Item 18: Implement the Standard Dispose Pattern

We've discussed the importance of disposing of objects that hold unmanaged resources. Now it's time to cover how to write your own resource-management code when you create types that contain resources other than memory. A standard pattern is used throughout the .NET Framework for disposing of nonmemory resources. The users of your type will expect you to follow this standard pattern. The standard dispose idiom frees your unmanaged resources using the `IDisposable` interface when clients remember, and it uses the finalizer defensively when clients forget. It works with the Garbage Collector to ensure that your objects pay the performance penalty associated with finalizers only when necessary. This is the right way to handle unmanaged resources, so it pays to understand it thoroughly.

The root base class in the class hierarchy should implement the `IDisposable` interface to free resources. This type should also add a finalizer as a defensive mechanism. Both of these routines delegate the work of freeing resources to a virtual method that derived classes can override for their own resource-management needs. The derived classes need override the virtual method only when the derived class must free its own resources and it must remember to call the base class version of the function.

To begin, your class must have a finalizer if it uses nonmemory resources. You should not rely on clients to always call the `Dispose()` method. You'll leak resources when they forget. It's their fault for not calling `Dispose`, but you'll get the blame. The only way you can guarantee that nonmemory resources get freed properly is to create a finalizer. So create one.

When the Garbage Collector runs, it immediately removes from memory any garbage objects that do not have finalizers. All objects that have finalizers remain in memory. These objects are added to a finalization queue, and the Garbage Collector spawns a new thread to run the finalizers on those objects. After the finalizer thread has finished its work, the garbage objects can be removed from memory. Objects that need finalization stay in memory for far longer than objects without a finalizer. But you have no choice. If you're going to be defensive, you must write a finalizer when your type holds unmanaged resources. But don't worry about performance just yet. The next steps ensure that it's easier for clients to avoid the performance penalty associated with finalization.

Implementing `IDisposable` is the standard way to inform users and the runtime system that your objects hold resources that must be released in a timely manner. The `IDisposable` interface contains just one method:

```
public interface IDisposable
{
  void Dispose( );
}
```

The implementation of your `IDisposable.Dispose()` method is responsible for four tasks:

1. Freeing all unmanaged resources.
2. Freeing all managed resources (this includes unhooking events).
3. Setting a state flag to indicate that the object has been disposed. You need to check this state and throw `ObjectDisposed` exceptions in your public methods, if any get called after disposing of an object.
4. Suppressing finalization. You call `GC.SuppressFinalize(this)` to accomplish this task.

You accomplish two things by implementing `IDisposable`: You provide the mechanism for clients to release all managed resources that you hold in a timely fashion, and you give clients a standard way to release all unmanaged resources. That's quite an improvement. After you've implemented `IDisposable` in your type, clients can avoid the finalization cost. Your class is a reasonably well-behaved member of the .NET community.

But there are still holes in the mechanism you've created. How does a derived class clean up its resources and still let a base class clean up as well? If derived classes override `finalize` or add their own implementation of `IDisposable`, those methods must call the base class; otherwise, the base class doesn't clean up properly. Also, `finalize` and `Dispose` share some of the same responsibilities: You have almost certainly duplicated code between the `finalize` method and the `Dispose` method. As you'll learn in Item 26, overriding interface functions does not work the way you'd expect. The third method in the standard `Dispose` pattern, a protected virtual helper function, factors out these common tasks and adds a hook for derived classes to free resources they allocate. The base

class contains the code for the core interface. The virtual function provides the hook for derived classes to clean up resources in response to `Dispose()` or `finalization`:

```
protected virtual void Dispose( bool isDisposing );
```

This overloaded method does the work necessary to support both `finalize` and `Dispose`, and because it is virtual, it provides an entry point for all derived classes. Derived classes can override this method, provide the proper implementation to clean up their resources, and call the base class version. You clean up managed and unmanaged resources when `isDisposing` is `true`; clean up only unmanaged resources when `isDisposing` is `false`. In both cases, call the base class's `Dispose(bool)` method to let it clean up its own resources.

Here is a short sample that shows the framework of code you supply when you implement this pattern. The `MyResourceHog` class shows the code to implement `IDisposable`, a finalizer, and create the virtual `Dispose` method:

```
public class MyResourceHog : IDisposable
{
  // Flag for already disposed
  private bool _alreadyDisposed = false;

  // finalizer:
  // Call the virtual Dispose method.
  ~MyResourceHog()
  {
    Dispose( false );
  }

  // Implementation of IDisposable.
  // Call the virtual Dispose method.
  // Suppress Finalization.
  public void Dispose()
  {
    Dispose( true );
    GC.SuppressFinalize( true );
  }
```

```
  // Virtual Dispose method
  protected virtual void Dispose( bool isDisposing )
  {
    // Don't dispose more than once.
    if ( _alreadyDisposed )
      return;
    if ( isDisposing )
    {
      // TODO: free managed resources here.
    }
    // TODO: free unmanaged resources here.
    // Set disposed flag:
    _alreadyDisposed = true;
  }
}
```

If a derived class needs to perform additional cleanup, it implements the protected `Dispose` method:

```
public class DerivedResourceHog : MyResourceHog
{
  // Have its own disposed flag.
  private bool _disposed = false;

  protected override void Dispose( bool isDisposing )
  {
    // Don't dispose more than once.
    if ( _disposed )
      return;
    if ( isDisposing )
    {
      // TODO: free managed resources here.
    }
    // TODO: free unmanaged resources here.

    // Let the base class free its resources.
    // Base class is responsible for calling
    // GC.SuppressFinalize( )
    base.Dispose( isDisposing );
```

```
    // Set derived class disposed flag:
    _disposed = true;
  }
}
```

Notice that both the base class and the derived class contain a flag for the disposed state of the object. This is purely defensive. Duplicating the flag encapsulates any possible mistakes made while disposing of an object to only the one type, not all types that make up an object.

You need to write `Dispose` and `finalize` defensively. Disposing of objects can happen in any order. You will encounter cases in which one of the member objects in your type is already disposed of before your `Dispose()` method gets called. You should not view that as a problem because the `Dispose()` method can be called multiple times. If it's called on an object that has already been disposed of, it does nothing. Finalizers have similar rules. Any object that you reference is still in memory, so you don't need to check null references. However, any object that you reference might be disposed of. It might also have already been finalized.

This brings me to the most important recommendation for any method associated with disposal or cleanup: You should be releasing resources only. Do not perform any other processing during a dispose method. You can introduce serious complications to object lifetimes by performing other processing in your `Dispose` or `finalize` methods. Objects are born when you construct them, and they die when the Garbage Collector reclaims them. You can consider them comatose when your program can no longer access them. If you can't reach an object, you can't call any of its methods. For all intents and purposes, it is dead. But objects that have finalizers get to breathe a last breath before they are declared dead. Finalizers should do nothing but clean up unmanaged resources. If a finalizer somehow makes an object reachable again, it has been *resurrected*. It's alive and not well, even though it has awoken from a comatose state. Here's an obvious example:

```
public class BadClass
{
  // Store a reference to a global object:
  private readonly ArrayList _finalizedList;
  private string _msg;
```

```
  public BadClass( ArrayList badList, string msg )
  {
    // cache the reference:
    _finalizedList = badList;
    _msg = (string)msg.Clone();
  }

  ~BadClass()
  {
    // Add this object to the list.
    // This object is reachable, no
    // longer garbage. It's Back!
    _finalizedList.Add( this );
  }
}
```

When a `BadClass` object executes its finalizer, it puts a reference to itself on a global list. It has just made itself reachable. It's alive again! The number of problems you've just introduced will make anyone cringe. The object has been finalized, so the Garbage Collector now believes there is no need to call its finalizer again. If you actually need to finalize a resurrected object, it won't happen. Second, some of your resources might not be available. The GC will not remove from memory any objects that are reachable only by objects in the finalizer queue, but it might have already finalized them. If so, they are almost certainly no longer usable. Although the members that `BadClass` owns are still in memory, they will have likely been disposed of or finalized. There is no way in the language that you can control the order of finalization. You cannot make this kind of construct work reliably. Don't try.

I've never seen code that has resurrected objects in such an obvious fashion, except as an academic exercise. But I have seen code in which the finalizer attempts to do some real work and ends up bringing itself back to life when some function that the finalizer calls saves a reference to the object. The moral is to look very carefully at any code in a finalizer and, by extension, both `Dispose` methods. If that code is doing anything other than releasing resources, look again. Those actions likely will cause bugs in your program in the future. Remove those actions, and make sure that finalizers and `Dispose()` methods release resources and do nothing else.

In a managed environment, you do not need to write a finalizer for every type you create; you do it only for types that store unmanaged types or when your type contains members that implement `IDisposable`. Even if you need only the `Disposable` interface, not a finalizer, implement the entire pattern. Otherwise, you limit your derived classes by complicating their implementation of the standard `Dispose` idiom. Follow the standard `Dispose` idiom I've described. That will make life easier for you, for the users of your class, and for those who create derived classes from your types.