|  |
| --- |
| **Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework** |
| Jeffrey Richter |
| This article assumes youï¿½re familiar with C and C++ |
| Level of Difficulty     1   2   3 |
| **SUMMARY** Garbage collection in the Microsoft .NET common language runtime environment completely absolves the developer from tracking memory usage and knowing when to free memory. However, you'll want to understand how it works. Part 1 of this two-part article on .NET garbage collection explains how resources are allocated and managed, then gives a detailed step-by-step description of how the garbage collection algorithm works. Also discussed are the way resources can clean up properly when the garbage collector decides to free a resource's memory and how to force an object to clean up when it is freed. |
| Implementing proper resource management for your applications can be a difficult, tedious task. It can distract your concentration from the real problems that you're trying to solve. Wouldn't it be wonderful if some mechanism existed that simplified the mind-numbing task of memory management for the developer? Fortunately, in .NET there is: garbage collection (GC).        Let's back up a minute. Every program uses resources of one sort or anotherâ"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 your program's use. To use any of these resources requires that memory be allocated to represent the type. The steps required to access a resource are as follows:   1. Allocate memory for the type that represents the resource. 2. Initialize the memory to set the initial state of the resource and to make the resource usable. 3. Use the resource by accessing the instance members of the type (repeat as necessary). 4. Tear down the state of the resource to clean up. 5. Free the memory.         This seemingly simple paradigm has been one of the major sources of programming errors. After all, how many times have you forgotten to free memory when it is no longer needed or attempted to use memory after you've already freed it?       These two bugs are worse than most other application bugs because what the consequences will be and when those consequences will occur are typically unpredictable. For other bugs, when you see your application misbehaving, you just fix it. But these two bugs cause resource leaks (memory consumption) and object corruption (destabilization), making your application perform in unpredictable ways at unpredictable times. In fact, there are many tools (such as the Task Manager, the System Monitor ActiveXÂ® Control, CompuWare's BoundsChecker, and Rational's Purify) that are specifically designed to help developers locate these types of bugs.       As I examine GC, you'll notice that it completely absolves the developer from tracking memory usage and knowing when to free memory. However, the garbage collector doesn't know anything about the resource represented by the type in memory. This means that a garbage collector can't know how to perform step fourâ"tearing down the state of a resource. To get a resource to clean up properly, the developer must write code that knows how to properly clean up a resource. In the .NET Framework, the developer writes this code in a Close, Dispose, or Finalize method, which I'll describe later. However, as you'll see later, the garbage collector can determine when to call this method automatically.       Also, many types represent resources that do not require any cleanup. For example, a Rectangle resource can be completely cleaned up simply by destroying the left, right, width, and height fields maintained in the type's memory. On the other hand, a type that represents a file resource or a network connection resource will require the execution of some explicit clean up code when the resource is to be destroyed. I will explain how to accomplish all of this properly. For now, let's examine how memory is allocated and how resources are initialized.  **Resource Allocation**        The MicrosoftÂ® .NET common language runtime requires that all resources be allocated from the managed heap. This is similar to a C-runtime heap except that you never free objects from the managed heapâ"objects are automatically freed when they are no longer needed by the application. This, of course, raises the question: how does the managed heap know when an object is no longer in use by the application? I will address this question shortly.       There are several GC algorithms in use today. Each algorithm is fine-tuned for a particular environment in order to provide the best performance. This article concentrates on the GC algorithm that is used by the common language runtime. Let's start with the basic concepts.       When a process is initialized, the runtime reserves a contiguous region of address space that initially has no storage allocated for it. This address space region is the managed heap. The heap also maintains a pointer, which I'll call the NextObjPtr. This pointer indicates where the next object is to be allocated within the heap. Initially, the NextObjPtr is set to the base address of the reserved address space region.       An application creates an object using the new operator. This operator first makes sure that the bytes required by the new object fit in the reserved region (committing storage if necessary). If the object fits, then NextObjPtr points to the object in the heap, this object's constructor is called, and the new operator returns the address of the object.  Figure 1 Managed Heap **Figure 1** **Managed Heap**        At this point, NextObjPtr is incremented past the object so that it points to where the next object will be placed in the heap. **Figure 1** shows a managed heap consisting of three objects: A, B, and C. The next object to be allocated will be placed where NextObjPtr points (immediately after object C).       Now let's look at how the C-runtime heap allocates memory. In a C-runtime heap, allocating memory for an object requires walking though a linked list of data structures. Once a large enough block is found, that block has to be split, and pointers in the linked list nodes must be modified to keep everything intact. For the managed heap, allocating an object simply means adding a value to a pointerâ"this is blazingly fast by comparison. In fact, allocating an object from the managed heap is nearly as fast as allocating memory from a thread's stack!        So far, it sounds like the managed heap is far superior to the C-runtime heap due to its speed and simplicity of implementation. Of course, the managed heap gains these advantages because it makes one really big assumption: address space and storage are infinite. This assumption is (without a doubt) ridiculous, and there must be a mechanism employed by the managed heap that allows the heap to make this assumption. This mechanism is called the garbage collector. Let's see how it works.       When an application calls the new operator to create an object, there may not be enough address space left in the region to allocate to the object. The heap detects this by adding the size of the new object to NextObjPtr. If NextObjPtr is beyond the end of the address space region, then the heap is full and a collection must be performed.        In reality, a collection occurs when generation 0 is completely full. Briefly, a generation is a mechanism implemented by the garbage collector in order to improve performance. The idea is that newly created objects are part of a young generation, and objects created early in the application's lifecycle are in an old generation. Separating objects into generations can allow the garbage collector to collect specific generations instead of collecting all objects in the managed heap. Generations will be discussed in more detail in Part 2 of this article.  **The Garbage Collection Algorithm**        The garbage collector checks to see if there are any objects in the heap that are no longer being used by the application. If such objects exist, then the memory used by these objects can be reclaimed. (If no more memory is available for the heap, then the new operator throws an OutOfMemoryException.) How does the garbage collector know if the application is using an object or not? As you might imagine, this isn't a simple question to answer.       Every application has a set of roots. Roots identify storage locations, which refer to objects on the managed heap or to objects that are set to null. For example, all the global and static object pointers in an application are considered part of the application's roots. In addition, any local variable/parameter object pointers on a thread's stack are considered part of the application's roots. Finally, any CPU registers containing pointers to objects in the managed heap are also considered part of the application's roots. The list of active roots is maintained by the just-in-time (JIT) compiler and common language runtime, and is made accessible to the garbage collector's algorithm.       When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage. In other words, it assumes that none of the application's roots refer to any objects in the heap. Now, the garbage collector starts walking the roots and building a graph of all objects reachable from the roots. For example, the garbage collector may locate a global variable that points to an object in the heap.       **Figure 2** shows a heap with several allocated objects where the application's roots refer directly to objects A, C, D, and F. All of these objects become part of the graph. When adding object D, the collector notices that this object refers to object H, and object H is also added to the graph. The collector continues to walk through all reachable objects recursively.  Figure 2 Allocated Objects in Heap **Figure 2** **Allocated Objects in Heap**        Once this part of the graph is complete, the garbage collector checks the next root and walks the objects again. As the garbage collector walks from object to object, if it attempts to add an object to the graph that it previously added, then the garbage collector can stop walking down that path. This serves two purposes. First, it helps performance significantly since it doesn't walk through a set of objects more than once. Second, it prevents infinite loops should you have any circular linked lists of objects.       Once all the roots have been checked, the garbage collector's graph contains the set of all objects that are somehow reachable from the application's roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space). The garbage collector then shifts the non-garbage objects down in memory (using the standard memcpy function that you've known for years), removing all of the gaps in the heap. Of course, moving the objects in memory invalidates all pointers to the objects. So the garbage collector must modify the application's roots so that the pointers point to the objects' new locations. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well. **Figure 3** shows the managed heap after a collection.  Figure 3 Managed Heap after Collection **Figure 3** **Managed Heap after Collection**        After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been fixed-up, the NextObjPtr is positioned just after the last non-garbage object. At this point, the new operation is tried again and the resource requested by the application is successfully created.       As you can see, a GC generates a significant performance hit, and this is the major downside of using a managed heap. However, keep in mind that GCs only occur when the heap is full and, until then, the managed heap is significantly faster than a C-runtime heap. The runtime's garbage collector also offers some optimizations that greatly improve the performance of garbage collection. I'll discuss these optimizations in Part 2 of this article when I talk about generations.       There are a few important things to note at this point. You no longer have to implement any code that manages the lifetime of any resources that your application uses. And notice how the two bugs I discussed at the beginning of this article no longer exist. First, it is not possible to leak resources, since any resource not accessible from your application's roots can be collected at some point. Second, it is not possible to access a resource that is freed, since the resource won't be freed if it is reachable. If it's not reachable, then your application has no way to access it. The code in [Figure 4](http://msdn.microsoft.com/en-us/magazine/bb985013.aspx) demonstrates how resources are allocated and managed.        If GC is so great, you might be wondering why it isn't in ANSI C++. The reason is that a garbage collector must be able to identify an application's roots and must also be able to find all object pointers. The problem with C++ is that it allows casting a pointer from one type to another, and there's no way to know what a pointer refers to. In the common language runtime, the managed heap always knows the actual type of an object, and the metadata information is used to determine which members of an object refer to other objects.  **Finalization**        The garbage collector offers an additional feature that you may want to take advantage of: finalization. Finalization allows a resource to gracefully clean up after itself when it is being collected. By using finalization, a resource representing a file or network connection is able to clean itself up properly when the garbage collector decides to free the resource's memory.       Here is an oversimplification of what happens: when the garbage collector detects that an object is garbage, the garbage collector calls the object's Finalize method (if it exists) and then the object's memory is reclaimed. For example, let's say you have the following type (in C#):  public class BaseObj {  public BaseObj() {  }  protected override void Finalize() {  // Perform resource cleanup code here...  // Example: Close file/Close network connection  Console.WriteLine("In Finalize.");  }  }  Now you can create an instance of this object by calling:  BaseObj bo = new BaseObj();        Some time in the future, the garbage collector will determine that this object is garbage. When that happens, the garbage collector will see that the type has a Finalize method and will call the method, causing "In Finalize" to appear in the console window and reclaiming the memory block used by this object.       Many developers who are used to programming in C++ draw an immediate correlation between a destructor and the Finalize method. However, let me warn you right now: object finalization and destructors have very different semantics and it is best to forget everything you know about destructors when thinking about finalization. Managed objects never have destructorsâ"period.        When designing a type it is best to avoid using a Finalize method. There are several reasons for this:   * Finalizable objects get promoted to older generations, which increases memory pressure and prevents the object's memory from being collected when the garbage collector determines the object is garbage. In addition, all objects referred to directly or indirectly by this object get promoted as well. Generations and promotions will be discussed in Part 2 of this article. * Finalizable objects take longer to allocate. * Forcing the garbage collector to execute a Finalize method can significantly hurt performance. Remember, each object is finalized. So if I have an array of 10,000 objects, each object must have its Finalize method called. * Finalizable objects may refer to other (non-finalizable) objects, prolonging their lifetime unnecessarily. In fact, you might want to consider breaking a type into two different types: a lightweight type with a Finalize method that doesn't refer to any other objects, and a separate type without a Finalize method that does refer to other objects. * You have no control over when the Finalize method will execute. The object may hold on to resources until the next time the garbage collector runs. * When an application terminates, some objects are still reachable and will not have their Finalize method called. This can happen if background threads are using the objects or if objects are created during application shutdown or AppDomain unloading. In addition, by default, Finalize methods are not called for unreachable objects when an application exits so that the application may terminate quickly. Of course, all operating system resources will be reclaimed, but any objects in the managed heap are not able to clean up gracefully. You can change this default behavior by calling the System.GC type's RequestFinalizeOnShutdown method. However, you should use this method with care since calling it means that your type is controlling a policy for the entire application. * The runtime doesn't make any guarantees as to the order in which Finalize methods are called. For example, let's say there is an object that contains a pointer to an inner object. The garbage collector has detected that both objects are garbage. Furthermore, say that the inner object's Finalize method gets called first. Now, the outer object's Finalize method is allowed to access the inner object and call methods on it, but the inner object has been finalized and the results may be unpredictable. For this reason, it is strongly recommended that Finalize methods not access any inner, member objects.         If you determine that your type must implement a Finalize method, then make sure the code executes as quickly as possible. Avoid all actions that would block the Finalize method, including any thread synchronization operations. Also, if you let any exceptions escape the Finalize method, the system just assumes that the Finalize method returned and continues calling other objects' Finalize methods.       When the compiler generates code for a constructor, the compiler automatically inserts a call to the base type's constructor. Likewise, when a C++ compiler generates code for a destructor, the compiler automatically inserts a call to the base type's destructor. However, as I've said before, Finalize methods are different from destructors. The compiler has no special knowledge about a Finalize method, so the compiler does not automatically generate code to call a base type's Finalize method. If you want this behaviorâ"and frequently you doâ"then you must explicitly call the base type's Finalize method from your type's Finalize method:  public class BaseObj {  public BaseObj() {  }  protected override void Finalize() {  Console.WriteLine("In Finalize.");  base.Finalize(); // Call base type's Finalize  }  }        Note that you'll usually call the base type's Finalize method as the last statement in the derived type's Finalize method. This keeps the base object alive as long as possible. Since calling a base type Finalize method is common, C# has a syntax that simplifies your work. In C#, the following code  class MyObject {  ~MyObject() {  â¢â¢â¢  }  }  causes the compiler to generate this code:   class MyObject {  protected override void Finalize() {  â¢â¢â¢  base.Finalize();  }  }  Note that this C# syntax looks identical to the C++ language's syntax for defining a destructor. But remember, C# doesn't support destructors. Don't let the identical syntax fool you.  **Finalization Internals**        On the surface, finalization seems pretty straightforward: you create an object and when the object is collected, the object's Finalize method is called. But there is more to finalization than this.       When an application creates a new object, the new operator allocates the memory from the heap. If the object's type contains a Finalize method, then a pointer to the object is placed on the finalization queue. The finalization queue is an internal data structure controlled by the garbage collector. Each entry in the queue points to an object that should have its Finalize method called before the object's memory can be reclaimed.        **Figure 5** shows a heap containing several objects. Some of these objects are reachable from the application's roots, and some are not. When objects C, E, F, I, and J were created, the system detected that these objects had Finalize methods and pointers to these objects were added to the finalization queue.  Figure 5 A Heap with Many Objects **Figure 5** **A Heap with Many Objects**        When a GC occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalization queue looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalization queue and appended to the freachable queue (pronounced "F-reachable"). The freachable queue is another internal data structure controlled by the garbage collector. Each pointer 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 6**. Here, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects did not have a Finalize method that needed to be called. However, the memory occupied by objects E, I, and J could not be reclaimed because their Finalize method has not been called yet.  Figure 6 Managed Heap after Garbage Collection **Figure 6** **Managed Heap after Garbage Collection**        There is a special runtime thread dedicated to calling Finalize methods. When the freachable queue is empty (which is usually the case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's Finalize method. Because of this, you should not execute any code in a Finalize method that makes any assumption about the thread that's executing the code. For example, avoid accessing thread local storage in the Finalize method.       The interaction of the finalization queue and the freachable queue is quite fascinating. First, let me tell you how the freachable queue got its name. The f is obvious and stands for finalization; every entry in the freachable queue should have its Finalize method called. The "reachable" part of the name means that the objects are reachable. To put it another way, the freachable queue is considered to be a root just like global and static variables are roots. Therefore, if an object is on the freachable queue, then the object is reachable and is not garbage.        In short, when an object is not reachable, the garbage collector considers the object garbage. Then, when the garbage collector moves an object's entry from the finalization queue to the freachable queue, the object is no longer considered garbage and its memory is not reclaimed. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been reclassified as not garbage. The garbage collector compacts the reclaimable memory and the special runtime thread empties the freachable queue, executing each object's Finalize method.  Figure 7 Managed Heap after Second Garbage Collection **Figure 7** **Managed Heap after Second Garbage Collection**        The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed. The important thing to understand here is that two GCs are required to reclaim memory used by objects that require finalization. In reality, more than two collections may be necessary since the objects could get promoted to an older generation. **Figure 7** shows what the managed heap looks like after the second GC.  **Resurrection**        The whole concept of finalization is fascinating. However, there is more to it than what I've described so far. You'll notice in the previous section that when an application is no longer accessing a live object, the garbage collector considers the object to be dead. However, if the object requires finalization, the object is considered live again until it is actually finalized, and then it is permanently dead. In other words, an object requiring finalization dies, lives, and then dies again. This is a very interesting phenomenon called resurrection. Resurrection, as its name implies, allows an object to come back from the dead.       I've already described a form of resurrection. When the garbage collector places a reference to the object on the freachable queue, the object is reachable from a root and has come back to life. Eventually, the object's Finalize method is called, no roots point to the object, and the object is dead forever after. But what if an object's Finalize method executed code that placed a pointer to the object in a global or static variable?  public class BaseObj {  protected override void Finalize() {  Application.ObjHolder = this;  }  }  class Application {  static public Object ObjHolder; // Defaults to null  â¢â¢â¢  }        In this case, when the object's Finalize method executes, a pointer to the object is placed in a root and the object is reachable from the application's code. This object is now resurrected and the garbage collector will not consider the object to be garbage. The application is free to use the object, but it is very important to note that the object has been finalized and that using the object may cause unpredictable results. Also note: if BaseObj contained members that pointed to other objects (either directly or indirectly), all objects would be resurrected, since they are all reachable from the application's roots. However, be aware that some of these other objects may also have been finalized.        In fact, when designing your own object types, objects of your type can get finalized and resurrected totally out of your control. Implement your code so that you handle this gracefully. For many types, this means keeping a Boolean flag indicating whether the object has been finalized or not. Then, if methods are called on your finalized object, you might consider throwing an exception. The exact technique to use depends on your type.       Now, if some other piece of code sets Application.ObjHolder to null, the object is unreachable. Eventually the garbage collector will consider the object to be garbage and will reclaim the object's storage. Note that the object's Finalize method will not be called because no pointer to the object exists on the finalization queue.       There are very few good uses of resurrection, and you really should avoid it if possible. However, when people do use resurrection, they usually want the object to clean itself up gracefully every time the object dies. To make this possible, the GC type offers a method called ReRegisterForFinalize, which takes a single parameter: the pointer to an object.  public class BaseObj {  protected override void Finalize() {  Application.ObjHolder = this;  GC.ReRegisterForFinalize(this);  }  }        When this object's Finalize method is called, it resurrects itself by making a root point to the object. The Finalize method then calls ReRegisterForFinalize, which appends the address of the specified object (this) to the end of the finalization queue. When the garbage collector detects that this object is unreachable again, it will queue the object's pointer on the freachable queue and the Finalize method will get called again. This specific example shows how to create an object that constantly resurrects itself and never dies, which is usually not desirable. It is far more common to conditionally set a root to reference the object inside the Finalize method.        Make sure that you call ReRegisterForFinalize no more than once per resurrection, or the object will have its Finalize method called multiple times. This happens because each call to ReRegisterForFinalize appends a new entry to the end of the finalization queue. When an object is determined to be garbage, all of these entries move from the finalization queue to the freachable queue, calling the object's Finalize method multiple times.  **Forcing an Object to Clean Up**        If you can, you should try to define objects that do not require any clean up. Unfortunately, for many objects, this is simply not possible. So for these objects, you must implement a Finalize method as part of the type's definition. However, it is also recommended that you add an additional method to the type that allows a user of the type to explicitly clean up the object when they want. By convention, this method should be called Close or Dispose.        In general, you use Close if the object can be reopened or reused after it has been closed. You also use Close if the object is generally considered to be closed, such as a file. On the other hand, you would use Dispose if the object should no longer be used at all after it has been disposed. For example, to delete a System.Drawing.Brush object, you call its Dispose method. Once disposed, the Brush object cannot be used, and calling methods to manipulate the object may cause exceptions to be thrown. If you need to work with another Brush, you must construct a new Brush object.       Now, let's look at what the Close/Dispose method is supposed to do. 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. Let's say that you create a new FileStream object and write just a few bytes of information to it. If these bytes don't fill the buffer, then the buffer is not written to disk. The FileStream type does implement a Finalize method, and when the FileStream object is collected the Finalize method flushes any remaining data from memory to disk and then closes the file.       But this approach may not be good enough for the user of the FileStream type. Let's say that the first FileStream object has not been collected yet, but the application wants to create a new FileStream object using the same disk file. In this scenario, the second FileStream object will fail to open the file if the first FileStream object had the file open for exclusive access. The user of the FileStream object must have some way to force the final memory flush to disk and to close the file.       If you examine the FileStream type's documentation, you'll see that it has a method called Close. When called, this method flushes the remaining data in memory to the disk and closes the file. Now the user of a FileStream object has control of the object's behavior.       But an interesting problem arises now: what should the FileStream's Finalize method do when the FileStream object is collected? Obviously, the answer is nothing. In fact, there is no reason for the FileStream's Finalize method to execute at all if the application has explicitly called the Close method. You know that Finalize methods are discouraged, and in this scenario you're going to have the system call a Finalize method that should do nothing. It seems like there ought to be a way to suppress the system's calling of the object's Finalize method. Fortunately, there is. The System.GC type contains a static method, SuppressFinalize, that takes a single parameter, the address of an object.       [Figure 8](http://msdn.microsoft.com/en-us/magazine/bb985013.aspx) shows FileStream's type implementation. When you call SuppressFinalize, it turns on a bit flag associated with the object. When this flag is on, the runtime knows not to move this object's pointer to the freachable queue, preventing the object's Finalize method from being called.       Let's examine another related issue. It is very common to use a StreamWriter object with a FileStream object.   FileStream fs = new FileStream("C:\\SomeFile.txt",  FileMode.Open, FileAccess.Write, FileShare.Read);  StreamWriter sw = new StreamWriter(fs);  sw.Write ("Hi there");  // The call to Close below is what you should do  sw.Close();  // NOTE: StreamWriter.Close closes the FileStream. The FileStream  // should not be explicitly closed in this scenario  Notice that the StreamWriter's constructor takes a FileStream object as a parameter. Internally, the StreamWriter object saves the FileStream's pointer. Both of these objects have internal data buffers that should be flushed to the file when you're finished accessing the file. Calling the StreamWriter's Close method writes the final data to the FileStream and internally calls the FileStream's Close method, which writes the final data to the disk file and closes the file. Since StreamWriter's Close method closes the FileStream object associated with it, you should not call fs.Close yourself.       What do you think would happen if you removed the two calls to Close? Well, the garbage collector would correctly detect that the objects are garbage and the objects would get finalized. But, the garbage collector doesn't guarantee the order in which the Finalize methods are called. So if the FileStream gets finalized first, it closes the file. Then when the StreamWriter gets finalized, it would attempt to write data to the closed file, raising an exception. Of course, if the StreamWriter got finalized first, then the data would be safely written to the file.       How did Microsoft solve this problem? Making the garbage collector finalize objects in a specific order is impossible because objects could contain pointers to each other and there is no way for the garbage collector to correctly guess the order to finalize these objects. So, here is Microsoft's solution: the StreamWriter type doesn't implement a Finalize method at all. Of course, this means that forgetting to explicitly close the StreamWriter object guarantees data loss. Microsoft expects that developers will see this consistent loss of data and will fix the code by inserting an explicit call to Close.        As stated earlier, the SuppressFinalize method simply sets a bit flag indicating that the object's Finalize method should not be called. However, this flag is reset when the runtime determines that it's time to call a Finalize method. This means that calls to ReRegisterForFinalize cannot be balanced by calls to SuppressFinalize. The code in [Figure 9](http://msdn.microsoft.com/en-us/magazine/bb985013.aspx) demonstrates exactly what I mean.        ReRegisterForFinalize and SuppressFinalize are implemented the way they are for performance reasons. As long as each call to SuppressFinalize has an intervening call to ReRegisterForFinalize, everything works. It is up to you to ensure that you do not call ReRegisterForFinalize or SuppressFinalize multiple times consecutively, or multiple calls to an object's Finalize method can occur.  **Conclusion**        The motivation for garbage-collected environments is to simplify memory management for the developer. The first part of this overview looked at some general GC concepts and internals. In Part 2, I will conclude this discussion. First, I will explore a feature called WeakReferences, which you can use to reduce the memory pressure placed on the managed heap by large objects. Then I'll examine a mechanism that allows you to artificially extend the lifetime of a managed object. Finally, I'll wrap up by discussing various aspects of the garbage collector's performance. I'll discuss generations, multithreaded collections, and the performance counters that the common language runtime exposes, which allow you to monitor the garbage collector's real-time behavior. |