Finalization

This chapter so far addressed in sufficient detail the specifics of managing one type of resources, namely, managed memory. However, in the real world, many other types of resources exist, which can collectively be called *unmanaged resources* because they are not managed by the CLR or by the garbage collector (such as kernel object handles, database connections, unmanaged memory etc.). Their allocation and deallocation are not governed by GC rules, and the standard memory reclamation techniques outlined above do not suffice when they are concerned.

Freeing unmanaged resources requires an additional feature called *finalization*, which associates an object (representing an unmanaged resource) with code that must be executed when the object is no longer needed. Oftentimes, this code should be executed in a deterministic fashion when the resource becomes eligible for deallocation; at other times, it can be delayed for a later non-deterministic point in time.

Manual Deterministic Finalization

Consider a fictitious File class that serves as a wrapper for a Win32 file handle. The class has a member field of type `System.IntPtr` which holds the handle itself. When the file is no longer needed, the `CloseHandle` Win32 API must be called to close the handle and release the underlying resources.

The deterministic finalization approach requires adding a method to the File class that will close the underlying handle. It is then the client's responsibility to call this method, even in the face of exceptions, in order to deterministically close the handle and release the unmanaged resource.

```
class File {
    private IntPtr handle;

    public File(string fileName) {
     handle = CreateFile(...); //P/Invoke call to Win32 CreateFile API
    }
    public void Close() {
     CloseHandle(handle); //P/Invoke call to Win32 CloseHandle API
    }
}
```

This approach is simple enough and is proven to work in unmanaged environments such as C++, where it is the client's responsibility to release resources. However, .NET developers accustomed to the practice of automatic resource reclamation might find this model inconvenient. The CLR is expected to provide a mechanism for automatic finalization of unmanaged resources.

Automatic Non-Deterministic Finalization

The automatic mechanism cannot be deterministic because it must rely on the garbage collector to discover whether an object is referenced. The GC's non-deterministic nature, in turn, implies that finalization will be non-deterministic. At times, this non-deterministic behavior is a show-stopper, because temporary "resource leaks" or holding a shared resource locked for just slightly longer than necessary might be unacceptable behaviors. At other times, it is acceptable, and we try to focus on the scenarios where it is.

Any type can override the protected `Finalize` method defined by `System.Object`to indicate that it requires automatic finalization. The C# syntax for requesting automatic finalization on the File class is the ~ File method. This method is called a *finalizer*, and it must be invoked when the object is destroyed.

Note Incidentally, only reference types (classes) can define a finalizer in C#, even though the CLR does not impose this limitation. However, it typically only makes sense for reference types to define a finalization behavior, because value types are eligible for garbage collection only when they are boxed (see Chapter 3 for a detailed treatment of boxing). When a value type is allocated on the stack, it is never added to the finalization queue. When the stack unfolds as part of returning from a method or terminating the frame because of an exception, value type finalizers are not called.

When an object with a finalizer is created, a reference to it is added to a special runtime-managed queue called the *finalization queue*. This queue is considered a root by the garbage collector, meaning that even if the application has no outstanding reference to the object, it is still kept alive by the finalization queue.

When the object becomes unreferenced by the application and a garbage collection occurs, the GC detects that the only reference to the object is the reference from the finalization queue. The GC consequently moves the object reference to another runtime-managed queue called the *f-reachable queue*. This queue is also considered a root, so at this point the object is still referenced and considered alive.

The object's finalizer is not run during garbage collection. Instead, a special thread called the *finalizer thread* is created during CLR initialization (there is one finalization thread per process, regardless of GC flavor, but it runs at `THREAD_PRIORITY_HIGHEST`). This thread repeatedly waits for the *finalization event* to become signaled. The GC signals this event after a garbage collection completes, if objects were moved to the f-reachable queue, and as a result the finalizer thread wakes up. The finalizer thread removes the object reference from the f-reachable queue and synchronously executes the finalizer method defined by the object. When the next garbage collection occurs, the object is no longer referenced and therefore the GC can reclaim its memory. Figure 4-18 contains all the moving parts:

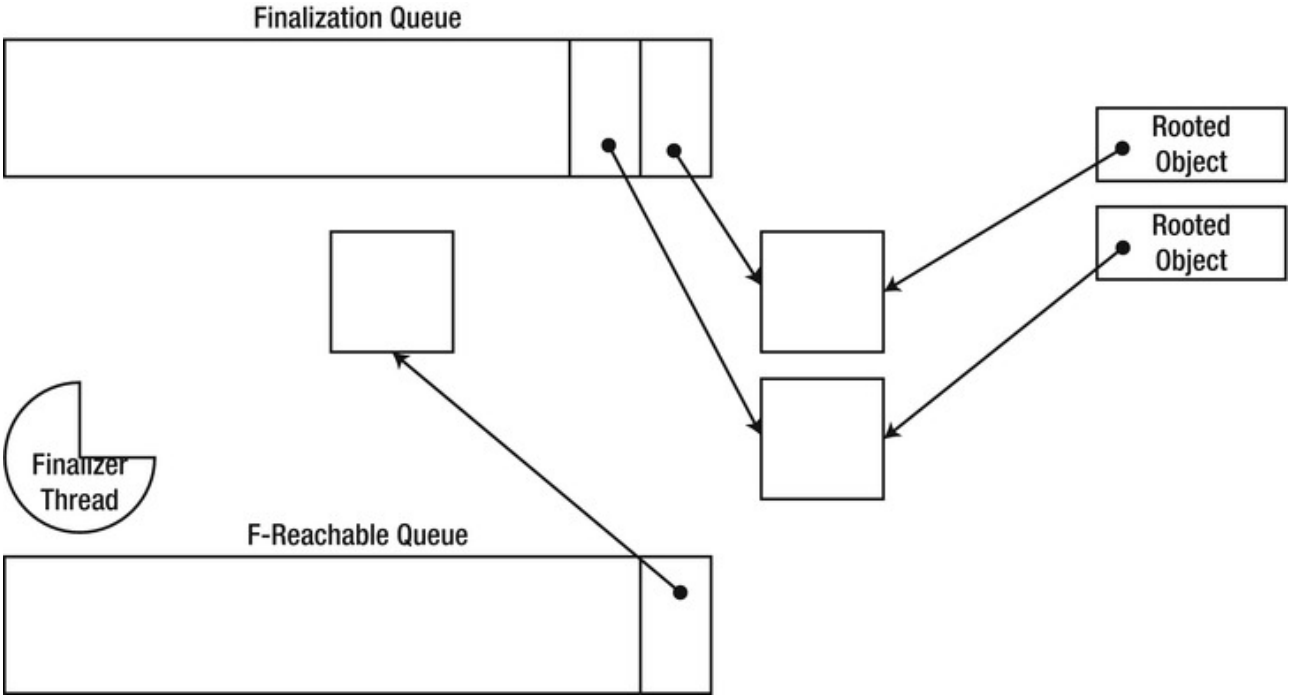

Figure 4-18 . The finalization queue holds references to objects that have a finalizer. When the application stops referencing them, the GC moves the object references to the f-reachable queue. The finalizer thread wakes up and executes the finalizers on these objects, and subsequently releases them

Why doesn't the GC execute the object's finalizer instead of deferring the work to an asynchronous thread? The motivation for doing so is closing the loop without the f-reachable queue or an additional finalizer thread, which would appear to be cheaper. However, the primary risk associated with running finalizers during GC is that finalizers (which are user-defined by their nature) can take a long time to complete, thus blocking the garbage collection process, which in turn blocks all application threads. Additionally, handling exceptions that occur in the middle of GC is not trivial, and handling memory allocations that the finalizer might trigger in the middle of GC is not trivial either. To conclude, because of reliability reasons the GC does not execute finalization code, but defers this processing to a special dedicated thread.

Pitfalls of Non-Deterministic Finalization

The finalization model just described carries a set of performance penalties. Some of them are insignificant, but others warrant reconsideration of whether finalization is applicable for your resources.

- Objects with finalizers are guaranteed to reach at least generation 1, which makes them more susceptible to the mid-life crisis phenomenon. This increases the chances of performing many full collections.
- Objects with finalizers are slightly more expensive to allocate because they are added to the finalization queue. This introduces contention in multi-processor scenarios. Generally speaking, this cost is negligible compared to the other issues.
- Pressure on the finalizer thread (many objects requiring finalization) might cause memory leaks. If the application threads are allocating objects at a higher rate than the finalizer thread is able to finalize them, then the application will steadily leak memory from objects waiting for finalization.

The following code shows an application thread that allocates objects at a higher rate than they can be finalized, because of blocking code executing in the finalizer. This causes a steady memory leak.

```
class File2 {
    public File2() {
     Thread.Sleep(10);
    }
    ~File2() {
     Thread.Sleep(20);
    }
    //Data to leak:
    private byte[] data = new byte[1024];
}

class Program {
    static void Main(string[] args) {
     while (true) {
     File2 f = new File2();
```

```
    }
  }
}
```

EXPERIMENTING WITH A FINALIZATION-RELATED LEAK

In this experiment, you will run a sample application that exhibits a memory leak and perform partial diagnostics before consulting the source code. Without giving away the problem completely, the leak is related to improper use of finalization, and is similar in spirit to the code listing for the File2 class.

1. Run the MemoryLeak.exe application from this chapter's source code folder.
2. Run Performance Monitor and monitor the following performance counters from the .NET CLR Memory category for this application (for more information on Performance Monitor and how to run it, consult Chapter 2): `# Bytes in All Heaps`, `# Gen 0 Collections`, `# Gen 1 Collections`, `# Gen 2 Collections`, `% Time in GC`, `Allocated Bytes/sec`, `Finalization Survivors`, `Promoted Finalization-Memory from Gen 0`.
3. Monitor these counters for a couple of minutes until patterns emerge. For example, you should see the # Bytes in All Heaps counter gradually go up, although it sometimes drops slightly. Overall, the application's memory usage is going up, indicating a likely memory leak.
4. Note that application allocates memory at an average rate of 1MB/s. This is not a very high allocation rate, and indeed the fraction of time in GC is very low—this is not a case of the garbage collector struggling to keep up with the application.
5. Finally, note that the Finalization Survivors counter, whenever it is updated, is quite high. This counter represents the number of objects that survived a recent garbage collection only because they are registered for finalization, and their finalizer has not yet run (in other words, these objects are rooted by the f-reachable queue). The Promoted Finalization-Memory from Gen 0 counter points toward a considerable amount of memory that is being retained by these objects.

Adding these pieces of information together, you can deduce that it is likely that the application is leaking memory because it puts insurmountable pressure on the finalizer thread. For example, it might be creating (and releasing) finalizable resources at a faster rate than the finalizer thread is able to clean them. You can now inspect the application's source code (use .NET Reflector, ILSpy, or any other decompiler for this) and verify that the source of leak is related to finalization, specifically the Employee and Schedule classes.

Aside from performance issues, using automatic non-deterministic finalization is also the source of bugs which tend to be extremely difficult to find and resolve. These bugs occur because finalization is asynchronous by definition, and because the order of finalization between multiple objects is undefined.

Consider a finalizable object *A* that holds a reference to another finalizable object *B*. Because the order of finalization is undefined, *A* can't assume that when its finalizer is called, *B* is valid for use—its finalizer might have executed already. For example, an instance of the `System.IO.StreamWriter` class can hold a reference to an instance of the `System.IO.FileStream` class. Both instances have finalizable resources: the stream writer contains a buffer that must be flushed to the underlying stream, and the file stream has a file handle that must be closed. If the stream writer is finalized first, it will flush the buffer to the valid underlying stream, and when the file stream is finalized it will close the file handle. However, because finalization order is undefined, the opposite scenario might occur: the file stream is finalized first and closes the file handle, and when the stream writer is finalized it flushes the buffer to an invalid file stream that was previously closed. This is an irresolvable issue, and the "solution" adopted by the .NET Framework is that `StreamWriter` does not define a finalizer, and relies on deterministic finalization only. If the client forgets to close the stream writer, its internal buffer is lost.

Tip It *is* possible for resource pairs to define the finalization order among themselves if one of the resources derives from the `System.Runtime.ConstrainedExecution.CriticalFinalizerObject` abstract class, which defines its finalizer as a *critical finalizer*. This special base class guarantees that its finalizer will be called *after* all other non-critical finalizers have been called. It is used by resource pairs such as `System.IO.FileStream` with `Microsoft.Win32.SafeHandles.SafeFileHandle` and `System.Threading.EventWaitHandle` with `Microsoft.Win32.SafeHandles.SafeWaitHandle`.

Another problem has to do with the asynchronous nature of finalization which occurs in a dedicated thread. A finalizer might attempt to acquire a lock that is held by the application code, and the application might be waiting for finalization to complete by calling `GC.WaitForPendingFinalizers()`. The only way to resolve this issue is to acquire the lock with a timeout and fail gracefully if it can't be acquired.

Yet another scenario is caused by the garbage collector's eagerness to reclaim memory as soon as possible. Consider the following code which represents a naïve implementation of a File class with a finalizer that closes the file handle:

```
class File3 {
  Handle handle;
  public File3(string filename) {
    handle = new Handle(filename);
  }
  public byte[] Read(int bytes) {
    return Util.InternalRead(handle, bytes);
  }
  ~File3() {
    handle.Close();
  }
```

```
}

class Program {
  static void Main() {
    File3 file = new File3("File.txt");
    byte[] data = file.Read(100);
    Console.WriteLine(Encoding.ASCII.GetString(data));
  }
}
```

This innocent piece of code can break in a very nasty manner. The Read method can take a long time to complete, and it only uses the handle contained within the object, and not the object itself. The rules for determining when a local variable is considered an active root dictate that the local variable held by the client is no longer relevant after the call to Read has been dispatched. Therefore, the object is considered eligible for garbage collection and its finalizer might execute before the Read method returns! If this happens, we might be closing the handle while it is being used, or just before it is used.

THE FINALIZER MIGHT NEVER BE CALLED

Even though finalization is normally considered a bullet-proof feature that guarantees resource deallocation, the CLR doesn't actually provide a guarantee that a finalizer will be called under any possible set of circumstances.

One obvious scenario in which finalization will not occur is the case of a brutal process shutdown. If the user closes the process via Task Manager or an application calls the TerminateProcess Win32 API, finalizers do not get a chance to reclaim resources. Therefore, it is incorrect to blindly rely on finalizers for cleaning up resources that cross process boundaries (such as deleting files on disk or writing specific data to a database).

Less obvious is the case when the application encounters an out of memory condition and is on the verge of shutting down. Typically, we would expect finalizers to run even in the face of exceptions, but what if the finalizer for some class has never been called yet, and has to be JITted? The JIT requires a memory allocation to compile the finalizer, but there is no memory available. This can be addressed by using .NET pre-compilation (NGEN) or deriving from `CriticalFinalizerObject` which guarantees that the finalizer will be eagerly JITted when the type is loaded.

Finally, the CLR imposes time limitations on finalizers that run as part of process shutdown or AppDomain unload scenarios. In these cases (which can be detected through Environment.HasShutdownStarted or AppDomain.IsFinalizingForUnload()), each individual finalizer has (roughly) two seconds to complete its execution, and all finalizers combined have (roughly) 40 seconds to complete their execution. If these time limits are breached, finalizers might not execute. This scenario can be diagnosed at runtime using the BreakOnFinalizeTimeout registry value. See Sasha Goldshtein's blog post, "Debugging Shutdown Finalization Timeout" (http://blog.sashag.net/archive/2008/08/27/debugging-shutdown-finalization-timeout.aspx, 2008) for more details.

The Dispose Pattern

We have reviewed multiple problems and limitations with the implementation of non-deterministic finalization. It is now appropriate to reconsider the alternative—deterministic finalization—which was mentioned earlier.

The primary problem with deterministic finalization is that the *client* is responsible for using the object properly. This contradicts the object-orientation paradigm, in which the object is responsible for its own state and invariants. This problem cannot be addressed to a full extent because automatic finalization is always non-deterministic. However, we can introduce a contractual mechanism that will strive to ensure that deterministic finalization occurs, and that will make it easier to use from the client side. In exceptional scenarios, we will have to provide automatic finalization, despite all the costs mentioned in the previous section.

The conventional contract established by the .NET Framework dictates that an object which requires deterministic finalization must implement the IDisposable interface, with a single Dispose method. This method should perform deterministic finalization to release unmanaged resources.

Clients of an object implementing the IDisposable interface are responsible for calling Dispose when they have finished using it. In C#, this can be accomplished with a using block, which wraps object usage in a try…finally block and calls Dispose within the finally block.

This contractual model is fair enough if we trust our clients completely. However, often we can't trust our clients to call Dispose deterministically, and must provide a backup behavior to prevent a resource leak. This can be done using automatic finalization, but brings up a new problem: if the client calls Dispose and later the finalizer is invoked, we will release the resources twice. Additionally, the idea of implementing deterministic finalization was avoiding the pitfalls of automatic finalization!

What we need is a mechanism for instructing the garbage collector that the unmanaged resources have already been released and that automatic finalization is no longer required for a particular object. This can be done using the GC.SuppressFinalize method, which disables finalization by setting a bit in the object's header word (see Chapter 3 for more details about the object header). The object still remains in the finalization queue, but most of the finalization cost is not incurred because the object's memory is reclaimed immediately after the first collection, and it is never seen by the finalizer thread.

Finally, we might want a mechanism to alert our clients of the case when a finalizer was called, because it implies that they haven't used the (significantly more efficient, predictable and reliable) deterministic finalization mechanism. This can be done using System.Diagnostics.Debug.Assert or a logging framework of some sort.

The following code is a rough draft of a class wrapping an unmanaged resource that follows these guidelines (there are more details to consider if the class were to derive from another class that also manages unmanaged resources):

```
class File3 : IDisposable {
  Handle handle;
  public File3(string filename) {
    handle = new Handle(filename);
  }
```

```
  public byte[] Read(int bytes) {
    Util.InternalRead(handle, bytes);
  }
  ~File3() {
    Debug.Assert(false, "Do not rely on finalization! Use Dispose!");
    handle.Close();
  }
  public void Dispose() {
    handle.Close();
    GC.SuppressFinalize(this);
  }
}
```

Note The finalization pattern described in this section is called the *Dispose pattern*, and covers additional areas such as the interaction between derived and base classes requiring finalization. For more information on the Dispose pattern, refer to the MSDN documentation. Incidentally, C++/CLI implements the Dispose pattern as part of its native syntax: !File is the C++/CLI finalizer and ~ File is the C++/CLI IDisposable.Dispose implementation. The details of calling the base class and ensuring finalization is suppressed are taken care of automatically by the compiler.

Ensuring that your implementation of the Dispose pattern is correct is not as difficult as ensuring that clients using your class will use deterministic finalization instead of automatic finalization. The assertion approach outlined earlier is a brute-force option that works. Alternatively, static code analysis can be used to detect improper use of disposable resources.

Resurrection

Finalization provides an opportunity for an object to execute arbitrary code after it is no longer referenced by the application. This opportunity can be used to create a new reference from the application to the object, reviving the object after it was already considered dead. This ability is called *resurrection*.

Resurrection is useful in a handful of scenarios, and should be used with great care. The primary risk is that other objects referenced by your object might have an invalid state, because their finalizers might have run already. This problem can't be solved without reinitializing all objects referenced by your object. Another issue is that your object's finalizer will not run again unless you use the obscure GC.ReRegisterForFinalize method, passing a reference to your object (typically this) as a parameter.

One of the applicable scenarios for using resurrection is *object pooling*. Object pooling implies that objects are allocated from a pool and returned to the pool when they are no longer used, instead of being garbage collected and recreated. Returning the object to the pool can be performed deterministically or delayed to finalization time, which is a classic scenario for using resurrection.