CHAPTER 21

The Managed Heap and Garbage Collection

In this chapter:

In this chapter, I'll discuss how managed applications construct new objects, how the managed heap controls the lifetime of these objects, and how the memory for these objects gets reclaimed. In short, I'll explain how the garbage collector in the common language runtime (CLR) works, and I'll explain various performance issues related to it. I'll also discuss how to design applications so that they use memory most efficiently.

Managed Heap Basics

Every program uses resources of one sort or another, be they files, memory buffers, screen space, network connections, database resources, and so on. In fact, in an object-oriented environment, every type identifies some resource available for a program's use. To use any of these resources requires memory to be allocated to represent the type. The following steps are required to access a resource:

1. Allocate memory for the type that represents the resource (usually accomplished by using C#'s new operator).
2. Initialize the memory to set the initial state of the resource and to make the resource usable. The type's instance constructor is responsible for setting this initial state.
3. Use the resource by accessing the type's members (repeating as necessary).
4. Tear down the state of a resource to clean up.
5. Free the memory. The garbage collector is solely responsible for this step.

This seemingly simple paradigm has been one of the major sources of problems for programmers that must manually manage their memory; for example, native C++ developers. Programmers responsible for managing their own memory routinely forget to free memory, which causes a memory leak. In addition, these programmers frequently use memory after having released it, causing their program to experience memory corruption resulting in bugs and security holes. Furthermore, these two bugs are worse than most others because you can't predict the consequences or the timing of them. For other bugs, when you see your application misbehaving, you just fix the line of code that is not working.

As long as you are writing verifiably type-safe code (avoiding C#'s `unsafe` keyword), then it is impossible for your application to experience memory corruption. It is still possible for your application to leak memory but it is not the default behavior. Memory leaks typically occur because your application is storing objects in a collection and never removes objects when they are no longer needed.

To simplify things even more, most types that developers use quite regularly do not require Step 4 (tear down the state of the resource to clean up). And so, the managed heap, in addition to abolishing the bugs I mentioned, also provides developers with a simple programming model: allocate and initialize a resource and use it as desired. For most types, there is no need to clean up the resource and the garbage collector will free the memory.

When consuming instances of types that require special cleanup, the programming model remains as simple as I've just described. However, sometimes, you want to clean up a resource as soon as possible, rather than waiting for a GC to kick in. In these classes, you can call one additional method (called `Dispose`) in order to clean up the resource on your schedule. On the other hand, implementing a type that requires special cleanup is quite involved. I describe the details of all this in the "Working with Types Requiring Special Cleanup" section later in this chapter. Typically, types that require special cleanup are those that wrap native resources like files, sockets, or database connections.

Allocating Resources from the Managed Heap

The CLR requires that all objects be allocated from the *managed heap*. When a process is initialized, the CLR allocates a region of address space for the managed heap. The CLR also maintains a pointer, which I'll call `NextObjPtr`. This pointer indicates where the next object is to be allocated within the heap. Initially, `NextObjPtr` is set to the base address of the address space region.

As region fills with non-garbage objects, the CLR allocates more regions and continues to do this until the whole process's address space is full. So, your application's memory is limited by the process's virtual address space. In a 32-bit process, you can allocate close to 1.5 gigabytes (GB) and in a 64-bit process, you can allocate close to 8 terabytes.

C#'s `new` operator causes the CLR to perform the following steps:

1. Calculate the number of bytes required for the type's fields (and all the fields it inherits from its base types).
2. Add the bytes required for an object's overhead. Each object has two overhead fields: a type object pointer and a sync block index. For a 32-bit application, each of these fields requires

32 bits, adding 8 bytes to each object. For a 64-bit application, each field is 64 bits, adding 16 bytes to each object.

3. The CLR then checks that the bytes required to allocate the object are available in the region. If there is enough free space in the managed heap, the object will fit, starting at the address pointed to by `NextObjPtr`, and these bytes are zeroed out. The type's constructor is called (passing `NextObjPtr` for the `this` parameter), and the `new` operator returns a reference to the object. Just before the reference is returned, `NextObjPtr` is advanced past the object and now points to the address where the next object will be placed in the heap.

Figure 21-1 shows a managed heap consisting of three objects: A, B, and C. If another object were to be allocated, it would be placed where `NextObjPtr` points to (immediately after object C).

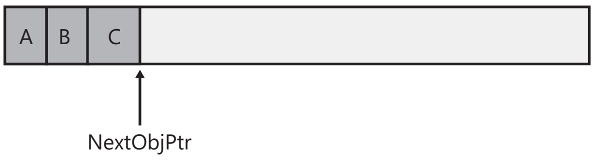


FIGURE 21-1 Newly initialized managed heap with three objects constructed in it.

For the managed heap, allocating an object simply means adding a value to a pointer—this is blazingly fast. In many applications, objects allocated around the same time tend to have strong relationships to each other and are frequently accessed around the same time. For example, it's very common to allocate a `FileStream` object immediately before a `BinaryWriter` object is created. Then the application would use the `BinaryWriter` object, which internally uses the `FileStream` object. Because the managed heap allocates these objects next to each other in memory, you get excellent performance when accessing these objects due to locality of reference. Specifically, this means that your process's working set is small, which means your application runs fast with less memory. It's also likely that the objects your code is accessing can all reside in the CPU's cache. The result is that your application will access these objects with phenomenal speed because the CPU will be able to perform most of its manipulations without having cache misses that would force slower access to RAM.

So far, it sounds like the managed heap provides excellent performance characteristics. However, what I have just described is assuming that memory is infinite and that the CLR can always allocate new objects at the end. However, memory is not infinite and so the CLR employs a technique known as garbage collection (GC) to "delete" objects in the heap that your application no longer requires access to.

The Garbage Collection Algorithm

When an application calls the `new` operator to create an object, there might not be enough address space left in the region to allocate the object. If insufficient space exists, then the CLR performs a GC.

Important What I've just said is an oversimplification. In reality, a GC occurs when generation 0 is full. I'll explain generations later in this chapter. Until then, it's easiest for you to think that a garbage collection occurs when the heap is full.

For managing the lifetime of objects, some systems use a reference counting algorithm. In fact, Microsoft's own Component Object Model (COM) uses reference counting. With a reference counting system, each object on the heap maintains an internal field indicating how many "parts" of the program are currently using that object. As each "part" gets to a place in the code where it no longer requires access to an object, it decrements that object's count field. When the count field reaches 0, the object deletes itself from memory. The big problem with many reference counting systems is that they do not handle circular references well. For example, in a GUI application, a window will hold a reference to a child UI element. And the child UI element will hold a reference to its parent window. These references prevent the two objects' counters from reaching 0, so both objects will never be deleted even if the application itself no longer has a need for the window.

Due to this problem with reference counting garbage collector algorithms, the CLR uses a referencing tracking algorithm instead. The reference tracking algorithm cares only about reference type variables, because only these variables can refer to an object on the heap; value type variables contain the value type instance directly. Reference type variables can be used in many contexts: static and instance fields within a class or a method's arguments or local variables. We refer to all reference type variables as *roots*.

When the CLR starts a GC, the CLR first suspends all threads in the process. This prevents threads from accessing objects and changing their state while the CLR examines them. Then, the CLR performs what is called the *marking* phase of the GC. First, it walks through all the objects in the heap setting a bit (contained in the sync block index field) to 0. This indicates that all objects should be deleted. Then, the CLR looks at all active roots to see which objects they refer to. This is what makes the CLR's GC a reference tracking GC. If a root contains `null`, the CLR ignores the root and moves on to examine the next root.

Any root referring to an object on the heap causes the CLR to mark that object. Marking an object means that the CLR sets the bit in the object's sync block index to 1. When an object is marked, the CLR examines the roots inside that object and marks the objects they refer to. If the CLR is about to mark an already-marked object, then it does not examine the object's fields again. This prevents an infinite loop from occurring in the case where you have a circular reference.

Figure 21-2 shows a heap containing several objects. In this example, the application roots refer directly to objects A, C, D, and F. All of these objects are marked. When marking object D, the garbage collector notices that this object contains a field that refers to object H, causing object H to be marked as well. The marking phase continues until all the application roots have been examined.

Once complete, the heap contains some marked and some unmarked objects. The marked objects must survive the collection because there is at least one root that refers to the object; we say that the object is reachable because application code can reach (or access) the object by way of the variable that still refers to it. Unmarked objects are unreachable because there is no root existing in the application that would allow for the object to ever be accessed again.

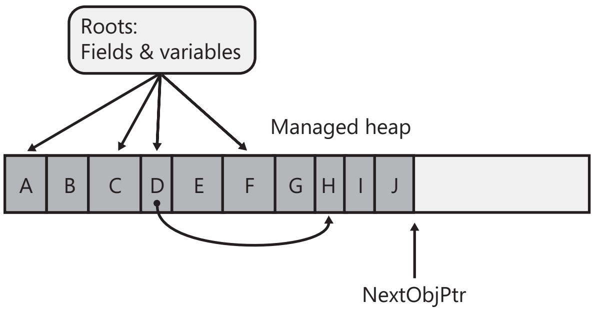


FIGURE 21-2 Managed heap before a collection.

Now that the CLR knows which objects must survive and which objects can be deleted, it begins the GC's compacting phase. During the compacting phase, the CLR shifts the memory consumed by the marked objects down in the heap, compacting all the surviving objects together so that they are contiguous in memory. This serves many benefits. First, all the surviving objects will be next to each other in memory; this restores locality of reference reducing your application's working set size, thereby improving the performance of accessing these objects in the future. Second, the free space is all contiguous as well, so this region of address space can be freed, allowing other things to use it. Finally, compaction means that there are no address space fragmentation issues with the managed heap as is known to happen with native heaps.[1]

When compacting memory, the CLR is moving objects around in memory. This is a problem because any root that referred to a surviving object now refers to where that object was in memory; not where the object has been relocated to. When the application's threads eventually get resumed, they would access the old memory locations and corrupt memory. Clearly, this can't be allowed and so, as part of the compacting phase, the CLR subtracts from each root the number of bytes that the object it referred to was shifted down in memory. This ensures that every root refers to the same object it did before; it's just that the object is at a different location in memory.

After the heap memory is compacted, the managed heap's `NextObjPtr` pointer is set to point to a location just after the last surviving object. This is where the next allocated object will be placed in memory. Figure 21-3 shows the managed heap after the compaction phase. After the compaction phase is complete, the CLR resumes all the application's threads and they continue to access the objects as if the GC never happened at all.

[1] Objects in the large object heap (discussed later in this chapter) do not get compacted, and therefore address space fragmentation is possible with the large object heap.

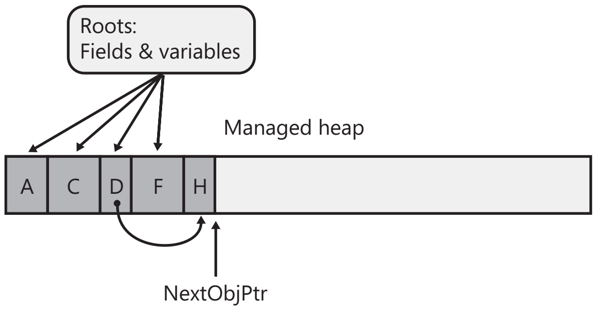


FIGURE 21-3 Managed heap after a collection.

If the CLR is unable to reclaim any memory after a GC and if there is no address space left in the processes to allocate a new GC segment, then there is just no more memory available for this process. In this case, the new operator that attempted to allocate more memory ends up throwing an `OutOfMemoryException`. Your application can catch this and recover from it but most applications do not attempt to do so; instead, the exception becomes an unhandled exception, Windows terminates the process, and then Windows reclaims all the memory that the process was using.

As a programmer, notice how the two bugs described at the beginning of this chapter no longer exist. First, it's not possible to leak objects because any object not accessible from your application's roots will be collected at some point. Second, it's not possible to corrupt memory by accessing an object that was freed because references can only refer to living objects, because this is what keeps the objects alive anyway.

Important A static field keeps whatever object it refers to forever or until the AppDomain that the types are loaded into is unloaded. A common way to leak memory is to have a static field refer to a collection object and then to keep adding items to the collection object. The static field keeps the collection object alive and the collection object keeps all its items alive. For this reason, it is best to avoid static fields whenever possible.

Garbage Collections and Debugging

As soon as a root goes out of scope, the object it refers to is unreachable and subject to having its memory reclaimed by a GC; objects aren't guaranteed to live throughout a method's lifetime. This can have an interesting impact on your application. For example, examine the following code.

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
       // Create a Timer object that knows to call our TimerCallback
       // method once every 2000 milliseconds.
       Timer t = new Timer(TimerCallback, null, 0, 2000);
```

```
      // Wait for the user to hit <Enter>.
      Console.ReadLine();
   }

   private static void TimerCallback(Object o) {
      // Display the date/time when this method got called.
      Console.WriteLine("In TimerCallback: " + DateTime.Now);

      // Force a garbage collection to occur for this demo.
      GC.Collect();
   }
}
```

Compile this code from the command prompt without using any special compiler switches. When you run the resulting executable file, you'll see that the `TimerCallback` method is called just once!

From examining the preceding code, you'd think that the `TimerCallback` method would get called once every 2,000 milliseconds. After all, a `Timer` object is created, and the variable `t` refers to this object. As long as the timer object exists, the timer should keep firing. But you'll notice in the `TimerCallback` method that I force a garbage collection to occur by calling `GC.Collect()`.

When the collection starts, it first assumes that all objects in the heap are unreachable (garbage); this includes the `Timer` object. Then, the collector examines the application's roots and sees that `Main` doesn't use the `t` variable after the initial assignment to it. Therefore, the application has no variable referring to the `Timer` object, and the garbage collection reclaims the memory for it; this stops the timer and explains why the `TimerCallback` method is called just once.

Let's say that you're using a debugger to step through `Main`, and a garbage collection just happens to occur just after `t` is assigned the address of the new `Timer` object. Then, let's say that you try to view the object that `t` refers to by using the debugger's Quick Watch window. What do you think will happen? The debugger can't show you the object because it was just garbage collected. This behavior would be considered very unexpected and undesirable by most developers, so Microsoft has come up with a solution.

When you compile your assembly by using the C# compiler's `/debug` switch, the compiler applies a `System.Diagnostics.DebuggableAttribute` with its `DebuggingModes`' `DisableOptimizations` flag set into the resulting assembly. At run time, when compiling a method, the JIT compiler sees this flag set, and artificially extends the lifetime of all roots to the end of the method. For my example, the JIT compiler tricks itself into believing that the `t` variable in `Main` must live until the end of the method. So, if a garbage collection were to occur, the garbage collector now thinks that `t` is still a root and that the `Timer` object that `t` refers to will continue to be reachable. The `Timer` object will survive the collection, and the `TimerCallback` method will get called repeatedly until `Console.ReadLine` returns and `Main` exits.

To see this, just recompile the program from a command prompt, but this time, specify the C# compiler's `/debug` switch. When you run the resulting executable file, you'll now see that the `TimerCallback` method is called repeatedly! Note, the C# compiler's `/optimize+` compiler switch turns optimizations back on, so this compiler switch should not be specified when performing this experiment.

The JIT compiler does this to help you with JIT debugging. You may now start your application normally (without a debugger), and if the method is called, the JIT compiler will artificially extend the lifetime of the variables to the end of the method. Later, if you decide to attach a debugger to the process, you can put a breakpoint in a previously compiled method and examine the root variables.

So now you know how to build a program that works in a debug build but doesn't work correctly when you make a release build! Because no developer wants a program that works only when debugging it, there should be something we can do to the program so that it works all of the time regardless of the type of build.

You could try modifying the `Main` method to the following.

```
public static void Main() {
    // Create a Timer object that knows to call our TimerCallback
    // method once every 2000 milliseconds.
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Wait for the user to hit <Enter>.
    Console.ReadLine();

    // Refer to t after ReadLine (this gets optimized away)
    t = null;
}
```

However, if you compile this (without the `/debug+` switch) and run the resulting executable file, you'll see that the `TimerCallback` method is still called just once. The problem here is that the JIT compiler is an optimizing compiler, and setting a local variable or parameter variable to `null` is the same as not referencing the variable at all. In other words, the JIT compiler optimizes the `t = null;` line out of the code completely, and therefore, the program still does not work as we desire. The correct way to modify the `Main` method is as follows.

```
public static void Main() {
    // Create a Timer object that knows to call our TimerCallback
    // method once every 2000 milliseconds.
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Wait for the user to hit <Enter>.
    Console.ReadLine();

    // Refer to t after ReadLine (t will survive GCs until Dispose returns)
    t.Dispose();
}
```

Now, if you compile this code (without the `/debug+` switch) and run the resulting executable file, you'll see that the `TimerCallback` method is called multiple times, and the program is fixed. What's happening here is that the object `t` is required to stay alive so that the `Dispose` instance method can be called on it. (The value in `t` needs to be passed as the `this` argument to `Dispose`.) It's ironic: by explicitly indicating where you want the timer to be disposed, it must remain alive up to that point.

Note Please don't read this whole discussion and then worry about your own objects being garbage collected prematurely. I use the `Timer` class in this discussion because it has special behavior that no other class exhibits. The "problem/feature" of `Timer` is that the existence of a `Timer` object in the heap causes something else to happen: A thread pool thread invokes a method periodically. No other type exhibits this behavior. For example, the existence of a `String` object in memory doesn't cause anything else to happen; the string just sits there. So, I use `Timer` to show how roots work and how object-lifetime works as related to the debugger, but the discussion is not really about how to keep objects alive. All non-`Timer` objects will live as needed by the application automatically.

Generations: Improving Performance

The CLR's GC is a *generational garbage collector* (also known as an *ephemeral garbage collector*, although I don't use the latter term in this book). A generational GC makes the following assumptions about your code:

- The newer an object is, the shorter its lifetime will be.
- The older an object is, the longer its lifetime will be.
- Collecting a portion of the heap is faster than collecting the whole heap.

Numerous studies have demonstrated the validity of these assumptions for a very large set of existing applications, and these assumptions have influenced how the garbage collector is implemented. In this section, I'll describe how generations work.

When initialized, the managed heap contains no objects. Objects added to the heap are said to be in generation 0. Stated simply, objects in generation 0 are newly constructed objects that the garbage collector has never examined. Figure 21-4 shows a newly started application with five objects allocated (A through E). After a while, objects C and E become unreachable.

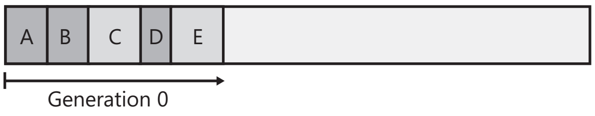


FIGURE 21-4 A newly initialized heap containing some objects, all in generation 0. No collections have occurred yet.

When the CLR initializes, it selects a budget size (in kilobytes) for generation 0. So if allocating a new object causes generation 0 to surpass its budget, a garbage collection must start. Let's say that objects A through E fill all of generation 0. When object F is allocated, a garbage collection must start.

The garbage collector will determine that objects C and E are garbage and will compact object D, causing it to be adjacent to object B. The objects that survive the garbage collection (objects A, B, and D) are said to be in generation 1. Objects in generation 1 have been examined by the garbage collector once. The heap now looks like Figure 21-5.

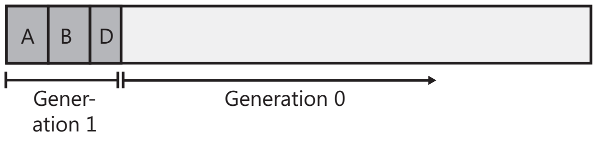


FIGURE 21-5 After one collection, generation 0 survivors are promoted to generation 1; generation 0 is empty.

After a garbage collection, generation 0 contains no objects. As always, new objects will be allocated in generation 0. Figure 21-6 shows the application running and allocating objects F through K. In addition, while the application was running, objects B, H, and J became unreachable and should have their memory reclaimed at some point.

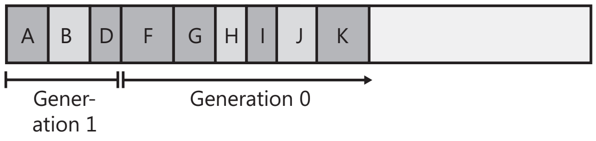


FIGURE 21-6 New objects are allocated in generation 0; generation 1 has some garbage.

Now let's say that attempting to allocate object L would put generation 0 over its budget. Because generation 0 has reached its budget, a garbage collection must start. When starting a garbage collection, the garbage collector must decide which generations to examine. Earlier, I said that when the CLR initializes, it selects a budget for generation 0. Well, it also selects a budget for generation 1.

When starting a garbage collection, the garbage collector also sees how much memory is occupied by generation 1. In this case, generation 1 occupies much less than its budget, so the garbage collector examines only the objects in generation 0. Look again at the assumptions that the generational garbage collector makes. The first assumption is that newly created objects have a short lifetime. So generation 0 is likely to have a lot of garbage in it, and collecting generation 0 will therefore reclaim a lot of memory. The garbage collector will just ignore the objects in generation 1, which will speed up the garbage collection process.

Obviously, ignoring the objects in generation 1 improves the performance of the garbage collector. However, the garbage collector improves performance more because it doesn't traverse every object in the managed heap. If a root or an object refers to an object in an old generation, the garbage collector can ignore any of the older objects' inner references, decreasing the amount of time required to build the graph of reachable objects. Of course, it's possible that an old object's field refers to a new object. To ensure that the updated fields of these old objects are examined, the garbage collector uses a mechanism internal to the JIT compiler that sets a bit when an object's reference field changes. This support lets the garbage collector know which old objects (if any) have been written to because the last collection. Only old objects that have had fields change need to be examined to see whether they refer to any new object in generation 0.[2]

Note Microsoft's performance tests show that it takes less than 1 millisecond to perform a garbage collection of generation 0. Microsoft's goal is to have garbage collections take no more time than an ordinary page fault.

A generational garbage collector also assumes that objects that have lived a long time will continue to live. So it's likely that the objects in generation 1 will continue to be reachable from the application. Therefore, if the garbage collector were to examine the objects in generation 1, it probably wouldn't find a lot of garbage. As a result, it wouldn't be able to reclaim much memory. So it is likely that collecting generation 1 is a waste of time. If any garbage happens to be in generation 1, it just stays there. The heap now looks like Figure 21-7.

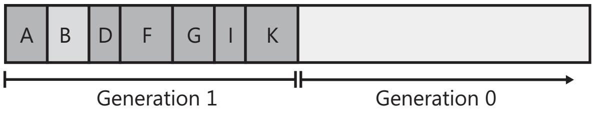


FIGURE 21-7 After two collections, generation 0 survivors are promoted to generation 1 (growing the size of generation 1); generation 0 is empty.

As you can see, all of the generation 0 objects that survived the collection are now part of generation 1. Because the garbage collector didn't examine generation 1, object B didn't have its memory reclaimed even though it was unreachable at the time of the last garbage collection. Again, after a collection, generation 0 contains no objects and is where new objects will be placed. In fact, let's say that the application continues running and allocates objects L through O. And while running, the application stops using objects G, L, and M, making them all unreachable. The heap now looks like Figure 21-8.

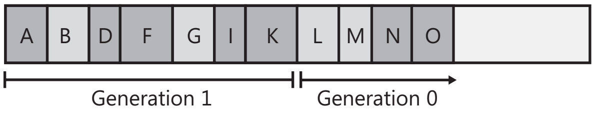


FIGURE 21-8 New objects are allocated in generation 0; generation 1 has more garbage.

Let's say that allocating object P causes generation 0 to exceed its budget, causing a garbage collection to occur. Because the memory occupied by all of the objects in generation 1 is less than its

[2] For the curious, here are some more details about this. When the JIT compiler produces native code that modifies a reference field inside an object, the native code includes a call to a `write barrier` method. This `write barrier` method checks whether the object whose field is being modified is in generation 1 or 2 and if it is, the `write barrier` code sets a bit in what is called the card table. The card table has 1 bit for every 128-byte range of data in the heap. When the next GC starts, it scans the card table to know which objects in generations 1 and 2 have had their fields changed because the last GC. If any of these modified objects refer to an object in generation 0, then the generation 0 objects survive the collection. After the GC, the card table is reset to all zeroes. The `write barrier` code causes a slight performance hit when writing to a reference field in an object (as opposed to a local variable or static field) and that performance hit is slightly worse if that object is in generation 1 or 2.

budget, the garbage collector again decides to collect only generation 0, ignoring the unreachable objects in generation 1 (objects B and G). After the collection, the heap looks like Figure 21-9.

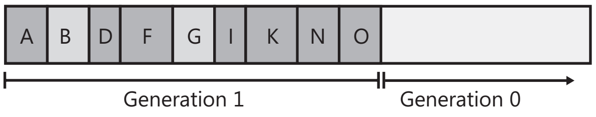


FIGURE 21-9 After three collections, generation 0 survivors are promoted to generation 1 (growing the size of generation 1 again); generation 0 is empty.

In Figure 21-9, you see that generation 1 keeps growing slowly. In fact, let's say that generation 1 has now grown to the point in which all of the objects in it occupy its full budget. At this point, the application continues running (because a garbage collection just finished) and starts allocating objects P through S, which fill generation 0 up to its budget. The heap now looks like Figure 21-10.

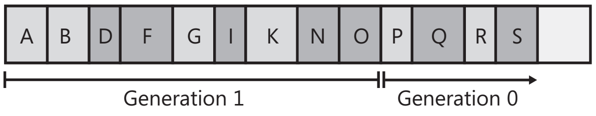


FIGURE 21-10 New objects are allocated in generation 0; generation 1 has more garbage.

When the application attempts to allocate object T, generation 0 is full, and a garbage collection must start. This time, however, the garbage collector sees that the objects in generation 1 are occupying so much memory that generation 1's budget has been reached. Over the several generation 0 collections, it's likely that a number of objects in generation 1 have become unreachable (as in our example). So this time, the garbage collector decides to examine all of the objects in generation 1 and generation 0. After both generations have been garbage collected, the heap now looks like Figure 21-11.

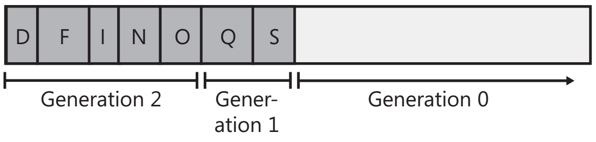


FIGURE 21-11 After four collections: generation 1 survivors are promoted to generation 2, generation 0 survivors are promoted to generation 1, and generation 0 is empty.

As before, any objects that were in generation 0 that survived the garbage collection are now in generation 1; any objects that were in generation 1 that survived the collection are now in generation 2. As always, generation 0 is empty immediately after a garbage collection and is where new objects will be allocated. Objects in generation 2 are objects that the garbage collector has examined two or more times. There might have been several collections, but the objects in generation 1 are examined only when generation 1 reaches its budget, which usually requires several garbage collections of generation 0.

The managed heap supports only three generations: generation 0, generation 1, and generation 2; there is no generation 3.[3] When the CLR initializes, it selects budgets for all three generations. However, the CLR's garbage collector is a self-tuning collector. This means that the garbage collector learns about your application's behavior whenever it performs a garbage collection. For example, if your application constructs a lot of objects and uses them for a very short period of time, it's possible that garbage collecting generation 0 will reclaim a lot of memory. In fact, it's possible that the memory for all objects in generation 0 can be reclaimed.

If the garbage collector sees that there are very few surviving objects after collecting generation 0, it might decide to reduce the budget of generation 0. This reduction in the allotted space will mean that garbage collections occur more frequently but will require less work for the garbage collector, so your process's working set will be small. In fact, if all objects in generation 0 are garbage, a garbage collection doesn't have to compact any memory; it can simply set `NextObjPtr` back to the beginning of generation 0, and then the garbage collection is performed. Wow, this is a fast way to reclaim memory!

Note The garbage collector works extremely well for applications with threads that sit idle at the top of their stack most of the time. Then, when the thread has something to do, it wakes up, creates a bunch of short-lived objects, returns, and then goes back to sleep. Many applications follow this architecture. For example, GUI applications tend to have the GUI thread sitting in a message loop most of its life. Occasionally, the user generates some input (like a touch, mouse, or keyboard event), the thread wakes up, processes the input and returns back to the message pump. Most objects created to process the input are probably garbage now.

Similarly, server applications tend to have thread pool threads sitting in the pool waiting for client requests to come in. When a client request comes in, new objects are created to perform work on behalf of the client request. When the result is sent back to the client, the thread returns to the thread pool and all the objects it created are garbage now.

On the other hand, if the garbage collector collects generation 0 and sees that there are a lot of surviving objects, not a lot of memory was reclaimed in the garbage collection. In this case, the garbage collector will grow generation 0's budget. Now, fewer collections will occur, but when they do, a lot more memory should be reclaimed. By the way, if insufficient memory has been reclaimed after a collection, the garbage collector will perform a full collection before throwing an `OutOfMemoryException`.

Throughout this discussion, I've been talking about how the garbage collector dynamically modifies generation 0's budget after every collection. But the garbage collector also modifies the budgets of generation 1 and generation 2 by using similar heuristics. When these generations are garbage collected, the garbage collector again sees how much memory is reclaimed and how many objects survived. Based on the garbage collector's findings, it might grow or shrink the thresholds of these

[3] The `System.GC` class's static `MaxGeneration` method returns 2.

generations as well to improve the overall performance of the application. The end result is that the garbage collector fine-tunes itself automatically based on the memory load required by your application—this is very cool!

The following `GCNotification` class raises an event whenever a generation 0 or generation 2 collection occurs. With these events, you could have the computer beep whenever a collection occurs or you calculate how much time passes between collections, how much memory is allocated between collections, and more. With this class, you could easily instrument your application to get a better understanding of how your application uses memory.

```
public static class GCNotification {
   private static Action<Int32> s_gcDone = null;  // The event's field

   public static event Action<Int32> GCDone {
      add {
         // If there were no registered delegates before, start reporting notifications now
         if (s_gcDone == null) { new GenObject(0); new GenObject(2); }
         s_gcDone += value;
      }
      remove { s_gcDone -= value; }
   }

   private sealed class GenObject {
      private Int32 m_generation;
      public GenObject(Int32 generation) { m_generation = generation; }
      ~GenObject() { // This is the Finalize method
         // If this object is in the generation we want (or higher),
         // notify the delegates that a GC just completed
         if (GC.GetGeneration(this) >= m_generation) {
            Action<Int32> temp = Volatile.Read(ref s_gcDone);
            if (temp != null) temp(m_generation);
         }

         // Keep reporting notifications if there is at least one delegate registered,
         // the AppDomain isn't unloading, and the process isn’t shutting down
         if ((s_gcDone != null)
            && !AppDomain.CurrentDomain.IsFinalizingForUnload()
            && !Environment.HasShutdownStarted) {
               // For Gen 0, create a new object; for Gen 2, resurrect the object
               // & let the GC call Finalize again the next time Gen 2 is GC'd
            if (m_generation == 0) new GenObject(0);
            else GC.ReRegisterForFinalize(this);
         } else { /* Let the objects go away */ }
      }
   }
}
```

Garbage Collection Triggers

As you know, the CLR triggers a GC when it detects that generation 0 has filled its budget. This is the most common trigger of a GC; however, there are additional GC triggers as listed here:

- **Code explicitly calls `System.GC`'s static Collect method** Code can explicitly request that the CLR perform a collection. Although Microsoft strongly discourages such requests, at times it might make sense for an application to force a collection. I discuss this more in the "Forcing Garbage Collections" section later in this chapter.

- **Windows is reporting low memory conditions** The CLR internally uses the Win32 `CreateMemoryResourceNotification` and `QueryMemoryResourceNotification` functions to monitor system memory overall. If Windows reports low memory, the CLR will force a garbage collection in an effort to free up dead objects to reduce the size of a process's working set.

- **The CLR is unloading an AppDomain** When an AppDomain unloads, the CLR considers nothing in the AppDomain to be a root, and a garbage collection consisting of all generations is performed. I'll discuss AppDomains in Chapter 22, "CLR Hosting and AppDomains."

- **The CLR is shutting down** The CLR shuts down when a process terminates normally (as opposed to an external shutdown via Task Manager, for example). During this shutdown, the CLR considers nothing in the process to be a root; it allows objects a chance to clean up but the CLR does not attempt to compact or free memory because the whole process is terminating, and Windows will reclaim all of the processes' memory.

Large Objects

There is one more performance improvement you might want to be aware of. The CLR considers each single object to be either a small object or a large object. So far, in this chapter, I've been focusing on small objects. Today, a large object is 85,000 bytes or more in size.[4] The CLR treats large objects slightly differently than how it treats small objects:

- Large objects are not allocated within the same address space as small objects; they are allocated elsewhere within the process' address space.

- Today, the GC doesn't compact large objects because of the time it would require to move them in memory. For this reason, address space fragmentation can occur between large objects within the process leading to an `OutOfMemoryException` being thrown. In a future version of the CLR, large objects may participate in compaction.

[4] In the future, the CLR could change the number of bytes required to consider an object to be a large object. Do not count 85,000 being a constant.

- Large objects are immediately considered to be part of generation 2; they are never in generation 0 or 1. So, you should create large objects only for resources that you need to keep alive for a long time. Allocating short-lived large objects will cause generation 2 to be collected more frequently, hurting performance. Usually large objects are large strings (like XML or JSON) or byte arrays that you use for I/O operations, such as reading bytes from a file or network into a buffer so you can process it.

For the most part, large objects are transparent to you; you can simply ignore that they exist and that they get special treatment until you run into some unexplained situation in your program (like why you're getting address space fragmentation).

Garbage Collection Modes

When the CLR starts, it selects a GC mode, and this mode cannot change during the lifetime of the process. There are two basic GC modes:

- **Workstation** This mode fine-tunes the garbage collector for client-side applications. It is optimized to provide for low-latency GCs in order to minimize the time an application's threads are suspended so as not to frustrate the end user. In this mode, the GC assumes that other applications are running on the machine and does not hog CPU resources.

- **Server** This mode fine-tunes the garbage collector for server-side applications. It is optimized for throughput and resource utilization. In this mode, the GC assumes no other applications (client or server) are running on the machine, and it assumes that all the CPUs on the machine are available to assist with completing the GC. This GC mode causes the managed heap to be split into several sections, one per CPU. When a garbage collection is initiated, the garbage collector dedicates one special thread per CPU; each thread collects its own section in parallel with the other threads. Parallel collections work well for server applications in which the worker threads tend to exhibit uniform behavior. This feature requires the application to be running on a computer with multiple CPUs so that the threads can truly be working simultaneously to attain a performance improvement.

By default, applications run with the Workstation GC mode. A server application (such as ASP.NET or Microsoft SQL Server) that hosts the CLR can request the CLR to load the Server GC. However, if the server application is running on a uniprocessor machine, then the CLR will always use Workstation GC mode. A stand-alone application can tell the CLR to use the Server GC mode by creating a configuration file (as discussed in Chapter 2, "Building, Packaging, Deploying, and Administering Applications and Types," and Chapter 3, "Shared Assemblies and Strongly Named Assemblies") that contains a `gcServer` element for the application. Here's an example of a configuration file.

```
<configuration>
    <runtime>
        <gcServer enabled="true"/>
    </runtime>
</configuration>
```

When an application is running, it can ask the CLR if it is running in the Server GC mode by querying the GCSettings class's IsServerGC read-only Boolean property.

```
using System;
using System.Runtime; // GCSettings is in this namespace

public static class Program {
   public static void Main() {
      Console.WriteLine("Application is running with server GC=" + GCSettings.IsServerGC);
    }
}
```

In addition to the two modes, the GC can run in two sub-modes: concurrent (the default) or non-concurrent. In concurrent mode, the GC has an additional background thread that marks objects concurrently while the application runs. When a thread allocates an object that pushes generation 0 over its budget, the GC first suspends all threads and then determines which generations to collect. If the garbage collector needs to collect generation 0 or 1, it proceeds as normal. However, if generation 2 needs collecting, the size of generation 0 will be increased beyond its budget to allocate the new object, and then the application's threads are resumed.

While the application's threads are running, the garbage collector has a normal priority background thread that finds unreachable objects. Once found, the garbage collector suspends all threads again and decides whether to compact memory. If the garbage collector decides to compact memory, memory is compacted, root references are fixed up, and the application's threads are resumed. This garbage collection takes less time than usual because the set of unreachable objects has already been built. However, the garbage collector might decide not to compact memory; in fact, the garbage collector favors this approach. If you have a lot of free memory, the garbage collector won't compact the heap; this improves performance but grows your application's working set. When using the concurrent garbage collector, you'll typically find that your application is consuming more memory than it would with the non-concurrent garbage collector.

You can tell the CLR not to use the concurrent collector by creating a configuration file for the application that contains a gcConcurrent element. Here's an example of a configuration file.

```
<configuration>
   <runtime>
      <gcConcurrent enabled="false"/>
   </runtime>
</configuration>
```

The GC mode is configured for a process and it cannot change while the process runs. However, your application can have some control over the garbage collection by using the GCSettings class's GCLatencyMode property. This read/write property can be set to any of the values in the GCLatencyMode enumerated type, as shown in Table 21-1.

The LowLatency mode requires some additional explanation. Typically, you would set this mode, perform a short-term, time-sensitive operation, and then set the mode back to either Batch or Interactive. While the mode is set to LowLatency, the GC will really avoid doing any generation 2 collections because these could take a long time. Of course, if you call GC.Collect(), then generation 2 still gets collected. Also, the GC will perform a generation 2 collection if Windows tells the CLR that system memory is low (see the "Garbage Collection Triggers" section earlier in this chapter).

TABLE 21-1 Symbols Defined by the GCLatencyMode Enumerated Type

Symbol Name	Description
Batch (default for the Server GC mode)	Turns off the concurrent GC.
Interactive (default for the Workstation GC mode)	Turns on the concurrent GC.
LowLatency	Use this latency mode during short-term, time-sensitive operations (like drawing animations) where a generation 2 collection might be disruptive.
SustainedLowLatency	Use this latency mode to avoid long GC pauses for the bulk of your application's execution. This setting prevents all blocking generation 2 collections from occurring as long as memory is available. In fact, users of these applications would prefer to install more RAM in the machine in order to avoid GC pauses. A stock market application that must respond immediately to price changes is an example of this kind of application.

Under LowLatency mode, it is more likely that your application could get an OutOfMemoryException thrown. Therefore, stay in this mode for as short a time as possible, avoid allocating many objects, avoid allocating large objects, and set the mode back to Batch or Interactive by using a constrained execution region (CER), as discussed in Chapter 20, "Exceptions and State Management." Also, remember that the latency mode is a process-wide setting and threads may be running concurrently. These other threads could even change this setting while another thread is using it, so you may want to update some kind of counter (manipulated via Interlocked methods) when you have multiple threads manipulating this setting. Here is some code showing how to use the LowLatency mode.

```
private static void LowLatencyDemo() {
   GCLatencyMode oldMode = GCSettings.LatencyMode;
   System.Runtime.CompilerServices.RuntimeHelpers.PrepareConstrainedRegions();
   try {
      GCSettings.LatencyMode = GCLatencyMode.LowLatency;
      // Run your code here...
   }
   finally {
      GCSettings.LatencyMode = oldMode;
   }
}
```

Forcing Garbage Collections

The System.GC type allows your application some direct control over the garbage collector. For starters, you can query the maximum generation supported by the managed heap by reading the GC.MaxGeneration property; this property always returns 2.

You can also force the garbage collector to perform a collection by calling GC class's Collect method, optionally passing in a generation to collect up to, a GCCollectionMode, and a Boolean indicating whether you want to perform a blocking (non-current) or background (concurrent) collection. Here is the signature of the most complex overload of the `Collect` method.

```
void Collect(Int32 generation, GCCollectionMode mode, Boolean blocking);
```

The `GCCollectionMode` type is an enum whose values are described in Table 21-2.

TABLE 21-2 Symbols Defined by the `GCCollectionMode` Enumerated Type

Symbol Name	Description
`Default`	The same as calling `GC.Collect` with no flag. Today, this is the same as passing Forced, but this may change in a future version of the CLR.
`Forced`	Forces a collection to occur immediately for all generations up to and including the specified generation.
`Optimized`	The garbage collector will only perform a collection if the collection would be productive either by freeing a lot of memory or by reducing fragmentation. If the garbage collection would not be productive, then the call has no effect

Under most circumstances, you should avoid calling any of the `Collect` methods; it's best just to let the garbage collector run on its own accord and fine-tune its generation budgets based on actual application behavior. However, if you're writing a console user interface (CUI) or GUI application, your application code owns the process and the CLR in that process. For these application types, you *might* want to suggest a garbage collection to occur at certain times using a `GCCollectionMode` of `Optimized`. Normally, modes of `Default` and `Forced` are used for debugging, testing, and looking for memory leaks.

For example, you might consider calling the `Collect` method if some non-recurring event has just occurred that has likely caused a lot of old objects to die. The reason that calling `Collect` in such a circumstance may not be so bad is that the GC's predictions of the future based on the past are not likely to be accurate for non-recurring events. For example, it might make sense for your application to force a full GC of all generations after your application initializes or after the user saves a data file. Because calling `Collect` causes the generation budgets to adjust, do not call `Collect` to try to improve your application's response time; call it to reduce your process's working set.

For some applications (especially server applications that tend to keep a lot of objects in memory), the time required for the GC to do a full collection that includes generation 2 can be excessive. In fact, if the collection takes a very long time to complete, then client requests might time out. To help these kinds of applications, the `GC` class offers a `RegisterForFullGCNotification` method. Using this method and some additional helper methods (`WaitForFullGCApproach`, `WaitForFullGCComplete`, and `CancelFullGCNotification`), an application can now be notified when the garbage collector is getting close to performing a full collection. The application can then call `GC.Collect` to force a collection at a more opportune time, or the application could communicate with another server to better load balance the client requests. For more information, examine these methods and the "Garbage Collection Notifications" topic in the Microsoft .NET Framework SDK documentation. Note that you should always call the `WaitForFullGCApproach` and `WaitForFullGCComplete` methods in pairs because the CLR handles them as pairs internally.

Monitoring Your Application's Memory Usage

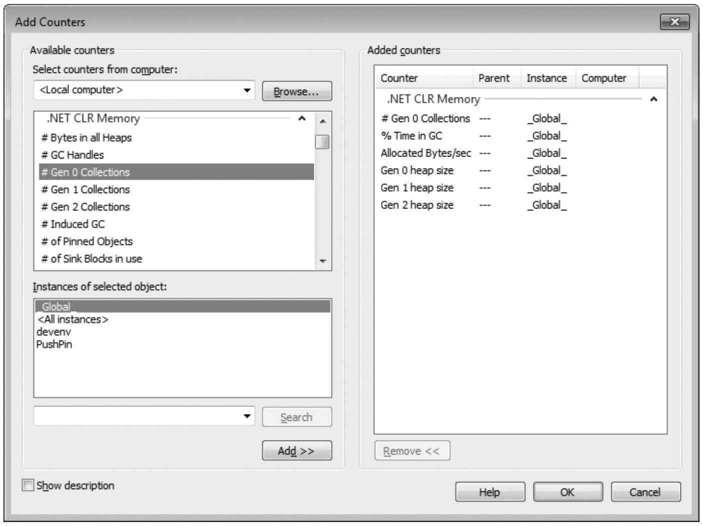
Within a process, there are a few methods that you can call to monitor the garbage collector. Specifically, the GC class offers the following static methods, which you can call to see how many collections have occurred of a specific generation or how much memory is currently being used by objects in the managed heap.

```
Int32 CollectionCount(Int32 generation);
Int64 GetTotalMemory(Boolean forceFullCollection);
```

To profile a particular code block, I have frequently written code to call these methods before and after the code block and then calculate the difference. This gives me a very good indication of how my code block has affected my process's working set and indicates how many garbage collections occurred while executing the code block. If the numbers are high, I know to spend more time tuning the algorithms in my code block.

You can also see how much memory is being used by individual AppDomains as opposed to the whole process. For more information about this, see the "AppDomain Monitoring" section in Chapter 22.

When you install the .NET Framework, it installs a set of performance counters that offer a lot of real-time statistics about the CLR's operations. These statistics are visible via the PerfMon.exe tool or the System Monitor ActiveX control that ships with Windows. The easiest way to access the System Monitor control is to run PerfMon.exe and click the + toolbar button, which causes the Add Counters dialog box shown in Figure 21-12 to appear.



Select counters from computer: <Local computer>

.NET CLR Memory
Bytes in all Heaps
GC Handles
Gen 0 Collections
Gen 1 Collections
Gen 2 Collections
Induced GC
# of Pinned Objects
# of Sink Blocks in use

Instances of selected object:
Global
<All instances>
devenv
PushPin

Counter	Parent	Instance	Computer
.NET CLR Memory			
# Gen 0 Collections	---	_Global_	
% Time in GC	---	_Global_	
Allocated Bytes/sec	---	_Global_	
Gen 0 heap size	---	_Global_	
Gen 1 heap size	---	_Global_	
Gen 2 heap size	---	_Global_	

FIGURE 21-12 PerfMon.exe showing the .NET CLR Memory counters.

To monitor the CLR's garbage collector, select the .NET CLR Memory performance object. Then select a specific application from the instance list box. Finally, select the set of counters that you're interested in monitoring, click Add, and then click OK. At this point, the System Monitor will graph the selected real-time statistics. For an explanation of a particular counter, select the desired counter and then select the Show Description check box.

Another great tool for analyzing the memory and performance of your application is PerfView. This tool can collect Event Tracing for Windows (ETW) logs and process them. The best way to acquire this tool is for you to search the web for PerfView. Finally, you should look into using the SOS Debugging Extension (SOS.dll), which can often offer great assistance when debugging memory problems and other CLR problems. For memory-related actions, the SOS Debugging Extension allows you to see how much memory is allocated within the process to the managed heap, displays all objects registered for finalization in the finalization queue, displays the entries in the `GCHandle` table per AppDomain or for the entire process, shows the roots that are keeping an object alive in the heap, and more.

Working with Types Requiring Special Cleanup

At this point, you should have a basic understanding of garbage collection and the managed heap, including how the garbage collector reclaims an object's memory. Fortunately for us, most types need only memory to operate. However, some types require more than just memory to be useful; some types require the use of a native resource in addition to memory.

The `System.IO.FileStream` type, for example, needs to open a file (a native resource) and store the file's handle. Then the type's `Read` and `Write` methods use this handle to manipulate the file. Similarly, the `System.Threading.Mutex` type opens a Windows mutex kernel object (a native resource) and stores its handle, using it when the `Mutex`'s methods are called.

If a type wrapping a native resource gets GC'd, the GC will reclaim the memory used by the object in the managed heap; but the native resource, which the GC doesn't know anything about, will be leaked. This is clearly not desirable, so the CLR offers a mechanism called *finalization*. Finalization allows an object to execute some code after the object has been determined to be garbage but before the object's memory is reclaimed from the managed heap. All types that wrap a native resource—such as a file, network connection, socket, or mutex—support finalization. When the CLR determines that one of these objects is no longer reachable, the object gets to finalize itself, releasing the native resource it wraps, and then, later, the GC will reclaim the object from the managed heap.

`System.Object`, the base class of everything, defines a protected and virtual method called `Finalize`. When the garbage collector determines that an object is garbage, it calls the object's `Finalize` method (if it is overridden). Microsoft's C# team felt that `Finalize` methods were a special kind of method requiring special syntax in the programming language (similar to how C# requires special syntax to define a constructor). So, in C#, you must define a `Finalize` method by placing a tilde symbol (~) in front of the class name, as shown in the following code sample.

```
internal sealed class SomeType {
   // This is the Finalize method
   ~SomeType() {
      // The code here is inside the Finalize method
   }
}
```

If you were to compile this code and examine the resulting assembly with ILDasm.exe, you'd see that the C# compiler did, in fact, emit a `protected override` method named `Finalize` into the module's metadata. If you examined the `Finalize` method's IL code, you'd also see that the code inside the method's body is emitted into a `try` block, and that a call to `base.Finalize` is emitted into a `finally` block.

Important If you're familiar with C++, you'll notice that the special syntax C# requires for defining a `Finalize` method looks just like the syntax you'd use to define a C++ destructor. In fact, the C# Programming Language Specification calls this method a *destructor*. However, a `Finalize` method doesn't work like a C++ destructor at all, and this has caused a great deal of confusion for developers migrating from one language to another.

The problem is that developers mistakenly believe that using the C# destructor syntax means that the type's objects will be deterministically destructed when they go out of lexical scope, just as they would be in C++. However, the CLR doesn't support deterministic destruction, preventing C# from providing this mechanism.

`Finalize` methods are called at the completion of a garbage collection on objects that the GC has determined to be garbage. This means that the memory for these objects cannot be reclaimed right away because the `Finalize` method might execute code that accesses a field. Because a finalizable object must survive the collection, it gets promoted to another generation, forcing the object to live much longer than it should. This is not ideal in terms of memory consumption and is why you should avoid finalization when possible. To make matters worse, when finalizable objects get promoted, any object referred to by its fields also get promoted because they must continue to live too. So, try to avoid defining finalizable objects with reference type fields.

Furthermore, be aware of the fact that you have no control over when the `Finalize` method will execute. `Finalize` methods run when a garbage collection occurs, which may happen when your application requests more memory. Also, the CLR doesn't make any guarantees as to the order in which `Finalize` methods are called. So, you should avoid writing a `Finalize` method that accesses other objects whose type defines a `Finalize` method; those other objects could have been finalized already. However, it is perfectly OK to access value type instances or reference type objects that do not define a `Finalize` method. You also need to be careful when calling static methods because these methods can internally access objects that have been finalized, causing the behavior of the static method to become unpredictable.

The CLR uses a special, high-priority dedicated thread to call `Finalize` methods to avoid some deadlock scenarios that could occur otherwise.[5] If a `Finalize` method blocks (for example, enters an infinite loop or waits for an object that is never signaled), this special thread can't call any more `Finalize` methods. This is a very bad situation because the application will never be able to reclaim the memory occupied by the finalizable objects—the application will leak memory as long as it runs.

[5] A future version of the CLR might use multiple finalizer threads to improve performance.

If a `Finalize` method throws an unhandled exception, then the process terminates; there is no way to catch this exception.

So, as you can see, there are a lot of caveats related to `Finalize` methods and they must be used with caution. Specifically, they are designed for releasing native resources. To simplify working with them, it is highly recommended that developers avoid overriding `Object`'s `Finalize` method; instead, use helper classes that Microsoft now provides in the Framework Class Library (FCL). The helper classes override `Finalize` and add some special CLR magic I'll talk about as we go on. You will then derive your own classes from the helper classes and inherit the CLR magic.

If you are creating a managed type that wraps a native resource, you should first derive a class from a special base class called `System.Runtime.InteropServices.SafeHandle`, which looks like the following (I've added comments in the methods to indicate what they do).

```
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {
   // This is the handle to the native resource
   protected IntPtr handle;

   protected SafeHandle(IntPtr invalidHandleValue, Boolean ownsHandle) {
      this.handle = invalidHandleValue;
      // If ownsHandle is true, then the native resource is closed when
      // this SafeHandle-derived object is collected
   }

   protected void SetHandle(IntPtr handle) {
      this.handle = handle;
   }


   // You can explicitly release the resource by calling Dispose
   // This is the IDisposable interface's Dispose method
   public void Dispose() { Dispose(true); }

   // The default Dispose implementation (shown here) is exactly what you want.
   // Overriding this method is strongly discouraged.
   protected virtual void Dispose(Boolean disposing) {
      // The default implementation ignores the disposing argument.
      // If resource already released, return
      // If ownsHandle is false, return
      // Set flag indicating that this resource has been released
      // Call virtual ReleaseHandle method
      // Call GC.SuppressFinalize(this) to prevent Finalize from being called
      // If ReleaseHandle returned true, return
      // If we get here, fire ReleaseHandleFailed Managed Debugging Assistant (MDA)
   }

   // The default Finalize implementation (shown here) is exactly what you want.
   // Overriding this method is very strongly discouraged.
   ~SafeHandle() { Dispose(false); }

   // A derived class overrides this method to implement the code that releases the resource
   protected abstract Boolean ReleaseHandle();

   public void SetHandleAsInvalid() {
      // Set flag indicating that this resource has been released
      // Call GC.SuppressFinalize(this) to prevent Finalize from being called
   }

   public Boolean IsClosed {
      get {
         // Returns flag indicating whether resource was released
      }
   }

   public abstract Boolean IsInvalid {
      // A derived class overrides this property.
      // The implementation should return true if the handle's value doesn't
      // represent a resource (this usually means that the handle is 0 or -1)
       get;
    }

   // These three methods have to do with security and reference counting;
   // I'll talk about them at the end of this section
   public void   DangerousAddRef(ref Boolean success) {...}
   public IntPtr DangerousGetHandle() {...}
   public void   DangerousRelease() {...}
}
```

The first thing to notice about the `SafeHandle` class is that it is derived from `CriticalFinalizer-Object`, which is defined in the `System.Runtime.ConstrainedExecution` namespace. The CLR treats this class and classes derived from it in a very special manner. In particular, the CLR endows this class with three cool features:

- The first time an object of any `CriticalFinalizerObject`-derived type is constructed, the CLR immediately JIT-compiles all of the `Finalize` methods in the inheritance hierarchy. Compiling these methods upon object construction guarantees that the native resource will be released when the object is determined to be garbage. Without this eager compiling of the `Finalize` method, it would be possible to allocate the native resource and use it, but not to get rid of it. Under low memory conditions, the CLR might not be able to find enough memory to compile the `Finalize` method, which would prevent it from executing, causing the native resource to leak. Or the resource might not be freed if the `Finalize` method contained code that referred to a type in another assembly, and the CLR failed to locate this other assembly.

- The CLR calls the `Finalize` method of `CriticalFinalizerObject`-derived types after calling the `Finalize` methods of non–`CriticalFinalizerObject`-derived types. This ensures that managed resource classes that have a `Finalize` method can access `Critical-FinalizerObject`-derived objects within their `Finalize` methods successfully. For example, the `FileStream` class's `Finalize` method can flush data from a memory buffer to an under-lying disk with confidence that the disk file has not been closed yet.

- The CLR calls the `Finalize` method of `CriticalFinalizerObject`-derived types if an AppDomain is rudely aborted by a host application (such as SQL Server or ASP.NET). This also is part of ensuring that the native resource is released even in a case in which a host applica-tion no longer trusts the managed code running inside of it.

The second thing to notice about SafeHandle is that the class is abstract; it is expected that another class will be derived from SafeHandle, and this class will provide a constructor that invokes the protected constructor, the abstract method ReleaseHandle, and the abstract IsInvalid property get accessor method.

Most native resources are manipulated with handles (32-bit values on a 32-bit system and 64-bit values on a 64-bit system). So the SafeHandle class defines a protected IntPtr field called handle. In Windows, most handles are invalid if they have a value of 0 or -1. The Microsoft.Win32.SafeHandles namespace contains another helper class called SafeHandleZeroOrMinusOneIsInvalid, which looks like this.

```
public abstract class SafeHandleZeroOrMinusOneIsInvalid : SafeHandle {
   protected SafeHandleZeroOrMinusOneIsInvalid(Boolean ownsHandle)
      : base(IntPtr.Zero, ownsHandle) {
   }

   public override Boolean IsInvalid {
      get {
         if (base.handle == IntPtr.Zero) return true;
         if (base.handle == (IntPtr) (-1)) return true;
         return false;
      }
   }
}
```

Again, you'll notice that the SafeHandleZeroOrMinusOneIsInvalid class is abstract, and therefore, another class must be derived from this one to override the protected constructor and the abstract method ReleaseHandle. The .NET Framework provides just a few public classes derived from SafeHandleZeroOrMinusOneIsInvalid, including SafeFileHandle, SafeRegistryHandle, SafeWaitHandle, and SafeMemoryMappedViewHandle. Here is what the SafeFileHandle class looks like.

```
public sealed class SafeFileHandle : SafeHandleZeroOrMinusOneIsInvalid {
   public SafeFileHandle(IntPtr preexistingHandle, Boolean ownsHandle)
      : base(ownsHandle) {
      base.SetHandle(preexistingHandle);
   }

   protected override Boolean ReleaseHandle() {
      // Tell Windows that we want the native resource closed.
      return Win32Native.CloseHandle(base.handle);
   }
}
```

The SafeWaitHandle class is implemented similarly to the SafeFileHandle class just shown. The only reason why there are different classes with similar implementations is to achieve type safety; the compiler won't let you use a file handle as an argument to a method that expects a wait handle, and vice versa. The SafeRegistryHandle class's ReleaseHandle method calls the Win32 RegCloseKey function.

It would be nice if the .NET Framework included additional classes that wrap various native resources. For example, one could imagine classes such as SafeProcessHandle, SafeThreadHandle, SafeTokenHandle, SafeLibraryHandle (its ReleaseHandle method would call the Win32 FreeLibrary function), SafeLocalAllocHandle (its ReleaseHandle method would call the Win32 LocalFree function), and so on.

All of the classes just listed (and more) actually do ship with the Framework Class Library (FCL). However, these classes are not publicly exposed; they are all internal to the assemblies that define them. Microsoft didn't expose these classes publicly because they didn't want to document them and do full testing of them. However, if you need any of these classes for your own work, I'd recommend that you use a tool such as ILDasm.exe or some IL decompiler tool to extract the code for these classes and integrate that code into your own project's source code. All of these classes are trivial to implement, and writing them yourself from scratch would also be quite easy.

The SafeHandle-derived classes are extremely useful because they ensure that the native resource is freed when a GC occurs. In addition to what we've already discussed, SafeHandle offers two more capabilities. First, the CLR gives SafeHandle-derived types special treatment when used in scenarios in which you are interoperating with native code. For example, let's examine the following code.

```
using System;
using System.Runtime.InteropServices;
using Microsoft.Win32.SafeHandles;

internal static class SomeType {
   [DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]
   // This prototype is not robust
   private static extern IntPtr CreateEventBad(
      IntPtr pSecurityAttributes, Boolean manualReset, Boolean initialState, String name);

   // This prototype is robust
   [DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]
   private static extern SafeWaitHandle CreateEventGood(
      IntPtr pSecurityAttributes, Boolean manualReset, Boolean initialState, String name);

   public static void SomeMethod() {
      IntPtr         handle = CreateEventBad(IntPtr.Zero, false, false, null);
      SafeWaitHandle swh    = CreateEventGood(IntPtr.Zero, false, false, null);
   }
}
```

You'll notice that the CreateEventBad method is prototyped as returning an IntPtr, which will return the handle back to managed code; however, interoperating with native code this way is not robust. You see, after CreateEventBad is called (which creates the native event resource), it is possible that a ThreadAbortException could be thrown prior to the handle being assigned to the handle variable. In the rare cases when this would happen, the managed code would leak the native resource. The only way to get the event closed is to terminate the whole process.

The SafeHandle class fixes this potential resource leak. Notice that the CreateEventGood method is prototyped as returning a SafeWaitHandle (instead of an IntPtr). When CreateEvent-Good is called, the CLR calls the Win32 CreateEvent function. As the CreateEvent function returns to managed code, the CLR knows that SafeWaitHandle is derived from SafeHandle, causing the CLR to automatically construct an instance of the SafeWaitHandle class on the managed heap, passing in the handle value returned from CreateEvent. The constructing of the SafeWaitHandle object and the assignment of the handle happen in native code now, which cannot be interrupted by a ThreadAbortException. Now, it is impossible for managed code to leak this native resource. Eventually, the SafeWaitHandle object will be garbage collected and its Finalize method will be called, ensuring that the resource is released.

One last feature of SafeHandle-derived classes is that they prevent someone from trying to exploit a potential security hole. The problem is that one thread could be trying to use a native resource while another thread tries to free the resource. This could manifest itself as a handle-recycling exploit. The SafeHandle class prevents this security vulnerability by using reference counting. Internally, the SafeHandle class defines a private field that maintains a count. When a SafeHandle-derived object is set to a valid handle, the count is set to 1. Whenever a SafeHandle-derived object is passed as an argument to a native method, the CLR knows to automatically increment the counter. Likewise, when the native method returns to managed code, the CLR knows to decrement the counter. For example, you would prototype the Win32 SetEvent function as follows.

```
[DllImport("Kernel32", ExactSpelling=true)]
private static extern Boolean SetEvent(SafeWaitHandle swh);
```

Now when you call this method passing in a reference to a SafeWaitHandle object, the CLR will increment the counter just before the call and decrement the counter just after the call. Of course, the manipulation of the counter is performed in a thread-safe fashion. How does this improve security? Well, if another thread tries to release the native resource wrapped by the SafeHandle object, the CLR knows that it cannot actually release it because the resource is being used by a native function. When the native function returns, the counter is decremented to 0, and the resource will be released.

If you are writing or calling code to manipulate a handle as an IntPtr, you can access it out of a SafeHandle object, but you should manipulate the reference counting explicitly. You accomplish this via SafeHandle's DangerousAddRef and DangerousRelease methods. You gain access to the raw handle via the DangerousGetHandle method.

I would be remiss if I didn't mention that the System.Runtime.InteropServices namespace also defines a CriticalHandle class. This class works exactly as the SafeHandle class in all ways except that it does not offer the reference-counting feature. The CriticalHandle class and the classes derived from it sacrifice security for better performance when you use it (because counters don't get manipulated). As does SafeHandle, the CriticalHandle class has two types derived from it: CriticalHandleMinusOneIsInvalid and CriticalHandleZeroOrMinusOneIsInvalid. Because Microsoft favors a more secure system over a faster system, the class library includes no types derived from either of these two classes. For your own work, I would recommend that you use Critical-Handle-derived types only if performance is an issue. If you can justify reducing security, you can switch to a CriticalHandle-derived type.

Using a Type That Wraps a Native Resource

Now that you know how to define a `SafeHandle`-derived class that wraps a native resource, let's take a look at how a developer uses it. Let's start by talking about the common `System.IO.FileStream` class. The `FileStream` class offers the ability to open a file, read bytes from the file, write bytes to the file, and close the file. When a `FileStream` object is constructed, the Win32 `CreateFile` function is called, the returned handle is saved in a `SafeFileHandle` object, and a reference to this object is maintained via a private field in the `FileStream` object. The `FileStream` class also offers several additional properties (such as `Length`, `Position`, `CanRead`) and methods (such as `Read`, `Write`, `Flush`).

Let's say that you want to write some code that creates a temporary file, writes some bytes to the file, and then deletes the file. You might start writing the code like this.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Write the bytes to the temporary file.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Delete the temporary file.
      File.Delete("Temp.dat");  // Throws an IOException
   }
}
```

Unfortunately, if you build and run this code, it might work, but most likely it won't. The problem is that the call to `File`'s static `Delete` method requests that Windows delete a file while it is still open. So `Delete` throws a `System.IO.IOException` exception with the following string message: The `process cannot access the file "Temp.dat" because it is being used by another process.`

Be aware that in some cases, the file might actually be deleted! If another thread somehow caused a garbage collection to start after the call to `Write` and before the call to `Delete`, the `FileStream`'s `SafeFileHandle` field would have its `Finalize` method called, which would close the file and allow `Delete` to work. The likelihood of this situation is extremely rare, however, and therefore the previous code will fail more than 99 percent of the time.

Classes that allow the consumer to control the lifetime of native resources it wraps implement the IDisposable interface, which looks like this.

```
public interface IDisposable {
   void Dispose();
}
```

Important If a class defines a field in which the field's type implements the dispose pattern, the class itself should also implement the dispose pattern. The Dispose method should dispose of the object referred to by the field. This allows someone using the class to call Dispose on it, which in turn releases the resources used by the object itself.

Fortunately, the FileStream class implements the IDisposable interface and its implementation internally calls Dispose on the FileStream object's private SafeFileHandle field. Now, we can modify our code to explicitly close the file when we want to as opposed to waiting for some GC to happen in the future. Here's the corrected source code.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Write the bytes to the temporary file.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Explicitly close the file when finished writing to it.
      fs.Dispose();

      // Delete the temporary file.
      File.Delete("Temp.dat");  // This always works now.
   }
}
```

Now, when File's Delete method is called, Windows sees that the file isn't open and successfully deletes it.

Keep in mind that calling `Dispose` is not required to guarantee native resource cleanup. Native resource cleanup will always happen eventually; calling `Dispose` lets you control when that cleanup happens. Also, calling `Dispose` does not delete the managed object from the managed heap. The only way to reclaim memory in the managed heap is for a garbage collection to kick in. This means you can still call methods on the managed object even after you dispose of any native resources it may have been using.

The following code calls the `Write` method after the file is closed, attempting to write more bytes to the file. Obviously, the bytes can't be written, and when the code executes, the second call to the `Write` method throws a `System.ObjectDisposedException` exception with the following string message: `Cannot access a closed file.`

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Write the bytes to the temporary file.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Explicitly close the file when finished writing to it.
      fs.Dispose();

      // Try to write to the file after closing it.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);  // Throws ObjectDisposedException

      // Delete the temporary file.
      File.Delete("Temp.dat");
   }
}
```

Note that no memory corruption occurs here because the memory for the `FileStream` object still exists in the managed heap; it's just that the object can't successfully execute its methods after it is explicitly disposed.

Important When defining your own type that implements the `IDisposable` interface, be sure to write code in all of your methods and properties to throw a `System.ObjectDisposedException` if the object has been explicitly cleaned up. A `Dispose` method should never throw an exception; if it's called multiple times, it should just return.

Important In general, I strongly discourage explicitly calling `Dispose` in your code. The reason is that the CLR's garbage collector is well written, and you should let it do its job. The garbage collector knows when an object is no longer accessible from application code, and only then will it collect the object.[6] When application code calls `Dispose`, it is effectively saying that it knows when the application no longer has a need for the object. For many applications, it is impossible to know for sure when an object is no longer required.

For example, if you have code that constructs a new object, and you then pass a reference to this object to another method, the other method could save a reference to the object in some internal field variable (a root). There is no way for the calling method to know that this has happened. Sure, the calling method can call `Dispose`, but later, some other code might try to access the object, causing an `ObjectDisposedException` to be thrown. I recommend that you call `Dispose` only at places in your code where you know you must clean up the resource (as in the case of attempting to delete an open file).

Along the same lines, it is possible to have multiple threads call `Dispose` on a single object simultaneously. However, the design guidelines state that `Dispose` does not have to be thread-safe. The reason is because code should be calling `Dispose` only if the code knows for a fact that no other thread is using the object.

The previous code examples show how to explicitly call a type's `Dispose` method. If you decide to call `Dispose` explicitly, I highly recommend that you place the call in an exception-handling `finally` block. This way, the cleanup code is guaranteed to execute. So it would be better to write the previous code example as follows.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);
      try {
         // Write the bytes to the temporary file.
         fs.Write(bytesToWrite, 0, bytesToWrite.Length);
      }
      finally {
         // Explicitly close the file when finished writing to it.
```

[6] There are many nice features about a garbage collected system: no memory leaks, no memory corruption, no address space fragmentation, and a reduced working set. And now, a new one: synchronization. That's right, you can use the GC as a thread synchronization mechanism. Question: How can you know when all threads are done using an object? Answer: the GC finalizes the object. There is nothing wrong with taking advantage of all the GC features as you architect your software.

```
         if (fs != null)  fs.Dispose();
      }

      // Delete the temporary file.
      File.Delete("Temp.dat");
   }
}
```

Adding the exception-handling code is the right thing to do, and you must have the diligence to do it. Fortunately, the C# language provides a `using` statement, which offers a simplified syntax that produces code identical to the code just shown. Here's how the preceding code would be rewritten using C#'s `using` statement.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Create the bytes to write to the temporary file.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Create the temporary file.
      using (FileStream fs = new FileStream("Temp.dat", FileMode.Create)) {
         // Write the bytes to the temporary file.
         fs.Write(bytesToWrite, 0, bytesToWrite.Length);
      }

      // Delete the temporary file.
      File.Delete("Temp.dat");
   }
}
```

In the `using` statement, you initialize an object and save its reference in a variable. Then you access the variable via code contained inside `using`'s braces. When you compile this code, the compiler automatically emits the `try` and `finally` blocks. Inside the `finally` block, the compiler emits code to cast the object to an `IDisposable` and calls the `Dispose` method. Obviously, the compiler allows the `using` statement to be used only with types that implement the `IDisposable` interface.

Note C#'s `using` statement supports the capability to initialize multiple variables as long as the variables are all of the same type. It also supports the capability to use just an already initialized variable. For more information about this topic, refer to the "Using Statements" topic in the C# Programmer's Reference.

An Interesting Dependency Issue

The System.IO.FileStream type allows the user to open a file for reading and writing. To improve performance, the type's implementation makes use of a memory buffer. Only when the buffer fills does the type flush the contents of the buffer to the file. A FileStream supports the writing of bytes only. If you want to write characters and strings, you can use a System.IO.StreamWriter, as is demonstrated in the following code.

```
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
sw.Write("Hi there");

// The following call to Dispose is what you should do.
sw.Dispose();
// NOTE: StreamWriter.Dispose closes the FileStream;
// the FileStream doesn't have to be explicitly closed.
```

Notice that the StreamWriter's constructor takes a reference to a Stream object as a parameter, allowing a reference to a FileStream object to be passed as an argument. Internally, the StreamWriter object saves the Stream's reference. When you write to a StreamWriter object, it internally buffers the data in its own memory buffer. When the buffer is full, the StreamWriter object writes the data to the Stream.

When you're finished writing data via the StreamWriter object, you should call Dispose. (Because the StreamWriter type implements the IDisposable interface, you can also use it with C#'s using statement.) This causes the StreamWriter object to flush its data to the Stream object and close the Stream object.[7]

Note You don't have to explicitly call Dispose on the FileStream object because the StreamWriter calls it for you. However, if you do call Dispose explicitly, the FileStream will see that the object has already been cleaned up—the method does nothing and just returns.

What do you think would happen if there were no code to explicitly call Dispose? Well, at some point, the garbage collector would correctly detect that the objects were garbage and finalize them. But the garbage collector doesn't guarantee the order in which objects are finalized. So if the FileStream object were finalized first, it would close the file. Then when the StreamWriter object was finalized, it would attempt to write data to the closed file, throwing an exception. If, on the other hand, the StreamWriter object were finalized first, the data would be safely written to the file.

[7] You can override this behavior by calling StreamWriter's constructor that accepts a Boolean leaveOpen parameter.

How was Microsoft to solve this problem? Making the garbage collector finalize objects in a specific order would have been impossible because objects could contain references to each other, and there would be no way for the garbage collector to correctly guess the order in which to finalize these objects. Here is Microsoft's solution: the `StreamWriter` type does not support finalization, and therefore it never flushes data in its buffer to the underlying `FileStream` object. This means that if you forget to explicitly call `Dispose` on the `StreamWriter` object, data is guaranteed to be lost. Microsoft expects developers to see this consistent loss of data and fix the code by inserting an explicit call to `Dispose`.

Note The .NET Framework offers a feature called Managed Debugging Assistants (MDAs). When an MDA is enabled, the .NET Framework looks for certain common programmer errors and fires a corresponding MDA. In the debugger, it looks like an exception has been thrown. There is an MDA available to detect when a `StreamWriter` object is garbage collected without previously having been explicitly disposed. To enable this MDA in Microsoft Visual Studio, open your project and select the `Debug.Exceptions` menu item. In the Exceptions dialog box, expand the Managed Debugging Assistants node and scroll to the bottom. There you will see the `StreamWriterBufferredDataLost` MDA. Select the Thrown check box to have the Visual Studio debugger stop whenever a `StreamWriter` object's data is lost.

Other GC Features for Use with Native Resources

Sometimes, a native resource consumes a lot of memory, but the managed object wrapping that resource occupies very little memory. The quintessential example of this is the bitmap. A bitmap can occupy several megabytes of native memory, but the managed object is tiny because it contains only an `HBITMAP` (a 4-byte or 8-byte value). From the CLR's perspective, a process could allocate hundreds of bitmaps (using little managed memory) before performing a collection. But if the process is manipulating many bitmaps, the process's memory consumption will grow at a phenomenal rate. To fix this situation, the GC class offers the following two static methods.

```
public static void AddMemoryPressure(Int64 bytesAllocated);
public static void RemoveMemoryPressure(Int64 bytesAllocated);
```

A class that wraps a potentially large native resource should use these methods to give the garbage collector a hint as to how much memory is really being consumed. Internally, the garbage collector monitors this pressure, and when it gets high, a garbage collection is forced.

There are some native resources that are fixed in number. For example, Windows formerly had a restriction that it could create only five device contexts. There had also been a restriction on the number of files that an application could open. Again, from the CLR's perspective, a process could allocate hundreds of objects (that use little memory) before performing a collection. But if the number of these native resources is limited, attempting to use more than are available will typically result in exceptions being thrown.

To fix this situation, the `System.Runtime.InteropServices` namespace offers the `HandleCollector` class.

```
public sealed class HandleCollector {
   public HandleCollector(String name, Int32 initialThreshold);
   public HandleCollector(String name, Int32 initialThreshold,  Int32 maximumThreshold);
   public void Add();
   public void Remove();

   public Int32 Count { get; }
   public Int32 InitialThreshold { get; }
   public Int32 MaximumThreshold { get; }
   public String Name { get; }
}
```

A class that wraps a native resource that has a limited quantity available should use an instance of this class to give the garbage collector a hint as to how many instances of the resource are really being consumed. Internally, this class object monitors the count, and when it gets high, a garbage collection is forced.

Note Internally, the `GC.AddMemoryPressure` and `HandleCollector.Add` methods call `GC.Collect`, forcing a garbage collection to start prior to generation 0 reaching its budget. Normally, forcing a garbage collection to start is strongly discouraged, because it usually has an adverse effect on your application's performance. However, classes that call these methods are doing so in an effort to keep limited native resources available for the application. If the native resources run out, the application will fail. For most applications, it is better to work with reduced performance than to not be working at all.

Here is some code that demonstrates the use and effect of the memory pressure methods and the `HandleCollector` class.

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

public static class Program {
   public static void Main() {
      MemoryPressureDemo(0);                 // 0    causes infrequent GCs
      MemoryPressureDemo(10 * 1024 * 1024);  // 10MB causes frequent GCs

      HandleCollectorDemo();
}

   private static void MemoryPressureDemo(Int32 size) {
      Console.WriteLine();
      Console.WriteLine("MemoryPressureDemo, size={0}", size);
      // Create a bunch of objects specifying their logical size
      for (Int32 count = 0; count < 15; count++) {
         new BigNativeResource(size);
      }
      // For demo purposes, force everything to be cleaned-up
      GC.Collect();
   }

   private sealed class BigNativeResource {
      private readonly Int32 m_size;

      public BigNativeResource(Int32 size) {
         m_size = size;
         // Make the GC think the object is physically bigger
         if (m_size > 0)  GC.AddMemoryPressure(m_size);
         Console.WriteLine("BigNativeResource create.");
      }

      ~BigNativeResource() {
         // Make the GC think the object released more memory
         if (m_size > 0)  GC.RemoveMemoryPressure(m_size);
         Console.WriteLine("BigNativeResource destroy.");
      }
   }


   private static void HandleCollectorDemo() {
      Console.WriteLine();
      Console.WriteLine("HandleCollectorDemo");
      for (Int32 count = 0; count < 10; count++) new LimitedResource();

      // For demo purposes, force everything to be cleaned-up
      GC.Collect();
   }

   private sealed class LimitedResource {
      // Create a HandleCollector telling it that collections should
      // occur when two or more of these objects exist in the heap
      private static readonly HandleCollector s_hc = new HandleCollector("LimitedResource", 2);

      public LimitedResource() {
         // Tell the HandleCollector a LimitedResource has been added to the heap
         s_hc.Add();
         Console.WriteLine("LimitedResource create.  Count={0}", s_hc.Count);
      }
      ~LimitedResource() {
         // Tell the HandleCollector a LimitedResource has been removed from the heap
         s_hc.Remove();
         Console.WriteLine("LimitedResource destroy. Count={0}", s_hc.Count);
      }
   }
}
```

If you compile and run the preceding code, your output will be similar to the following output.

```
MemoryPressureDemo, size=0
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.

MemoryPressureDemo, size=10485760
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.

HandleCollectorDemo
LimitedResource create.  Count=1
LimitedResource create.  Count=2
LimitedResource create.  Count=3
LimitedResource destroy. Count=3
LimitedResource destroy. Count=2
LimitedResource destroy. Count=1
LimitedResource create.  Count=1
LimitedResource create.  Count=2
LimitedResource create.  Count=3
LimitedResource destroy. Count=2
LimitedResource create.  Count=3
LimitedResource destroy. Count=3
LimitedResource destroy. Count=2
LimitedResource destroy. Count=1
LimitedResource create.  Count=1
LimitedResource create.  Count=2
LimitedResource create.  Count=3
LimitedResource destroy. Count=2
LimitedResource destroy. Count=1
LimitedResource destroy. Count=0
```

Finalization Internals

On the surface, finalization seems pretty straightforward: you create an object and its `Finalize` method is called when it is collected. But after you dig in, finalization is more complicated than this.

When an application creates a new object, the `new` operator allocates the memory from the heap. If the object's type defines a `Finalize` method, a pointer to the object is placed on the *finalization list* just before the type's instance constructor is called. The finalization list is an internal data structure controlled by the garbage collector. Each entry in the list points to an object that should have its `Finalize` method called before the object's memory can be reclaimed.

Figure 21-13 shows a heap containing several objects. Some of these objects are reachable from application roots, and some are not. When objects C, E, F, I, and J were created, the system detected that these objects' types defined a `Finalize` method and so added references to these objects to the finalization list.

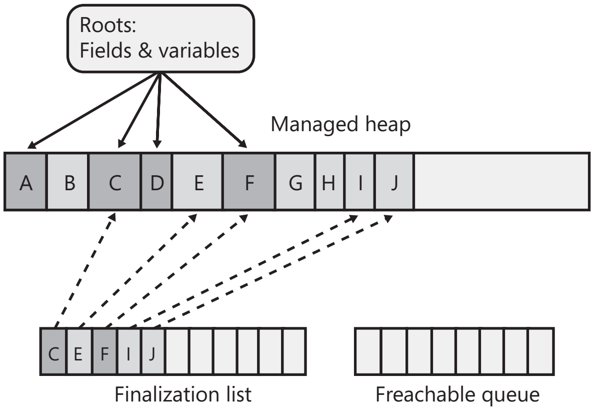


FIGURE 21-13 The managed heap showing pointers in its finalization list.

Note Even though `System.Object` defines a `Finalize` method, the CLR knows to ignore it; that is, when constructing an instance of a type, if the type's `Finalize` method is the one inherited from `System.Object`, the object isn't considered finalizable. One of the derived types must override `Object`'s `Finalize` method.

When a garbage collection occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalization list looking for references to these objects. When a reference is found, the reference is removed from the finalization list and appended to the *freachable queue*. The freachable queue (pronounced "F-reachable") is another of the garbage collector's internal data structures. Each reference in the freachable queue identifies an object that is ready to have its `Finalize` method called. After the collection, the managed heap looks like Figure 21-14.

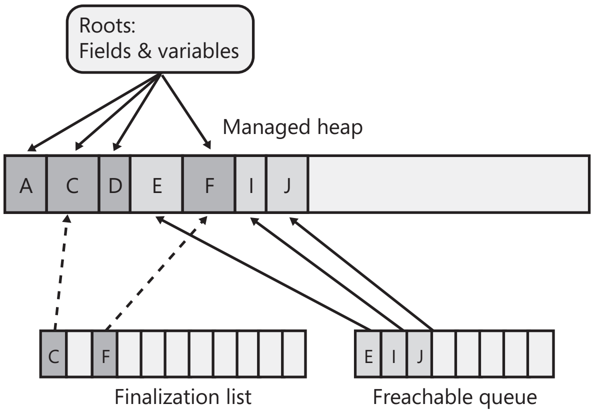


FIGURE 21-14 The managed heap showing pointers that moved from the finalization list to the freachable queue.

In this figure, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects didn't have a `Finalize` method. However, the memory occupied by objects E, I, and J couldn't be reclaimed because their `Finalize` methods haven't been called yet.

A special high-priority CLR thread is dedicated to calling `Finalize` methods. A dedicated thread is used to avoid potential thread synchronization situations that could arise if one of the application's normal-priority threads were used instead. When the freachable queue is empty (the usual case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and then calls each object's `Finalize` method. Because of the way this thread works, you shouldn't execute any code in a `Finalize` method that makes any assumptions about the thread that's executing the code. For example, avoid accessing thread-local storage in the `Finalize` method.

In the future, the CLR may use multiple finalizer threads. So you should avoid writing any code that assumes that `Finalize` methods will be called serially. With just one finalizer thread, there could be performance and scalability issues in the scenario in which you have multiple CPUs allocating finalizable objects but only one thread executing `Finalize` methods—the one thread might not be able to keep up with the allocations.

The interaction between the finalization list and the freachable queue is fascinating. First, I'll tell you how the freachable queue got its name. Well, the "f" is obvious and stands for *finalization*; every entry in the freachable queue is a reference to an object in the managed heap that should have its `Finalize` method called. But the *reachable* part of the name means that the objects are reachable. To put it another way, the freachable queue is considered a root, just as static fields are roots. So a reference in the freachable queue keeps the object it refers to reachable and is *not* garbage.

In short, when an object isn't reachable, the garbage collector considers the object to be garbage. Then when the garbage collector moves an object's reference from the finalization list to the freachable queue, the object is no longer considered garbage and its memory can't be reclaimed. When an object is garbage and then not garbage, we say that the object has been *resurrected*.

As freachable objects are marked, objects referred to by their reference type fields are also marked recursively; all these objects must get resurrected in order to survive the collection. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been resurrected. The garbage collector compacts the reclaimable memory, which promotes the resurrected object to an older generation (not ideal). And now, the special finalization thread empties the freachable queue, executing each object's `Finalize` method.

The next time the garbage collector is invoked on the older generation, it will see that the finalized objects are truly garbage because the application's roots don't point to it and the freachable queue no longer points to it either. The memory for the object is simply reclaimed. The important point to get from all of this is that two garbage collections are required to reclaim memory used by objects that require finalization. In reality, more than two collections will be necessary because the objects get promoted to another generation. Figure 21-15 shows what the managed heap looks like after the second garbage collection.

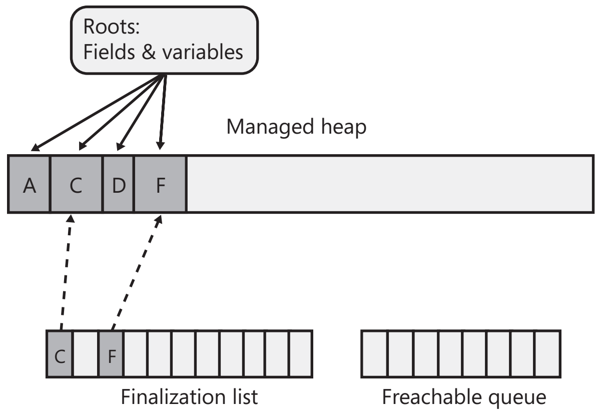


FIGURE 21-15 Status of the managed heap after second garbage collection.

Monitoring and Controlling the Lifetime of Objects Manually

The CLR provides each AppDomain with a GC handle table. This table allows an application to monitor the lifetime of an object or manually control the lifetime of an object. When an AppDomain is created, the table is empty. Each entry on the table consists of a reference to an object on the managed heap and a flag indicating how you want to monitor or control the object. An application adds and removes entries from the table via the `System.Runtime.InteropServices.GCHandle` type, as follows.

```
// This type is defined in the System.Runtime.InteropServices namespace
public struct GCHandle {
   // Static methods that create an entry in the table
   public static GCHandle Alloc(object value);
   public static GCHandle Alloc(object value, GCHandleType type);

   // Static methods that convert a GCHandle to an IntPtr
   public static explicit operator IntPtr(GCHandle value);
   public static IntPtr ToIntPtr(GCHandle value);

   // Static methods that convert an IntPtr to a GCHandle
   public static explicit operator GCHandle(IntPtr value);
   public static GCHandle FromIntPtr(IntPtr value);

   // Static methods that compare two GCHandles
   public static Boolean operator ==(GCHandle a, GCHandle b);
   public static Boolean operator !=(GCHandle a, GCHandle b);

   // Instance method to free the entry in the table (index is set to 0)
   public void Free();

   // Instance property to get/set the entry's object reference
   public object Target { get; set; }
   // Instance property that returns true if index is not 0
   public Boolean IsAllocated { get; }

   // For a pinned entry, this returns the address of the object
   public IntPtr AddrOfPinnedObject();

   public override Int32 GetHashCode();
   public override Boolean Equals(object o);
}
```

Basically, to control or monitor an object's lifetime, you call `GCHandle`'s static `Alloc` method, passing a reference to the object that you want to monitor/control, and a `GCHandleType`, which is a flag indicating how you want to monitor/control the object. The `GCHandleType` type is an enumerated type defined as follows.

```
public enum GCHandleType {
   Weak = 0,                  // Used for monitoring an object's existence
   WeakTrackResurrection = 1, // Used for monitoring an object's existence
   Normal = 2,                // Used for controlling an object's lifetime
   Pinned = 3                 // Used for controlling an object's lifetime
}
```

Now, here's what each flag means:

- **Weak** This flag allows you to *monitor* the lifetime of an object. Specifically, you can detect when the garbage collector has determined this object to be unreachable from application code. Note that the object's `Finalize` method may or may not have executed yet and therefore, the object may still be in memory.

- **WeakTrackResurrection** This flag allows you to *monitor* the lifetime of an object. Specifically, you can detect when the garbage collector has determined that this object is unreachable from application code. Note that the object's `Finalize` method (if it exists) has definitely executed, and the object's memory has been reclaimed.

- **Normal** This flag allows you to *control* the lifetime of an object. Specifically, you are telling the garbage collector that this object must remain in memory even though there may be no roots in the application that refer to this object. When a garbage collection runs, the memory for this object can be compacted (moved). The `Alloc` method that doesn't take a `GCHandleType` flag assumes that `GCHandleType.Normal` is specified.

- **Pinned** This flag allows you to *control* the lifetime of an object. Specifically, you are telling the garbage collector that this object must remain in memory even though there might be no roots in the application that refer to this object. When a garbage collection runs, the memory for this object cannot be compacted. This is typically useful when you want to hand the address of the memory out to native code. The native code can write to this memory in the managed heap knowing that a GC will not move the object.

When you call `GCHandle`'s static `Alloc` method, it scans the AppDomain's GC handle table, looking for an available entry where the reference of the object you passed to `Alloc` is stored, and a flag is set to whatever you passed for the `GCHandleType` argument. Then, `Alloc` returns a `GCHandle` instance back to you. A GCHandle is a lightweight value type that contains a single instance field, an IntPtr, which refers to the index of the entry in the table. When you want to free this entry in the GC handle table, you take the GCHandle instance and call the Free method (which also invalidates the GCHandle instance by setting its IntPtr field to zero).

Here's how the garbage collector uses the GC handle table. When a garbage collection occurs:

1. The garbage collector marks all of the reachable objects (as described at the beginning of this chapter). Then, the garbage collector scans the GC handle table; all Normal or Pinned objects are considered roots, and these objects are marked as well (including any objects that these objects refer to via their fields).

2. The garbage collector scans the GC handle table looking for all of the Weak entries. If a Weak entry refers to an object that isn't marked, the reference identifies an unreachable object (garbage), and the entry has its reference value changed to null.

3. The garbage collector scans the finalization list. If a reference in the list refers to an unmarked object, the reference identifies an unreachable object, and the reference is moved from the finalization list to the freachable queue. At this point, the object is marked because the object is now considered reachable.

4. The garbage collector scans the GC handle table looking for all of the WeakTrackResurrection entries. If a WeakTrackResurrection entry refers to an object that isn't marked (which now is an object referenced by an entry in the freachable queue), the reference identifies an unreachable object (garbage), and the entry has its reference value changed to null.

5. The garbage collector compacts the memory, squeezing out the holes left by the unreachable objects. Pinned objects are not compacted (moved); the garbage collector will move other objects around them.

Now that you have an understanding of the mechanism, let's take a look at when you'd use them. The easiest flags to understand are the Normal and Pinned flags, so let's start with these two. Both of these flags are typically used when interoperating with native code.

The Normal flag is used when you need to hand a pointer to a managed object to native code because, at some point in the future, the native code is going to call back into managed code, passing it the pointer. You can't actually pass a pointer to a managed object out to native code, because if a garbage collection occurs, the object could move in memory, invalidating the pointer. So to work around this, you would call GCHandle's Alloc method, passing in a reference to the object and the Normal flag. Then you'd cast the returned GCHandle instance to an IntPtr and pass the IntPtr into the native code. When the native code calls back into managed code, the managed code would cast the passed IntPtr back to a GCHandle and then query the Target property to get the reference (or current address) of the managed object. When the native code no longer needs the reference, you'd call GCHandle's Free method, which allows a future garbage collection to free the object (assuming no other root exists to this object).

Notice that in this scenario, the native code is not actually using the managed object itself; the native code wants a way just to reference the object. In some scenarios, the native code needs to actually use the managed object. In these scenarios, the managed object must be pinned. Pinning prevents the garbage collector from moving/compacting the object. A common example is when you want to pass a managed `String` object to a Win32 function. In this case, the `String` object must be pinned because you can't pass the reference of a managed object to native code and then have the garbage collector move the object in memory. If the `String` object were moved, the native code would either be reading or writing to memory that no longer contained the `String` object's characters—this will surely cause the application to run unpredictably.

When you use the CLR's P/Invoke mechanism to call a method, the CLR pins the arguments for you automatically and unpins them when the native method returns. So, in most cases, you never have to use the `GCHandle` type to explicitly pin any managed objects yourself. You do have to use the `GCHandle` type explicitly when you need to pass the pointer to a managed object to native code; then the native function returns, but native code might still need to use the object later. The most common example of this is when performing asynchronous I/O operations.

Let's say that you allocate a byte array that should be filled as data comes in from a socket. Then, you would call `GCHandle`'s `Alloc` method, passing in a reference to the array object and the `Pinned` flag. Then, using the returned `GCHandle` instance, you call the `AddrOfPinnedObject` method. This returns an `IntPtr` that is the actual address of the pinned object in the managed heap; you'd then pass this address into the native function, which will return back to managed code immediately. While the data is coming from the socket, this byte array buffer should not move in memory; preventing this buffer from moving is accomplished by using the `Pinned` flag. When the asynchronous I/O operation has completed, you'd call `GCHandle`'s `Free` method, which will allow a future garbage collection to move the buffer. Your managed code should still have a reference to the buffer so that you can access the data, and this reference will prevent a garbage collection from freeing the buffer from memory completely.

It is also worth mentioning that C# offers a `fixed` statement that effectively pins an object over a block of code. Here is some code that demonstrates its use.

```
unsafe public static void Go() {
   // Allocate a bunch of objects that immediately become garbage
   for (Int32 x = 0; x < 10000; x++) new Object();

   IntPtr  originalMemoryAddress;
   Byte[] bytes = new Byte[1000];   // Allocate this array after the garbage objects

   // Get the address in memory of the Byte[]
   fixed (Byte* pbytes = bytes) { originalMemoryAddress = (IntPtr) pbytes; }

   // Force a collection; the garbage objects will go away & the Byte[] might be compacted
   GC.Collect();

   // Get the address in memory of the Byte[] now & compare it to the first address
   fixed (Byte* pbytes = bytes) {
      Console.WriteLine("The Byte[] did{0} move during the GC",
         (originalMemoryAddress == (IntPtr) pbytes) ? " not" : null);
   }
}
```

Using C#'s fixed statement is more efficient that allocating a pinned GC handle. What happens is that the C# compiler emits a special "pinned" flag on the pbytes local variable. During a garbage collection, the GC examines the contents of this root, and if the root is not null, it knows not to move the object referred to by the variable during the compaction phase. The C# compiler emits IL to initialize the pbytes local variable to the address of the object at the start of a fixed block, and the compiler emits an IL instruction to set the pbytes local variable back to null at the end of the fixed block so that the variable doesn't refer to any object, allowing the object to move when the next garbage collection occurs.

Now, let's talk about the next two flags, Weak and WeakTrackResurrection. These two flags can be used in scenarios when interoperating with native code, but they can also be used in scenarios that use only managed code. The Weak flag lets you know when an object has been determined to be garbage but the object's memory is not guaranteed to be reclaimed yet. The WeakTrackResurrection flag lets you know when an object's memory has been reclaimed. Of the two flags, the Weak flag is much more commonly used than the WeakTrackResurrection flag. In fact, I've never seen anyone use the WeakTrackResurrection flag in a real application.

Let's say that Object-A periodically calls a method on Object-B. However, the fact that Object-A has a reference to Object-B forbids Object-B from being garbage collected, and in some rare scenarios, this may not be desired; instead, we might want Object-A to call Object-B's method if Object-B is still alive in the managed heap. To accomplish this scenario, Object-A would call GCHandle's Alloc method, passing in the reference to Object-B and the Weak flag. Object-A would now just save the returned GCHandle instance instead of the reference to Object-B.

At this point, Object-B can be garbage collected if no other roots are keeping it alive. When Object-A wants to call Object-B's method, it would query GCHandle's read-only Target property. If this property returns a non-null value, then Object-B is still alive. Object-A's code would then cast the returned reference to Object-B's type and call the method. If the Target property returns null, then Object-B has been collected (but not necessarily finalized) and Object-A would not attempt to call the method. At this point, Object-A's code would probably also call GCHandle's Free method to relinquish the GCHandle instance.

Because working with the GCHandle type can be a bit cumbersome and because it requires elevated security to keep or pin an object in memory, the System namespace includes a WeakReference<T> class to help you.

```
public sealed class WeakReference<T> : ISerializable where T : class {
   public WeakReference(T target);
   public WeakReference(T target, Boolean trackResurrection);
   public void SetTarget(T target);
   public Boolean TryGetTarget(out T target);
}
```

This class is really just an object-oriented wrapper around a GCHandle instance: logically, its constructor calls GCHandle's Alloc, its TryGetTarget method queries GCHandle's Target property, its SetTarget method sets GCHandle's Target property, and its Finalize method (not shown in the preceding code, because it's protected) calls GCHandle's Free method. In addition, no special permissions are required for code to use the WeakReference<T> class because the class supports only weak references; it doesn't support the behavior provided by `GCHandle` instances allocated with a `GCHandleType` of `Normal` or `Pinned`. The downside of the `WeakReference<T>` class is that an instance of it must be allocated on the heap. So the `WeakReference<T>` class is a heavier-weight object than a `GCHandle` instance.

Important When developers start learning about weak references, they immediately start thinking that they are useful in caching scenarios. For example, they think it would be cool to construct a bunch of objects that contain a lot of data and then to create weak references to these objects. When the program needs the data, the program checks the weak reference to see if the object that contains the data is still around, and if it is, the program just uses it; the program experiences high performance. However, if a garbage collection occurred, the objects that contained the data would be destroyed, and when the program has to re-create the data, the program experiences lower performance.

The problem with this technique is the following: garbage collections do not only occur when memory is full or close to full. Instead, garbage collections occur whenever generation 0 is full. So objects are being tossed out of memory much more frequently than desired, and your application's performance suffers greatly.

Weak references can be used quite effectively in caching scenarios, but building a good cache algorithm that finds the right balance between memory consumption and speed is very complex. Basically, you want your cache to keep strong references to all of your objects and then, when you see that memory is getting tight, you start turning strong references into weak references. Currently, the CLR offers no mechanism to notify an application that memory is getting tight. But some people have had much success by periodically calling the Win32 `GlobalMemoryStatusEx` function and checking the returned `MEMORYSTATUSEX` structure's `dwMemoryLoad` member. If this member reports a value above 80, memory is getting tight, and you can start converting strong references to weak references based on whether you want a least-recently used algorithm, a most-frequently used algorithm, a time-base algorithm, or whatever.

Developers frequently want to associate a piece of data with another entity. For example, you can associate data with a thread or with an AppDomain. It is also possible to associate data with an individual object by using the `System.Runtime.CompilerServices.ConditionalWeakTable<TKey,TValue>` class, which looks like this.

```
public sealed class ConditionalWeakTable<TKey, TValue>
   where TKey : class where TValue : class {
   public ConditionalWeakTable();
   public void    Add(TKey key, TValue value);
   public TValue  GetValue(TKey key, CreateValueCallback<TKey, TValue> createValueCallback);
   public Boolean TryGetValue(TKey key, out TValue value);
   public TValue  GetOrCreateValue(TKey key);
   public Boolean Remove(TKey key);

   public delegate TValue CreateValueCallback(TKey key);  // Nested delegate definition
}
```

If you want to associate some arbitrary data with one or more objects, you would first create an instance of this class. Then, call the Add method, passing in a reference to some object for the key parameter and the data you want to associate with the object in the value parameter. If you attempt to add a reference to the same object more than once, the Add method throws an ArgumentException; to change the value associated with an object, you must remove the key and then add it back in with the new value. Note that this class is thread-safe so multiple threads can use it concurrently, although this means that the performance of the class is not stellar; you should test the performance of this class to see how well it works for your scenario.

Of course, a table object internally stores a WeakReference to the object passed in as the key; this ensures that the table doesn't forcibly keep the object alive. But what makes the ConditionalWeakTable class so special is that it guarantees that the value remains in memory as long as the object identified by the key is in memory. So this is more than a normal WeakReference because if it were, the value could be garbage collected even though the key object continued to live. The ConditionalWeakTable class could be used to implement the dependency property mechanism used by XAML. It can also be used internally by dynamic languages to dynamically associate data with objects.

Here is some code that demonstrates the use of the ConditionalWeakTable class. It allows you to call the GCWatch extension method on any object passing in some String tag. Then it notifies you via the console window whenever that particular object gets garbage collected.

```
internal static class ConditionalWeakTableDemo {
   public static void Main() {
      Object o = new Object().GCWatch("My Object created at " + DateTime.Now);
      GC.Collect();     // We will not see the GC notification here
      GC.KeepAlive(o);  // Make sure the object o refers to lives up to here
      o = null;         // The object that o refers to can die now

      GC.Collect();     // We'll see the GC notification sometime after this line
      Console.ReadLine();
   }
}

internal static class GCWatcher {
   // NOTE: Be careful with Strings due to interning and MarshalByRefObject proxy objects
   private readonly static ConditionalWeakTable<Object, NotifyWhenGCd<String>> s_cwt =
      new ConditionalWeakTable<Object, NotifyWhenGCd<String>>();

   private sealed class NotifyWhenGCd<T> {
      private readonly T m_value;

      internal NotifyWhenGCd(T value) { m_value = value; }
      public override string ToString() { return m_value.ToString(); }
      ~NotifyWhenGCd() { Console.WriteLine("GC'd: " + m_value); }
   }

   public static T GCWatch<T>(this T @object, String tag) where T : class {
      s_cwt.Add(@object, new NotifyWhenGCd<String>(tag));
      return @object;
   }
}
```