Memory Management C# Tutorial

In this exercise you will learn more about the C# garbage collector, and implement the IDisposable interface.

The Garbage Collector:

.NET automatically manages memory for us during a program's life. Allocations are made onto the heap with the **new** keyword, but unlike C++, the memory will automatically be released when a reference goes out of scope, thus we don't need a corresponding **delete** to be called. This can make our life much easier in C#, but don't be fooled – we still need to clean up after ourselves.

For example, the following code opens up a StreamWriter object, and writes some text to it. If a call to StreamWriter.Close() is not made the file handle is still in use even though the original StreamWriter object has gone out of scope and we no longer have access to it!

```
class Program
{
    static void Main(string[] args)
    {
        BadFileIOExample();
        BadFileIOExample();

        while (true) ; //this just keeps our console window open
    }

    static void BadFileIOExample()
    {
        StreamWriter stream = new StreamWriter("test.txt");

        stream.WriteLine("Hello");
    }
}
```

This code will cause a System.IOException to be called when the method is called a second time.

IDisposable:

We cannot choose when the Garbage Collector will decide to clean up objects. Before an object's memory is reclaimed, the Finalize() method on that object will be called. This is not tied to the lifetime of an object however, making it difficult to know when Finalize() is actually going to be called. Thus, if you are holding on to an expensive resource (such as a file handle) you can't be sure of when it is actually going to be released. This is known as non-deterministic finalization.

Luckily, C# provides us with an interface that we can implement, IDisposable, which provides a method, Dispose() for explicit clean up. We will now take a look at how to implement the IDisposable interface in our code.

We are going to add a small test class to see how the IDisposable pattern works. Add the following class to your program:

```
class MyTestClass : IDisposable
{
    public MyTestClass()
    {
        Console.WriteLine("Constructing");
    }

    public void Dispose()
    {
        Console.WriteLine("Disposing");
    }

    ~MyTestClass()
    {
        Console.WriteLine("Finalizing");
    }
}
```

And in your main function:

```
static void Main(string[] args)
{
    MyTestClass myTestClass = new MyTestClass();

    Console.ReadLine();

    myTestClass.Dispose();

    Console.ReadLine();
}
```

The Dispose() method is meant to fulfil the same purpose as a delete call in C++. Notice how the actual Destructor is called after the program exits. (You should see “Finalizing” pop up for a brief moment before the console window closes).

To ensure consistent behaviour between implementations, IDisposable has a few rules that should be followed:

1. **The method should be able to be called more than once without consequence**
2. **The object should implement a finalize method, which calls Dispose in the event that Dispose was not explicitly called**
3. **Dispose should called GC.SuppressFinalize to prevent the GC from calling Finalize**
4. **Methods called after Dispose should throw an ObjectDisposedException if that code relies on disposed data**

Let’s add these rules now to make our implementation more robust. Add the following code to your program (changes are in **bold**):

```
class MyTestClass : IDisposable
{
    private bool isDisposed = false;

    public MyTestClass()
    {
        Console.WriteLine("Constructing");
    }

    public void DoSomething()
    {
        if (isDisposed)
        {
            throw new ObjectDisposedException("MyTestClass");
        }

        //Use resources if they haven't been disposed
    }

    public void Dispose()
    {
        if (isDisposed)
        {
            isDisposed = true;
            Console.WriteLine("Disposing");
            GC.SuppressFinalize(this);

            //Dispose of resources here
        }
        else
        {
            throw new ObjectDisposedException("MyTestClass", "Already disposed");
        }
    }

    ~MyTestClass()
    {
        Console.WriteLine("Finalizing");
        this.Dispose();
    }
}
```

We use a flag to check whether or not our object has been disposed of. The method DoSomething() checks to see if the object has been disposed of first and throws an exception if it has already been disposed to avoid using un-managed memory that has already been deleted if dispose was called earlier.

Now we have a good, simple implementation of the IDisposable interface.