

Paradigm for multiple IDisposables

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2006-09/msg03782.html>

- *From:* "Zach" <divisortheory@xxxxxxxxxx>
 - *Date:* 22 Sep 2006 12:06:51 -0700
-

I'm sure this comes up often, but I have a situation where I have at least 4 objects that all implement IDisposable. It gets very tedious having to write finally blocks that Dispose all of them in a row, but the using statement is very limited in that it doesn't (seem to) allow me to have a list of objects specified. Well, I guess that's not entirely true, if they are all the same type it does, but in my case they aren't all the same type. I've figured out a couple ways of handling, but I wonder if there is something more elegant than what I'm doing. Here's what I've thought of so far:

Solution 1: Initialize everything outside of the using block, then use a list of IDisposables inside the using statement, and the compiler will implicitly convert each one to an IDisposable, and since they will all be the same type after implicit conversion, the syntax is allowed.

Example 1:

```
Disposable1 First = new Disposable1();
Disposable2 Second = new Disposable2();
using (IDisposable dummy1 = First, dummy2 = Second)
{
}
}
```

Solution 2: Write a new class (MultiDispose), which itself implements IDisposable and contains a list of classes which implement IDisposable, and receives a list of items in its constructor. In MultiDispose.Dispose, run through the list and Dispose() each item in the list.

Example 2:

```
class MultiDispose : IDisposable
{
private List<IDisposable> _DisposeList;
public MultiDispose(params IDisposable[] DisposeList)
{
_DisposeList = new List<IDisposable>(DisposeList);
}
```

Paradigm for multiple IDisposables

```
}  
  
public void Dispose()  
{  
    foreach (IDisposable DisposeItem in _DisposeList)  
        DisposeItem.Dispose();  
}  
}  
  
//Some other function  
using (MultiDispose DisposeScope = new MultiDispose(First, Second,  
Third, Fourth))  
{  
  
}
```

Solution 3: Do nothing special, just use tons of nested using statements. To improve readability, wrap only the innermost using block with parentheses, and don't indent consecutive using blocks so as to keep the indentation level small.

Example 3:

```
using (Disposable1 First = new Disposable1())  
using (Disposable2 Second = new Disposable2())  
using (Disposable3 Third = new Disposable3())  
using (Disposable4 Fourth = new Disposable4())  
{  
    //Code here  
}
```

So far I prefer Solution #2 above, because it has the nice benefit of allowing me to add a method to MultiDispose called Add(), so I can add a new item to the list in the middle of the block. This provides an elegant way to Dispose of items which aren't valid unless certain operations (which could throw) have already been performed on one of the items declared in the using block. An example with Database code:

```
SqlConnection Connection = new SqlConnection(ConnectionString);  
SqlCommand Command = new SqlCommand(CommandText, Connection);  
using (MultiDispose DisposeScope = new MultiDispose(Connection,  
Command))  
{  
    Connection.Open();  
    SqlDataReader Reader =  
    Command.ExecuteReader(CommandBehavior.CloseConnection);  
  
    DisposeScope.Add(Reader);  
    if (!Reader.HasRows)
```

Paradigm for multiple IDisposables

```
return false;  
//Process the data in the rows.  
}
```

The alternative in this situation would have been to make ANOTHER using block just for the Reader, which would start to hinder readability, and possibly performance.

I admit I don't know too much about CLR internals or some of the intricacies of C# language syntax, so I'm having a hard time analyzing all the pros and cons of each method. For example, Solution 3 creates a nesting of try/finally blocks N levels deep, where N is the number of items you're "using". Perhaps this could have some performance implications. Solution 2 (my preferred solution so far) could have some pitfalls in that you're circumventing some of the typical checks the compiler performs for you, which may make it easier to introduce subtle programming errors.

Anyone have any thoughts on this problem, or is there a standard "accepted" way of handling this?

Thanks

.