System.IDisposable Interface

```
[ILAsm]
.class interface public abstract IDisposable

[C#]
public interface IDisposable
```

Assembly Info:

- *Name:* mscorlib
- *Public Key:* [00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00]
- *Version:* 2.0.x.x
- *Attributes:*
  - CLSCompliantAttribute(true)

Summary

Implemented by classes that require explicit control over resource cleanup.

Library: BCL

Description

Objects that need to free resources that cannot safely be reclaimed by the garbage collector implement the `System.IDisposable` interface.

It is a version breaking change to add the `System.IDisposable` interface to an existing class, as it changes the semantics of the class.

[*Note:* `System.IDisposable` contains the `System.IDisposable.Dispose` method. The consumer of an object should call this method when the object is no longer needed. The `System.IDisposable` interface is generally provided for the release of unmanaged resources that need to be reclaimed in some order or time dependent manner. It is important to note that the actual release of these resources happens at the first call to `System.IDisposable.Dispose` for any given object that supports this interface. Programmers should take care to pair the creation of objects that implement `IDisposable` with at most one invocation of the `Dispose` method. Though it is legal to invoke `Dispose` more than once, if this happens it might indicate the presence of a bug since such an object is usually rendered otherwise unusable after the first `Dispose` invocation.]

IDisposable.Dispose() Method

```
[ILAsm]
.method public hidebysig virtual abstract void Dispose()

[C#]
void Dispose()
```

Summary

Performs application-defined tasks associated with freeing or resetting resources.

Description

[*Note:* This method is, by convention, used for all tasks associated with freeing resources held by an object, or preparing an object for reuse.

When implementing the `System.IDisposable.Dispose` method, objects should seek to ensure that all held resources are freed by propagating the call through the containment hierarchy. For example, if an object A allocates an object B, and B allocates an object C, then A's `System.IDisposable.Dispose` implementation should call `System.IDisposable.Dispose` on B, which should in turn call `System.IDisposable.Dispose` on C. Objects should also call the `System.IDisposable.Dispose` method of their base class if the base class implements `System.IDisposable`.

If an object's `System.IDisposable.Dispose` method is called more than once, the object should ignore all calls after the first one. The object should not throw an exception if its `System.IDisposable.Dispose` method is called multiple times. `System.IDisposable.Dispose` can throw an exception if an error occurs because a resource has already been freed and `System.IDisposable.Dispose` had not been called previously.

A resource type might use a particular convention to denote an allocated state versus a freed state. An example of this is stream classes, which are traditionally thought of as open or closed. Classes that have such conventions might choose to implement a public method with a customized name, which calls the `System.IDisposable.Dispose` method.

Because the `System.IDisposable.Dispose` method must be called explicitly, objects that implement `System.IDisposable` should also implement a finalizer to handle freeing resources when `System.IDisposable.Dispose` is not called. By default, the garbage collector will automatically call an object's finalizer prior to reclaiming its memory. However, once the `System.IDisposable.Dispose` method has been called, it is typically unnecessary and/or undesirable for the garbage collector to call the disposed object's finalizer. To prevent automatic finalization, `System.IDisposable.Dispose` implementations can call `System.GC.SuppressFinalize`. For additional information on implementing finalizers, see `System.GC` and `System.Object.Finalize`.

]

Example

Resource classes should follow the pattern illustrated by this example:

[C#]

```
class ResourceWrapper: BaseType, IDisposable {
  // Pointer to a external resource.
  private int handle;
  private OtherResource otherRes; //Other resource you use.
  private bool disposed = false;

  public ResourceWrapper () {
    handle = //Allocate on the unmanaged side.
    otherRes = new OtherResource (...);
  }
  // Free your own state.
  private void freeState () {
    if (!disposed) {
       CloseHandle (handle);
       dispose = true;
    }
  }

  // Free your own state, call dispose on all state you hold,
  // and take yourself off the Finalization queue.
  public void Dispose () {
    freeState ();
    OtherRes.Dispose();
    // If base type implements dispose, call it.
    base.Dispose();
    GC.SuppressFinalize(this);
  }

  // Free your own state (not other state you hold)
  // and give your base class a chance to finalize.
  ~ResourceWrapper (){
     freeState();
  }
}
```