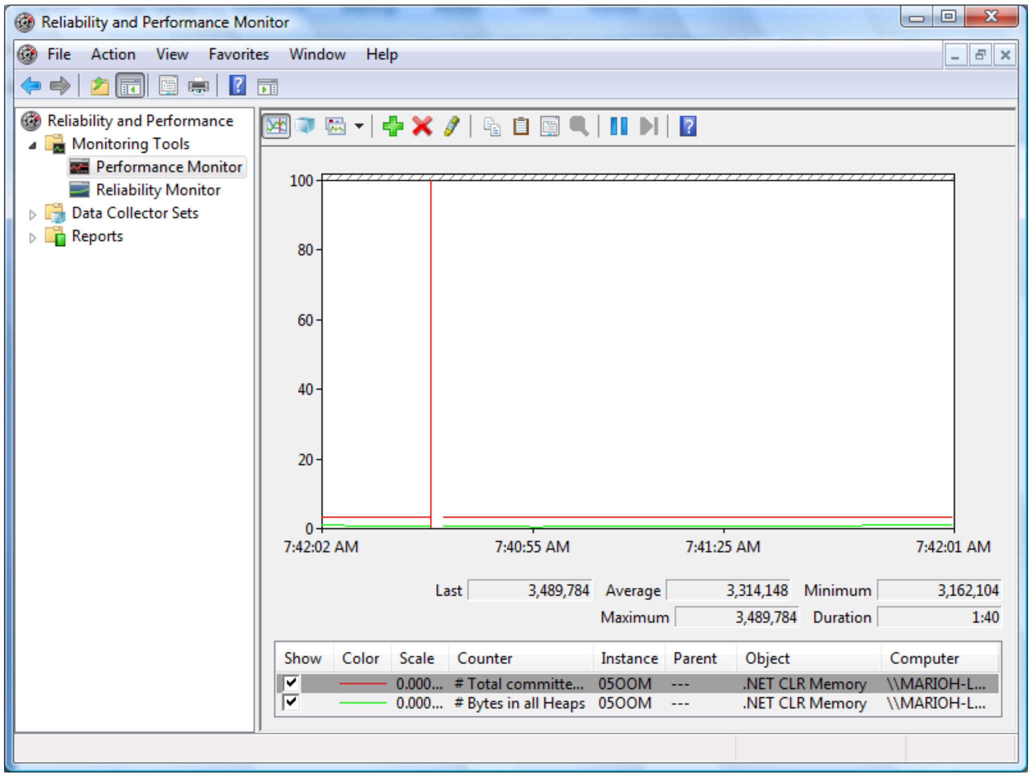
Based on the plethora of available categories, in our specific example, we are interested in finding our more details on the memory consumption (.NET CLR Memory) of our sample application. Table 5-5 shows the specific counters available in this category as well as their descriptions.

Table 5-5

.NET CLR Memory performance counters

Performance Counter	**Description**
# Bytes in all heaps	The total number of bytes in all heaps (gen 0, gen 1, gen 2 and large object heap)
# GC Handles	Total number of GC handles
# Gen 0 collections	Total number of generation 0 garbage collections
# Gen 1 collections	Total number of generation 1 garbage collections
# Gen 2 collections	Total number of generation 2 garbage collections
# Induced GC	Total number of times a call to GC.Collect has been made
# Pinned objects	Total number of pinned objects in the managed heap during the last garbage collection. Please note that it only displays the number of pinned objects from the generations that were collected. As such, if a garbage collection resulted in only generation 0 being collected, this number only states how many pinned objects were in that generation.
# of Sink blocks in use	Current number of sync blocks in use. Useful when diagnosing performance problems related to heavy synchronization usage.
# Total committed bytes	Total number of virtual bytes committed by the CLR heap manager
# Total reserved bytes	Total number of virtual bytes reserved by the CLR heap manager
% Time in GC	Percentage of total elapsed time spent in the garbage collector since the last garbage collection
Allocated bytes/sec	Number of allocated bytes per second. Updated at the beginning of every garbage collection
Finalization Survivors	The number of garbage collected objects that survive a collection due to waiting for finalization
Gen 0 heap size	Maximum number of bytes that can be allocated in generation 0.
Gen 0 Promoted bytes/sec	Number of promoted bytes per second in generation 0
Gen 1 heap size	Current number of bytes in generation 1.
Gen 1 Promoted bytes/sec	Number of promoted bytes per second in generation 1
Gen 2 heap size	Current number of bytes in generation 1
Large object heap size	Current size of the large object heap
Process ID	Process identifier of process being monitored
Promoted finalization – Memory from gen 0	The number of bytes that are promoted to generation 1 due to waiting to be finalized
Promoted memory from Gen 0	The number of bytes promoted from generation 0 to generation 1 (minus objects that are waiting to be finalized)
Promoted memory from Gen 1	The number of bytes promoted from generation 1 to generation 2 (minus objects that are waiting to be finalized)

To monitor our sample application’s memory usage let’s pick the # total bytes counter as well as the # total committed bytes counter. This can give us valuable clues as to whether the memory is on the managed heap or elsewhere in the process. Start the `05OOM.exe` application followed by launching the Windows Reliability and Performance Monitoring tool. Add the two counters and specify the `05OOM.exe` instance in the list of available instances. Figure 5-17 shows the output of the tool after about 2 minutes of 05OOM.exe runtime.

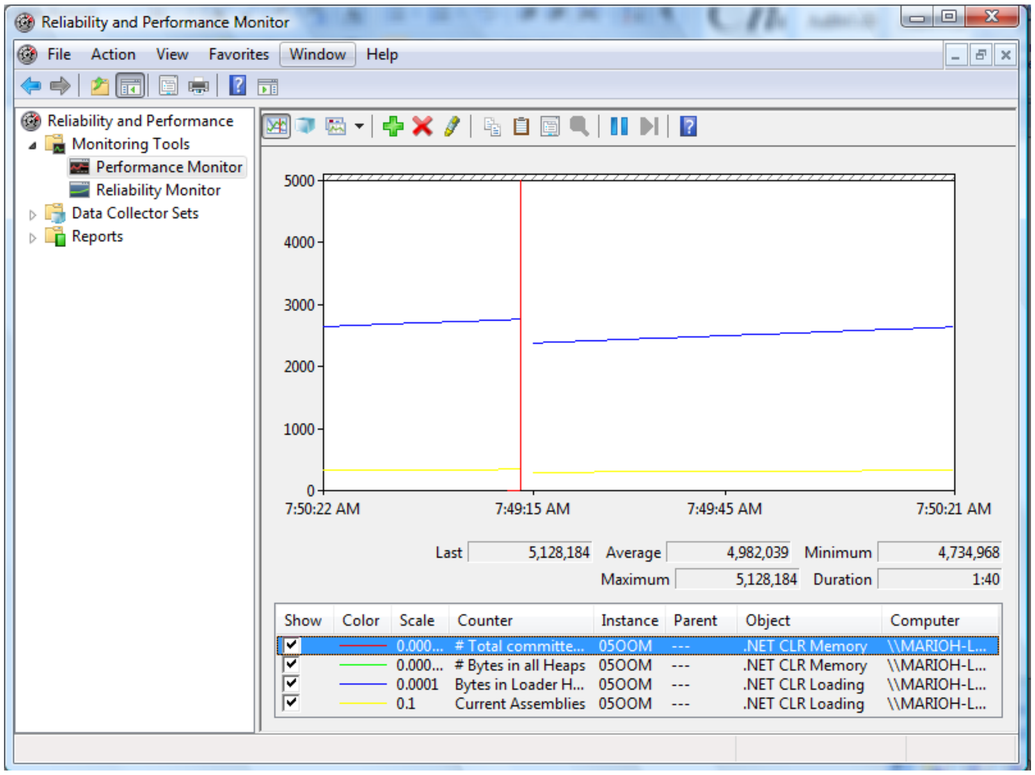


```
Last  3,489,784   Average  3,314,148   Minimum  3,162,104
                  Maximum  3,489,784   Duration      1:40
```

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	0.000...	# Total committe...	05OOM	---	.NET CLR Memory	\\MARIOH-L...
✓	🟩	0.000...	# Bytes in all Heaps	05OOM	---	.NET CLR Memory	\\MARIOH-L...

Figure 5-17
Monitoring 05OOM.exe total and committed bytes

The counters look pretty stable with no major uptick. Yet if we look at the 05OOM process in Windows Task Manager we can see that memory consumption is increasing quite a bit. Where is the memory coming from? At this point, we have eliminated the managed heap as being the cause for memory usage growing and our strategy is now to use the other various counters available to see if we can spot an uptick. For example, let’s choose the bytes in loader heap and current assemblies (both under the .NET CLR Loading category) and see what the output shows (See Figure 5-18).



Last	5,128,184	Average	4,982,039	Minimum	4,734,968
		Maximum	5,128,184	Duration	1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	0.000...	# Total committe...	05OOM	---	.NET CLR Memory	\\MARIOH-L...
✓	🟩	0.000...	# Bytes in all Heaps	05OOM	---	.NET CLR Memory	\\MARIOH-L...
✓	🟦	0.0001	Bytes in Loader H...	05OOM	---	.NET CLR Loading	\\MARIOH-L...
✓	🟨	0.1	Current Assemblies	05OOM	---	.NET CLR Loading	\\MARIOH-L...

Figure 5-18
Monitoring 05OOM.exe current assemblies and bytes in loader heap performance counters

Note that you may have to change the vertical scale maximum (under properties) to a larger number depending on how long the application has been executing. In Figure 5-18 the vertical scale maximum has been set to 5000. This time we can see some more interesting data. Both the bytes in loader heap and current assemblies performance counters are slowly increasing over time. One of our theories can then be that we may be looking at a potential assembly leak. To verify this we can attach the debugger to the `05OOM.exe` process (`ntsd –pn 05oom.exe`) and use the `eeheap -loader` command:

```
0:003> !eeheap -loader
Loader Heap:
--------------------------------------
System Domain: 7a3bc8b8
LowFrequencyHeap: Size: 0x0(0)bytes.
HighFrequencyHeap: 002a2000(8000:1000) Size: 0x1000(4096)bytes.
StubHeap: 002aa000(2000:2000) Size: 0x2000(8192)bytes.
Virtual Call Stub Heap:
  IndcellHeap: Size: 0x0(0)bytes.
  LookupHeap: Size: 0x0(0)bytes.
  ResolveHeap: Size: 0x0(0)bytes.
  DispatchHeap: Size: 0x0(0)bytes.
  CacheEntryHeap: Size: 0x0(0)bytes.
Total size: 0x3000(12288)bytes
--------------------------------------
Shared Domain: 7a3bc560
LowFrequencyHeap: 002d0000(2000:1000) Size: 0x1000(4096)bytes.
HighFrequencyHeap: 002d2000(8000:1000) Size: 0x1000(4096)bytes.
StubHeap: 002da000(2000:1000) Size: 0x1000(4096)bytes.
Virtual Call Stub Heap:
  IndcellHeap: 00870000(2000:1000) Size: 0x1000(4096)bytes.
  LookupHeap: 00875000(2000:1000) Size: 0x1000(4096)bytes.
  ResolveHeap: 0087b000(5000:1000) Size: 0x1000(4096)bytes.
  DispatchHeap: 00877000(4000:1000) Size: 0x1000(4096)bytes.
  CacheEntryHeap: 00872000(3000:1000) Size: 0x1000(4096)bytes.
Total size: 0x7000(28672)bytes
--------------------------------------
Domain 1: 304558
LowFrequencyHeap: 002b0000(2000:2000) 00ca0000(10000:10000)
01cf0000(10000:10000) 04070000(10000:10000) 04170000(10000:10000)
…
…
…
 165e0000(10000:10000) 166b0000(10000:10000) 16770000(10000:10000)
16830000(10000:10000) 16900000(10000:10000) 169c0000(10000:10000)
16a80000(10000:a000) Size: 0x16fc000(24100864)bytes.
HighFrequencyHeap: 002b2000(8000:8000) 03e50000(10000:10000)
04370000(10000:10000) 046c0000(10000:10000) 04a10000(10000:10000)
…
…
…
15bf0000(10000:10000) 15f30000(10000:10000) 16270000(10000:10000)
165a0000(10000:10000) 168f0000(10000:a000) Size: 0x572000(5709824)bytes.
StubHeap: 002ba000(2000:1000) Size: 0x1000(4096)bytes.
Virtual Call Stub Heap:
  IndcellHeap: Size: 0x0(0)bytes.
  LookupHeap: Size: 0x0(0)bytes.
  ResolveHeap: 002ca000(6000:1000) Size: 0x1000(4096)bytes.
  DispatchHeap: 002c7000(3000:1000) Size: 0x1000(4096)bytes.
  CacheEntryHeap: 002c2000(4000:1000) Size: 0x1000(4096)bytes.
Total size: 0x1c71000(29822976)bytes
--------------------------------------
Jit code heap:
LoaderCodeHeap: 165f0000(10000:b000) Size: 0xb000(45056)bytes.
LoaderCodeHeap: 15de0000(10000:10000) Size: 0x10000(65536)bytes.
LoaderCodeHeap: 15600000(10000:10000) Size: 0x10000(65536)bytes.
…
…
…

LoaderCodeHeap: 04710000(10000:10000) Size: 0x10000(65536)bytes.
LoaderCodeHeap: 009e0000(10000:10000) Size: 0x10000(65536)bytes.
Total size: 0x23b000(2338816)bytes
--------------------------------------
Module Thunk heaps:
Module 790c2000: Size: 0x0(0)bytes.
Module 002d2564: Size: 0x0(0)bytes.
…
…
…
Module 168f8e40: Size: 0x0(0)bytes.
Module 168f93b8: Size: 0x0(0)bytes.
Module 168f9930: Size: 0x0(0)bytes.
Total size: 0x0(0)bytes
--------------------------------------
Module Lookup Table heaps:
Module 790c2000: Size: 0x0(0)bytes.
Module 002d2564: Size: 0x0(0)bytes.
Module 002d21d8: Size: 0x0(0)bytes.
…
…
```

```
…
Module 168f93b8: Size: 0x0(0)bytes.
Module 168f9930: Size: 0x0(0)bytes.
Total size: 0x0(0)bytes
--------------------------------------
Total LoaderHeap size: 0x1eb6000(32202752)bytes
=======================================
```

The first two domains (system and shared) seem to look reasonable, but the default application domain has a ton of data in it. More specifically it contains the bulk of the overall loader heap (size `32202752`). Why does the application domain contain so much data? We can get further information about the default application domain by using the `DumpDomain` command and specifying the address of the default application domain (found in output from previous `eeheap` command):

```
0:003> !DumpDomain 304558
--------------------------------------
Domain 1: 00304558
LowFrequencyHeap: 0030457c
HighFrequencyHeap: 003045d4
StubHeap: 0030462c
Stage: OPEN
SecurityDescriptor: 00305ab8
Name: 05OOM.exe
Assembly: 0030d1b0
[C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.
dll]
ClassLoader: 002fc988
SecurityDescriptor: 0030dfd8
  Module Name
790c2000
C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.d
ll
002d2564
C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sortkey.nl
p
002d21d8
C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\sorttbls.n
lp

Assembly: 0032f1b8 [C:\ADNDBin\05OOM.exe]
ClassLoader: 002fd168
SecurityDescriptor: 00330f30
  Module Name
002b2c3c C:\ADNDBin\05OOM.exe

Assembly: 0033bb98
[C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c561934e089\Syste
m.Xml.dll]
ClassLoader: 002fd408
SecurityDescriptor: 00326b18
  Module Name
639f8000
C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c561934e089\System
.Xml.dll
…
…
…
```

```
Assembly: 00346408 [4ql4a3hq, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null]
ClassLoader: 003423a8
SecurityDescriptor: 00346380
  Module Name
002b46f8 4ql4a3hq, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

Assembly: 003465a0 [lx4qjutk, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null]
ClassLoader: 00342488
SecurityDescriptor: 00346518
  Module Name
002b4ce4 lx4qjutk, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

Assembly: 003466b0 [uds1hfbo, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null]
ClassLoader: 003424f8
SecurityDescriptor: 00346628
  Module Name
002b5258 uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

…
…
…
```

As we can see, there are a ton of assemblies loaded into the default application domain. Furthermore, the names of the assemblies seem rather random. Why are all these assemblies being loaded? Our code in Listing 5-9 certainly doesn't directly load any assemblies, which means that these assemblies have to be dynamically generated. To further investigate what these assemblies contain, we can pick one of them and dump out the associated module information using the `DumpModule` command:

```
0:003> !DumpModule 002b5258
Name: uds1hfbo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Attributes: PEFile
Assembly: 003466b0
LoaderHeap: 00000000
TypeDefToMethodTableMap: 00ca2df8
TypeRefToMethodTableMap: 00ca2e10
MethodDefToDescMap: 00ca2e6c
FieldDefToDescMap: 00ca2ed0
MemberRefToDescMap: 00ca2ef8
FileReferencesMap: 00ca2fec
AssemblyReferencesMap: 00ca2ff0
MetaData start address: 00cc07c8 (4344 bytes)
```

Next, we dump the metadata of the module using the `dc` command specifying the starting address and the ending address (starting address+size of metadata):

```
0:003> dc 00cc07c8 00cc07c8+0n4344
00cc07c8  424a5342 00010001 00000000 0000000c  BSJB............
00cc07d8  302e3276 3730352e 00003732 00050000  v2.0.50727......
00cc07e8  0000006c 00000528 00007e23 00000594  l...(...#~......
00cc07f8  0000077c 72745323 73676e69 00000000  |...#Strings....
…
…
…
```

```
00cc0d58  00000000 6f4d3c00 656c7564 6475003e  .....<Module>.ud
00cc0d68  66683173 642e6f62 58006c6c 65536c6d  s1hfbo.dll.XmlSe
00cc0d78  6c616972 74617a69 576e6f69 65746972  rializationWrite
00cc0d88  72655072 006e6f73 7263694d 666f736f  rPerson.Microsof
00cc0d98  6d582e74 65532e6c 6c616972 74617a69  t.Xml.Serializat
00cc0da8  2e6e6f69 656e6547 65746172 73734164  ion.GeneratedAss
00cc0db8  6c626d65 6d580079 7265536c 696c6169  embly.XmlSeriali
00cc0dc8  6974617a 65526e6f 72656461 73726550  zationReaderPers
00cc0dd8  58006e6f 65536c6d 6c616972 72657a69  on.XmlSerializer
00cc0de8  65500031 6e6f7372 69726553 7a696c61  1.PersonSerializ
00cc0df8  58007265 65536c6d 6c616972 72657a69  er.XmlSerializer
00cc0e08  746e6f43 74636172 73795300 2e6d6574  Contract.System.
00cc0e18  006c6d58 74737953 582e6d65 532e6c6d  Xml.System.Xml.S
00cc0e28  61697265 617a696c 6e6f6974 6c6d5800  erialization.Xml
00cc0e38  69726553 7a696c61 6f697461 6972576e  SerializationWri
00cc0e48  00726574 536c6d58 61697265 617a696c  ter.XmlSerializa
00cc0e58  6e6f6974 64616552 58007265 65536c6d  tionReader.XmlSe
00cc0e68  6c616972 72657a69 6c6d5800 69726553  rializer.XmlSeri
…
…
…
```

Now we are starting to get somewhere. From the output of the metadata we can see that the module associated with the assembly contains references to some form of XML serialization. Furthermore, it seems that the module contains XML serialization types that are specific to the serialization of the `Person` class in our code. Based on this evidence, we can now hypothesize that the XML serialization code in our application is causing all of these dynamic assemblies to be generated. The next step is the documentation for the `XmlSerializer` class. MSDN clearly states that using the `XmlSerializer` class, for performance reasons, may in fact create a specialized dynamic assembly to handle the serialization. More specifically, seven of the `XmlSerializer` constructors result in dynamic assemblies being generated, whereas the remaining two have reuse logic that reduces the number of dynamic assemblies.

The above scenario illustrated how we can use the Windows Task Manager to monitor the overall memory usage of a .NET application and the Windows Reliability and Performance Monitor tool to drill down into the CLR specifics. The scenario assumed that we had the luxury of running and monitoring the application live. In many cases, the application simply runs until it runs out of memory and throws an `OutOfMemoryException`. If we let our sample application run indefinitely, the `OutOfMemoryException` would have been reported as follows:

```
(1830.1f20): CLR exception - code e0434f4d (first/second chance not
available)
eax=0027ed2c ebx=e0434f4d ecx=00000001 edx=00000000 esi=0027edb4
edi=00338510
eip=775842eb esp=0027ed2c ebp=0027ed7c iopl=0         nv up ei pl nz na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols
for kernel32.dll -
kernel32!RaiseException+0x58:
```

As discussed earlier, to get further information on the managed exception we can use the `PrintException` command:

```
0:000> **kb**
ChildEBP RetAddr  Args to Child
0027ed7c 79f071ac e0434f4d 00000001 00000001
kernel32!RaiseException+0x58
0027eddc 79f0a780 ***51e10dac*** 00000001 00000000
mscorwks!RaiseTheExceptionInternalOnly+0x2a8
*** WARNING: Unable to verify checksum for System.ni.dll
0027ee80 7a53e025 0027f14c 79f0a3d9 0027f338 mscorwks!JIT_Rethrow+0xbf
0027ef4c 7a53d665 51df597c 00000000 51de0050 System_ni+0xfe025
0027ef80 7a4d078a 51df597c 51de0050 638fcb39 System_ni+0xfd665
*** WARNING: Unable to verify checksum for System.Xml.ni.dll
0027efec 638fb6e5 00000000 51de02cc 00000000 System_ni+0x9078a
0027f078 638fa683 51ddff88 00000000 51de02cc System_Xml_ni+0x15b6e5
0027f09c 63960d09 00000000 00000000 00000000 System_Xml_ni+0x15a683
0027f0c4 6396090c 00000000 00000000 00000000 System_Xml_ni+0x1c0d09
0027f120 79e7c74b 00000000 0027f158 0027f1b0 System_Xml_ni+0x1c090c
00000000 00000000 00000000 00000000 00000000
mscorwks!CallDescrWorker+0x33
0:000> **!PrintException *51e10dac***
Exception object: 51e10dac
***Exception type: System.OutOfMemoryException***
Message: <none>
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0027EE94 7942385A
mscorlib_ni!System.Reflection.Assembly.Load(Byte[], Byte[],
System.Security.Policy.Evidence)+0x3a
    0027EEB0 7A4BF513
System_ni!Microsoft.CSharp.CSharpCodeGenerator.FromFileBatch(System.Code
Dom.Compiler.CompilerParameters, System.String[])+0x3ab
    0027EF00 7A53E025
System_ni!Microsoft.CSharp.CSharpCodeGenerator.FromSourceBatch(System.Co
deDom.Compiler.CompilerParameters, System.String[])+0x1f1
    0027EF58 7A53D665
System_ni!Microsoft.CSharp.CSharpCodeGenerator.System.CodeDom.Compiler.I
CodeCompiler.CompileAssemblyFromSourceBatch(System.CodeDom.Compiler.Comp
ilerParameters, System.String[])+0x29
    0027EF8C 7A4D078A
System_ni!System.CodeDom.Compiler.CodeDomProvider.CompileAssemblyFromSou
rce(System.CodeDom.Compiler.CompilerParameters, System.String[])+0x16
    0027EF98 638FCB39
System_Xml_ni!System.Xml.Serialization.Compiler.Compile(System.Reflectio
n.Assembly, System.String,
System.Xml.Serialization.XmlSerializerCompilerParameters,
System.Security.Policy.Evidence)+0x269
    0027F000 638FB6E5
System_Xml_ni!System.Xml.Serialization.TempAssembly.GenerateAssembly(Sys
tem.Xml.Serialization.XmlMapping[], System.Type[], System.String,
System.Security.Policy.Evidence,
System.Xml.Serialization.XmlSerializerCompilerParameters,
System.Reflection.Assembly, System.Collections.Hashtable)+0x7e9
    0027F094 638FA683
System_Xml_ni!System.Xml.Serialization.TempAssembly..ctor(System.Xml.Ser
ialization.XmlMapping[], System.Type[], System.String, System.String,
System.Security.Policy.Evidence)+0x4b
    0027F0B4 63960D09
System_Xml_ni!System.Xml.Serialization.XmlSerializer..ctor(System.Type,
System.Xml.Serialization.XmlAttributeOverrides, System.Type[],
System.Xml.Serialization.XmlRootAttribute, System.String, System.String,
System.Security.Policy.Evidence)+0xed
    0027F0E4 6396090C
System_Xml_ni!System.Xml.Serialization.XmlSerializer..ctor(System.Type,
System.Xml.Serialization.XmlRootAttribute)+0x28
    0027F0F4 009201D6
05OOM!Advanced.NET.Debugging.Chapter5.OOM.Run()+0xe6
    0027F118 009200A7
05OOM!Advanced.NET.Debugging.Chapter5.OOM.Main(System.String[])+0x37

StackTraceString: <none>
HResult: 8007000e
There are nested exceptions on this thread. Run with -nested for details
```

At this point, the application has already failed and we can't rely on runtime monitoring tools to gauge the application's memory usage. In situations like this, we have to rely solely on the debugger commands to analyze where the memory is being consumed. Unfortunately, there is no one cookbook recipe on the exact commands and steps to take but as a general rule of thumb utilizing the various diagnostics commands (such as `eeheap`, `dumpheap`, `dumpdomain` etc) can give invaluable clues as to where in the CLR the memory is being consumed. The excessive memory consumption can, of course, also be as a result of a native code leak which we will see an example of in Chapter 7, "Interoperability".

Immediately Break on OutOfMemoryException

When a process gets into a situation where it is running out of memory things can get very tricky and the application may not be able to properly handle the condition. Since an `OutOfMemoryException` gets propagated up the chain and will not fault the process until the exception is deemed unhandled, a lot of code may still get executed as part of the unwinding making troubleshooting more difficult in certain situations. Furthermore, if the code is hosted in a process that it does not own, the process may catch all kinds of exceptions and continue running. To ensure that an `OutOfMemoryException` always breaks under the debugger, the CLR introduced a registry value called `GCBreakOnOOM` (DWORD) under the following registry path: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework`. The value can be set to 1, in which case an event log message is logged, it can be set to 2, in which case the out of memory condition will cause a break in the debugger, or it can be set to 4 in which case a more extensive event log is written that includes memory statistics at the point where the out of memory condition was encountered.

Summary

Effective debugging of tricky application problems in the managed heap and garbage collector requires a solid internal understanding of how these components work. In this chapter we took a detailed tour of how the CLR heap manager and garbage collector functions. We started by looking at the high level architecture and how the CLR heap manager fits into the overall Windows memory architecture followed by an in-depth discussion of the various concepts (generations, roots, finalization, etc) utilized by the garbage collector. Sample code was shown in tandem with the debugger and associated tools to illustrate how these concepts work in practice. Lastly, we looked at a number of examples of common programming mistakes and how they manifest themselves in the CLR. The examples included how to track down the source of heap corruptions on the managed heap, how to track down the source of out of memory situations, and how to debug faulty finalization code.