C# Disposed in Derived Types

People make lots of references to Microsoft articles about how to properly handle Dispose in derived class types.
But, there is literally no article on the Microsoft that actually shows this use case.
So, here's a collected method, based on a few references, testing, and honing from years observing subtle edge cases.

References

Here are references that was useful:

http://reedcopsey.com/2009/03/28/idisposable-part-1-releasing-unmanaged-resources/

http://reedcopsey.com/2009/03/30/idisposable-part-2-subclass-from-an-idisposable-class/

Dispose Simple Case

First, the simple case of a class that implements IDisposable.

Here's what a minimal class will contain if implementing IDisposable:

```
public class ServiceA : IDisposable
{
    private bool _isdisposed = false;

    public ServiceA() { }

    // // TODO: override finalizer only if 'Dispose(bool disposing)' has code to free unmanaged resources
    // ~ServiceA()
    // {
    //     // Do not change this code. Put cleanup code in 'Dispose(bool disposing)' method
```

```
//     Dispose(disposing: false);
// }

protected virtual void Dispose(bool disposing)
{
    // Do dispose work if we have not already...
    if (!_isdisposed)
    {
        // First time through, or we failed to finish before.
        if (disposing)
        {
            // TODO: dispose managed state (managed objects)
        }

        // Release unmanaged resources...
    }

    // Call the base dispose...
    base.Dispose(disposing);

    // Set our disposed flag...
    _isdisposed = true;
}

/// <summary>
/// Public dispose method.
/// </summary>
public void Dispose()
{
    // Do not change this code. Put cleanup code in 'Dispose(bool disposing)' method
    Dispose(disposing: true);

    GC.SuppressFinalize(this);
}

public void DoSomething()
{
  // Throw an exception if we are already disposed...
  if(this._isdisposed)
    throw new ObjectDisposedException();
```

```
    }
}
```

The above example follows the classic, off the shelf, IDisposable implementation.

It includes several things:

- Derives from IDisposable
  This allows runtime services, like a DI container, to know that instances of ServiceA must be disposed after use.
- Private is disposed flag
  This is set in our protected Dispose method, and observed by our class logic, to prevent instance usage after disposal.
- Commented out Finalizer
  The finalizer (destructor) of our ServiceA class is commented out, as it doesn't currently have any unmanaged resources that need to be released during disposal.
  If we do add logic to our protected Dispose method to release unmanaged resources, we simply uncomment the finalizer, so it will call our protected Dispose(false) method.
- Protected Dispose method
  This is the protected Dispose method of our class.
  We add logic to this method to release all of the managed and unmanaged resources that our type holds.
  This method will only perform actions once, since it checks our _isdisposed flag on entry, and sets the flag on exit.
  NOTE: This protected Dispose method is virtual, so it can be overridden in derived class types, to add additional disposal logic.
- Public Dispose Method
  This is the public dispose method for our class and ALL derived types.
  There is no need to modify this method, nor to derive from it.
  All resource releasing and teardown logic should go in the protected Dispose method.
  All derived types will call this same public Dispose method on the base class.
  One exception to this rule, is that you may want to add some logging to the public Dispose method, so that you can have a diagnostic timestamp of when Dispose was called.

Deriving from IDisposable Types

The previous example is a simple class that implements IDisposable.
And, it works for lots of use cases.

But as we mentioned at the beginning of this article, the documentation for a proper cascade Dispose pattern is scarce.

Here’s how to handle Dispose in different scenarios of derived types.

Derived Type with Nothing to Release

There is a fallacy for the simple derived type, that you don’t require any additional Dispose handling or overrides. But, this is not true, without breaking encapsulation, and creating a state safety issue.

Most articles will indicate that this scenario (derived type with nothing to release) doesn’t require any additional dispose logic.
But, they forget that the Dispose pattern still requires for a class’s method to reject execution by throwing an ObjectDisposedException.

And, throwing this exception is not possible in a derived type, without access to the _isdisposed flag of the base class, which is private.
So, any methods in your derived type cannot see the state of that flag (of the base), and throw exceptions as required by the Dispose pattern.

Now. You could choose to simply make the _isdisposed flag of your base class as protected, to allow visibility, and fix this.
But, that creates other problems that your base class no longer has positive control over its own state, because that the state safety of the base class is in questions.

So, to properly derive from a disposable class type, and still be able to throw ObjectDisposedExceptions, you must as well, override the protected Dispose call, and set your own private _isdisposed flag.

What this means is that for derived types with nothing to release, must follow the same pattern as derived types with resources to release.

We will show it, next.

Derived Type with Stuff to Release

If your derived type includes resources that need released, or your derived type has a need to know when it’s been disposed, both of these scenarios will follow the pattern below.

This example is a class, ServiceB, that derives from a disposable type, ServiceA.

Here’s what that minimal derived type looks like:

```
/// Derives from ServiceA, and chains the Dispose methods.
public class ServiceB : ServiceA, IDisposable
{
    // Has its own disposed flag...
    private bool _isdisposed = false;

    override protected void Dispose(bool disposing)
    {
        // Do dispose work if we have not already...
        if (!_isdisposed)
        {
            // First time through, or we failed to finish before.
            if (disposing)
            {
                // TODO: dispose managed state (managed objects)
            }

            // Free any resources of our derived type...
        }

        // Call the dispose on the base class...
        base.Dispose(disposing);

        // Set our own is disposed flag...
        _isdisposed = true;
    }

    public void DoSomethingElse()
    {
      // Throw an exception if we are already disposed...
      if(this._isdisposed)
        throw new ObjectDisposedException();
    }
}
```

The above implementation includes our own is disposed flag for the derived type.

And, it overrides the protected Dispose method, in order to check and set its own is disposed flag, and to release any resources that the derived type holds.

If your derived type includes unmanaged resources, you will need to add a finalizer (destructor) that calls Dispose(false)…. if your base class does not already include this.

And, the way that the dispose pattern is implemented, each class’s dispose logic will only run once, if its is disposed flag is already set.
So, there is no concern for double calls to Dispose when multiple finalizers are called for the same instance.

Revision #1
Created 30 April 2025 02:05:43 by glwhite
Updated 30 April 2025 02:07:44 by glwhite