Allocation of Disposable Members

By Bill Wagner
April 2011

Introduction

Object lifetimes continue to cause developers confusion. It seems that most developers understand the normal object lifetime cycle. Where the confusion starts is when developers need to write code to handle cases where objects don't get constructed correctly, or when they are used after being disposed.

.NET, like all garbage collected environments, simplifies the task of managing object ownership and object lifetime. Furthermore, by implementing the Dispose pattern types can handle those cases where an object owns non-memory resources that must be freed. I won't cover that pattern in detail here. See Section 9.4 of the Framework Design Guidelines (2nd edition), or Item 17 in Effective C# (2nd edition) for details on how to implement the pattern. In the samples below, I'm not implementing the full pattern, and some of the examples show the effects of having a finalizer, even though the sample types do not need one. I'm doing that here to save space and focus on techniques I am explaining. Please use the references above for guidance for production code.

What happens if an object that allocates disposable objects encounters problems in its constructor? Typical implementations of the dispose pattern do not correctly handle this scenario. The object hasn't quite finished creating itself, and isn't accessible to any portion of your program yet. Consider the following (obviously contrived) sample:

```
class Program
{
   static void Main(string[] args)
   {
      using (var owner = new OwnerOfDisposableThings())
      {
         owner.DoThings();
      };
   }
}
```

```
public class OwnerOfDisposableThings : IDisposable
{
   private DisposableThing memberOne;
   private DisposableThing memberTwo;

   public OwnerOfDisposableThings()
   {
      memberOne = DisposableThing.GetFirst();
      memberTwo = DisposableThing.GetSecond();
   }

   internal void DoThings() { }

   public void Dispose()
   {
      memberOne.Dispose();
      memberTwo.Dispose();
   }
}

public class DisposableThing : IDisposable
{
   public static DisposableThing GetFirst()
   {
      return new DisposableThing();
   }

   public static DisposableThing GetSecond()
   {
```

```
        throw new InvalidOperationException("Sample failure");
    }

    public void Dispose() { Console.WriteLine("I've been Disposed"); }
}
```

When Constructors Throw Exceptions

If you run the above code sample, you'll see that Dispose() is never called on the OwnerOfDisposableThings object. Neither of the DisposableThing objects (memberOne and memberTwo) get disposed. That's because its constructor did not complete successfully, and the reference to it (owner) never gets assigned. The constructor throws an exception instead of completing successfully. That means the program does not does not complete the resource acquisition and does not enter the embedded statement for the using block.

Though this sample is contrived designed to demonstrate the issue, you can easily run into this in practice. Whenever you create a type that implements IDisposable, you must be careful about your own resource acquisition code to make sure that you do not leak resources when the constructors exit by throwing exceptions.

The potential to leak resources occurs because this code allocates objects of disposable types in its constructor, then goes on to do more work. That later work might fail, causing the constructor to fail. It didn't completely get constructed, and therefore the calling code has no reference to it. It cannot be disposed by the calling code. You need to code your constructors carefully so that your constructor explicitly cleans up anything it's allocated when this happens.

As you read through the rest of this column, understand that this situation is a single case of a common issue: when constructors throw exceptions, the constructor must clean up whatever work it had in progress. No other application code can because the object's constructor did not finish.

There is one other wrinkle to this situation: If the new object has a finalizer, it will get called, provided your program exits normally. Note this example:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            using (var owner = new OwnerOfDisposableThings())
```

```
        {
            owner.DoThings();
        };
    } catch (InvalidOperationException)
    {
        // so program ends normally and finalizers are called
    }
  }
}

public class OwnerOfDisposableThings : IDisposable
{
  private DisposableThing memberOne;
  private DisposableThing memberTwo;

  public OwnerOfDisposableThings()
  {
    memberOne = DisposableThing.GetFirst();
    memberTwo = DisposableThing.GetSecond();
  }

  public void DoThings() { }

  public void Dispose()
  {
    memberOne.Dispose();
    memberTwo.Dispose();
  }
}
```

```
public class DisposableThing : IDisposable
{
   public static DisposableThing GetFirst()
   {
      return new DisposableThing();
   }

   public static DisposableThing GetSecond()
   {
      throw new InvalidOperationException("Sample failure");
   }

   public void Dispose() { Console.WriteLine("I've been Disposed"); }
   ~DisposableThing() { Console.WriteLine("I've been Finalized"); }
}
```

If you run this version, you will see that even though Dispose() never gets called, the finalizer for the DisposableThing is executed. Finalizers have their own performance related issues, as detailed in the references above. Do not use a finalizer to fix this problem. The rest of this column will show you two better techniques to make sure that your types execute all necessary cleanup code not matter what bad things happen.

Goals of a fix

Before diving into techniques you can use to fix this situation, let's look at the goals.

The overall goal is to avoid leaking resources. You don't want any exceptional condition to result in open file handles, unmanaged memory leaks, COM objects not being cleaned up, or similar concerns. More specifically, the goal is to clean up resources as soon as they are no longer needed.

Next, we'd like any solution to work in multi-threaded environments without extra burdens on callers. One reason to acquire resources in constructors is that a constructor does not need any synchronization code: only one thread can create the same object. One justification for doing more work in a constructor is to avoid synchronization of multiple threads when an object must acquire exactly one copy of a resource. For example, a class that writes to a stream should not try and open the same file in write exclusive mode twice. Here, I'm concerned with exceptions that could occur in typical

programming situations (the network is down, the file is locked, and so on). I'm not discussing conditions such as thread aborts, and some system exceptions that indicate serious runtime problems.

Third, if possible, we'd like our solution to avoid relying on finalizers except when explicitly necessary. Our goal is to release resources as quickly as possible. Finalizers do not run as quickly as possible; the garbage collector will run them at some later time. "Some later time" does not meet the goal of "as quickly as possible". (See the Framework Design Guidelines for a detailed discussion about the performance implications of finalizers). In fact, the remaining samples do not include a finalizer at all. You should include a finalizer in your types if and only if they directly allocate unmanaged resources. If you indirectly allocate unmanaged resources, such as opening a file using the System.IO classes, you only need to use Dispose(). The framework classes that acquire the unmanaged resource (like the operating system file handles) use a finalizer to clean up the unmanaged handles. Users of those classes do not need to provide that code.

Finally, we want to keep the API for this type as easy to use as possible. After all, real developers are busy and you save them time by creating types that are as easy to use as possible.

There are two general techniques you can use to avoid this problem: two-phase construction and defensive constructors. Both have advantages and disadvantages, and a complete solution may well include both techniques.

Separate Allocation from Construction

The first technique is to finesse the problem by not allocating resources in your constructor. Simply set all your object's member variables to default values (like null) and force callers to initialize resources in a separate call. You can update the original sample like this:

```
class Program
{
  static void Main(string[] args)
  {
    try
    {
      using (var owner = new OwnerOfDisposableThings())
      {
        owner.Acquire();
        owner.DoThings();
      };
    } catch (InvalidOperationException)
    {
```

```
            // so program ends normally and finalizers are called
        }
    }
}

public class OwnerOfDisposableThings : IDisposable
{
    private DisposableThing memberOne;
    private DisposableThing memberTwo;

    public OwnerOfDisposableThings()
    {
        // memberOne and memberTwo are null.
    }

    public void Acquire()
    {
        memberOne = DisposableThing.GetFirst();
        memberTwo = DisposableThing.GetSecond();
    }
    public void DoThings() { }

    public void Dispose()
    {
        if (memberOne != null)
            memberOne.Dispose();
        if (memberTwo != null)
            memberTwo.Dispose();
    }
```

```
}


public class DisposableThing : IDisposable
{
   public static DisposableThing GetFirst()
   {
      return new DisposableThing();
   }

   public static DisposableThing GetSecond()
   {
      throw new InvalidOperationException("Sample failure");
   }

   public void Dispose() { Console.WriteLine("I've been Disposed"); }
}
```

This version cannot not leak resources acquired in the constructor, because no resources were acquired in the constructor. The Dispose() method can test if either or both memberOne and memberTwo were allocated before freeing those resources. A second advantage is that an object can be created long before its resources are acquired. This moves the acquisition closer to the point in time where those resources are used, thereby decreasing the window of time in which the precious resources are in use.

Let's see how this technique works with our three stated goals. First of all, it won't leak resources. No disposable objects are allocated until the object is fully constructed, so callers can follow standard techniques to clean up after the allocation.

The second goal is to ensure that this code works well in multi-threaded environments, without placing extra burden on the caller. This idiom doesn't add any new burdens on callers, but the burden of lifetime ownership was already on the caller. Imagine that an OwnerOfDisposableThings was being used by multiple threads. Exactly one of those threads would have to create the object. Exactly one of those threads would have to call Dispose(). Now, exactly one of those threads has to call Acquire(), and exactly one of those threads has to call Dispose(). You might be considering putting locks in both Acquire() and Dispose(). However, that really doesn't help. Consider this sequence of calls on the same object:

- Thread 1 calls Acquire().
- Thread 2 calls Dispose()

- Thread 1 calls DoThings() (causes ObjectDisposedException)

No amount of locking in the OwnerOfDisposableThings class will fix that problem. The only solution is for the calling code to have an explicit agreement about acquiring resources and disposing them. If that protocol exists, OwnerOfDisposableThings does not need locking. Without it, no amount of locking will magically fix the issue.

But remember that we started down this path because we were concerned that our constructor might throw exceptions. What happens if this new Acquire() method throws an exception? The caller can still call Dispose(), and the implementation of Dispose() must determine if one or both needed resources has been created.

The third goal is met because the OwnerOfDisposableThings does not need a finalizer.

On the last goal, we've taken a step back. The new API is a bit more clumsy than before. We've forced our users to make an extra call before using the disposable resource. Of course, you can fix that by calling Acquire() for them before they attempt to use the object.

Before closing the section on this technique, let's examine how well this technique composes with itself. What does the code look like if DisposableThing itself has an Acquire() to allocate whatever it needs before you can use it. It composes quite well. You can simply call Acquire() in your own Acquire method:

```
public void Acquire()
{
   memberOne = DisposableThing.GetFirst();
   memberTwo = DisposableThing.GetSecond();
   memberOne.Acquire();
   memberTwo.Acquire();
}
```

Clean up your Constructors

The other strategy is to be very defensive about resource allocations in your constructor. You continue to create the disposable resources in the constructor, so that callers can immediately use it. However, now you have to be very careful so that any resources you've allocated are freed even if the constructor exits by throwing an exception. It looks similar to the first example, but all the work in the constructor is enclosed in a try/catch block. In the catch block, you call your own Dispose() method, and then rethrow the exception.

```
public class OwnerOfDisposableThings : IDisposable
{
   private DisposableThing memberOne;
```

```
    private DisposableThing memberTwo;

    public OwnerOfDisposableThings()
    {
        try
        {
            memberOne = DisposableThing.GetFirst();
            memberTwo = DisposableThing.GetSecond();
        }
        catch (Exception)
        {
            Dispose();
            throw;
        }
    }

    public void DoThings() { }

    public void Dispose()
    {
        if (memberOne != null)
            memberOne.Dispose();
        if (memberTwo != null)
            memberTwo.Dispose();
    }
}
```

There is one problem with the code above: It doesn't behave well if, for some reason, Dispose() throws an exception. You are calling Dispose() inside a catch block. That catch block executes in the context of a live exception. If Dispose() throws, the first exception is lost and replaced by the second exception. Callers will not see the original exception, which will make it much harder to diagnose the original

problem. Note that a proper Dispose() implementation should not exit by throwing an exception, see Framework Design Guidelines). This is a small concern, but it is real. In order to fix it, you need to wrap the call to Dispose() in its own try block:

```
public class OwnerOfDisposableThings : IDisposable
{
   private DisposableThing memberOne;
   private DisposableThing memberTwo;

   public OwnerOfDisposableThings()
   {
      try
      {
         memberOne = DisposableThing.GetFirst();
         memberTwo = DisposableThing.GetSecond();
      }
      catch (Exception)
      {
         try { Dispose(); } catch (Exception){}
         throw;
      }
   }

   public void DoThings() { }

   public void Dispose()
   {
      if (memberOne != null)
         memberOne.Dispose();
      if (memberTwo != null)
```

```
        memberTwo.Dispose();
    }
}
```

This is one of very few places where I can recommend writing a catch clause for the base Exception class that eats any exceptions.

Once again, let's look at this solution from the standpoint of our stated goals.

On the subject of leaking resources, it works fine. If there are any problems with resource allocation, all other resources are cleaned up. As with the first solution, this object will cleanup properly when used with a using statement.

This version is better for multi-threaded programs. The resource allocation is in the constructor, and therefore resource allocation can't happen concurrently. That makes this version a bit simpler. Note that resource allocation can't happen late, which may mean you are allocating scarce resources before they are needed.

The third goal of avoiding unnecessary finalizers is met. The OwnerOfDisposableThings does not need a finalizer.

The final goal hasn't changed: The public API is the same as it was.

Conclusion

In all your types, there is a period of time when the objects have been allocated, but are not accessible to your program: during the time constructors execute. That is an important window to examine for resource leaks. Anything that gets allocated during construction cannot be freed using the standard .NET Dispose() pattern. Calling code does not have a reference upon which to call Dispose(). You need to modify your API, or defensively code your constructor to ensure that those issues do not occur for your types. I've shown you two techniques to accomplish that. You can pick which works best for you, based on your desired API design.