Memory Leaks

Memory leaks occur when accessing unmanaged code that does not properly manage its memory. As discussed in previous chapters, dead objects in managed code are handled by the GC. Dead objects are automatically collected. While behavior that looks suspiciously like memory leaks *can* occur in managed code, the cause is subtly (but importantly) different, and we'll briefly touch upon the distinction in a moment. Thanks to the GC, a true memory leak is rare in managed code.

Unmanaged resources should be released when they are no longer needed, and the onus to do this is, naturally, entirely on the developer. In stamping out memory leaks, you will spend a good deal of time working through your code handling these situations to ensure that your application is properly releasing references to any expensive unmanaged (or "system") resources.

Pseudo memory leaks (memory hogs) occur when the GC is unable to collect objects because they are being kept alive by other objects. This is a coding error and can be fixed by analyzing the code which references the object in question. Although not a true memory leak (as the developer is not directly responsible for resource allocation), the effect is the same: the application consumes more resources than necessary and, in worst case scenarios, encounters out of memory exceptions. With proper attention paid to the object lifespan, this can be avoided even in long-running applications. We will explore various issues that cause such memory leaks throughout the rest of this chapter.

Disposing of unmanaged resources

The `IDisposable` interface defines the `Dispose` method used to release allocated (i.e. unmanaged) resources. When utilizing a class that implements this interface, it is clearly best to ensure that the `Dispose` method is called when the object is no longer needed. There are two ways to do so without an exception preventing the method's execution.

```
FileStream stream = new FileStream("message.txt", FileMode.Open);
try
{
 StreamReader reader = new StreamReader(stream);
 try
 {
  Console.WriteLine(reader.ReadToEnd());
 }
 finally
 {
  reader.Dispose();
 }
}
finally
{
 stream.Dispose();
}
```

Listing 4.9: Calling the `Dispose` method through a `finally` block.

```
using (FileStream stream = new FileStream("message.txt", FileMode.Open))
using (StreamReader reader = new StreamReader(stream))
{
 Console.WriteLine(reader.ReadToEnd());
}
```

Listing 4.10: Calling the `Dispose` method with a `using` statement.

In most situations, the `using` syntax is cleaner, but there will be times when a `try` block is preferable. Later, we will see an example in WCF where the `Disposable` pattern is not implemented properly, causing an exception to be thrown and the `Dispose` method to be ignored. In this case, we want to use the `finally` statement to ensure that the `Dispose` will be called even if there is an exception.

The important thing is that the `Dispose` method is called, whether explicitly or implicitly. If it is not, and the `Disposable` pattern is implemented properly in the object, its resources will still be picked up at garbage collection. But this will be done in a non-deterministic way, since we don't know when the GC will run. More importantly, the process of garbage collection is complicated enough without having to worry about disposing of our objects, and having to dispose of our managed resources will slow this process down.

So, as a best practice, if an object implements the `IDisposable` interface, call the `Dispose` method when you're finished with it, employing either a `using` statement or the `finalize` block. Either approach will result in the resources being freed in a more timely manner and will simplify the garbage collection process.

Finalization

A common problem with custom implementations of disposable objects is that these implementations are not complete, and so resources are not released. This is the definition of the `IDisposable` interface (see Figure 4.1) and this is sometimes directly implemented on a class without any attention given to the finalizer (if you are tempted to draw parallels to destructors, don't. They are syntactically similar, but semantically very different).

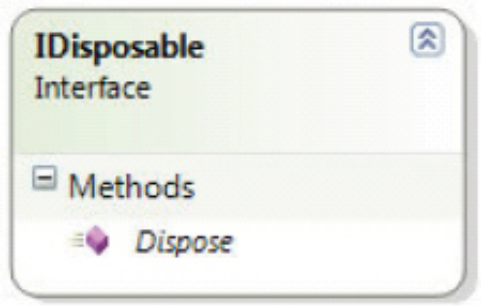

Figure 4.1: The `IDisposable` definition.

```
public class Sample :IDisposable
{
 public void Dispose()
 {
  // Clean up resources
 }
}
```

Listing 4.11: Sample implementation of the interface.

The issue arises when the `Dispose` method isn't called. Despite all the best practices, it is important to realize that not everyone remembers to call the `Dispose` method, or to employ the `using` statement in conjunction with a `disposable.` Sometimes a developer may not have noticed the `Dispose` method in the Intellisense method list. Or they may get caught up with timelines and simply miss an all-too-common bug.

The *full* implementation of the `disposable` pattern depends on providing a finalizer to call the `Dispose` method if it is not explicitly called during normal usage. Any code with unmanaged resources that does *not* have a finalizer defined could

introduce memory leaks to your software. Even worse, if the developer using the object does not explicitly call `Dispose` to release the unmanaged resources *and* there is no finalizer, then the unmanaged resources may become orphaned.

```
public class Sample :IDisposable
{
 public void Dispose()
 {
  // Clean up resources
  GC.SuppressFinalize(this);
 }
 ~Sample ()
 {
  this.Dispose();
 }
}
```

Listing 4.12: Sample implementation of the `IDisposable` interface with the finalizer.

In this example, the `~Sample()` is the finalizer, and you may notice that the finalizer looks a lot like a destructor from C++. This often causes some confusion, but the C# finalizer is not a true destructor; it is a language implementation for the `Finalize` method. This can be an area of confusion for developers coming from a C++ background, but you can see the difference if you look at how the above code gets translated by Reflector with no optimizations defined.

```
public class Samplel : IDisposable
{

 public void Dispose()
 {
  GC.SuppressFinalize(this);
  return;
 }

 protected override void Finalize()
 {
 Label_0000:
  try
  {
   this.Dispose();
   goto Label_0013;
   }
   finally
   {
   Label_000B:
    base.Finalize();
   }
 Label_0013:
   return;
 }
}
```

Listing 4.13: The underlying implementation of a finalizer.

The correct approach is to use the C++ style destructor syntax, but remember that this is *not* a C++ finalizer. A destructor is responsible for reclaiming all of the resources, whereas the finalizer only needs to worry about calling the `Dispose` method. The second big difference is that the finalizer is not deterministic. A true destructor is implicitly called as soon as the object goes out of scope, but a finalizer is called as a part of garbage collecting an object that has defined a finalizer. We will see, later, that *when* the finalizer is actually called is even less well defined than when the GC runs. You cannot make any assumptions about when the finalizer will run.

To make sure this all runs smoothly when finalizers are correctly implemented, the GC keeps track of the objects with finalizers. When such an object is created, a reference to it is added to the **finalization** queue, a structure maintained internal to the GC, which keeps track of the objects that need to have their finalizer called. When an object in the finalization queue is collected, the GC checks to see if it needs to call the `Finalize` method. If it does, the GC will explicitly call the finalizer – we'll come back to that process in just a moment. If the finalizer does not need to be called, then the object will be immediately collected.

When the GC needs to call an object's finalizer, it will be added to the `fReachable` queue, another structure maintained by the GC. At this point, the object is reachable only through the `fReachable` queue, which becomes the only root to the object. Essentially, at this point, the object has been collected (since only the GC can access it) but the finalizer has not been called and allocated resources have not been reclaimed. The finalizer will be called the next time the GC runs in a separate, high-priority thread spawned by the GC.

This has a couple of implications that we need to be aware of. To start with, the GC will have to deal with our object twice (once to make its initial collection attempt, and once to *actually* collect it – a third time if you consider the call to the finalizer), which is unnecessarily resource-intensive. The other important thing to bear in mind is that our finalizer will not be called from the same thread that the object was created in. This means that you should not make any assumptions about the running thread when the finalizer is called.

Clearly, we want to avoid having objects added to the `fReachable` queue.

To make sure that the object is *not* placed in the `fReachable` queue, the GC must be informed that finalization should be suppressed after `Dispose` is called. So, in addition to cleaning up the unmanaged resources, be sure to call `GC.SuppressFinalize(this)` to tell the GC to *not* run the finalizer.

If you remember to do that then, as long as the developer using your class calls the `Dispose` method, all of your unmanaged resources will be cleaned up and we have not added any extra load to the GC. Even if the developer *doesn't* call `Dispose`, we are still protected from orphaned resources; life is just a little harder for the GC, and we have to wait a little later for the resources to be cleared up.