Cleaning Up Unmanaged Resources

For the majority of the objects that your app creates, you can rely on the .NET garbage collector to handle memory management. However, when you create objects that include unmanaged resources, you must explicitly release those resources when you finish using them. The most common types of unmanaged resources are objects that wrap operating system resources, such as files, windows, network connections, or database connections. Although the garbage collector is able to track the lifetime of an object that encapsulates an unmanaged resource, it doesn't know how to release and clean up the unmanaged resource.

If your types use unmanaged resources, you should do the following:

- Implement the dispose pattern. This requires that you provide an `IDisposable.Dispose` implementation to enable the deterministic release of unmanaged resources. A consumer of your type calls Dispose when the object (and the resources it uses) are no longer needed. The Dispose method immediately releases the unmanaged resources.
- In the event that a consumer of your type forgets to call Dispose, provide a way for your unmanaged resources to be released. There are two ways to do this:
  1. Use a safe handle to wrap your unmanaged resource. This is the recommended technique. Safe handles are derived from the `System.Runtime.InteropServices.SafeHandle` abstract class and include a robust Finalize method. When you use a safe handle, you simply implement the `IDisposable` interface and call your safe handle's Dispose method in your `IDisposable.Dispose` implementation. The safe handle's finalizer is called automatically by the garbage collector if its Dispose method is not called.
  2. Override the `Object.Finalize` method. Finalization enables the non-deterministic release of unmanaged resources when the consumer of a type fails to call `IDisposable.Dispose` to dispose of them deterministically. Define a finalizer by overriding the `Object.Finalize` method.

Note: However, because object finalization can be a complex and an error-prone operation, we recommend that you use a safe handle instead of providing your own finalizer.

Object.Finalize Method

This method allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.

Example:

```
using System;
using System.Diagnostics;
public class ExampleClass
{
   Stopwatch sw;
   public ExampleClass()
   {
```

```
      sw = Stopwatch.StartNew();
      Console.WriteLine("Instantiated object");
   }
   public void ShowDuration()
   {
      Console.WriteLine("This instance of {0} has been in existence for {1}",
                        this, sw.Elapsed);
   }
   ~ExampleClass()
   {
      Console.WriteLine("Finalizing object");
      sw.Stop();
      Console.WriteLine("This instance of {0} has been in existence for {1}",
                        this, sw.Elapsed);
   }
}
public class Demo
{
   public static void Main()
   {
      ExampleClass ex = new ExampleClass();
      ex.ShowDuration();
   }
}
// The example displays output like the following:
//    Instantiated object
//    This instance of ExampleClass has been in existence for 00:00:00.0011060
//    Finalizing object
//    This instance of ExampleClass has been in existence for 00:00:00.0036294
```

How object finalization works

The Object class provides no implementation for the Finalize method, and the garbage collector does not mark types derived from Object for finalization unless they override the Finalize method.

If a type does override the Finalize method, the garbage collector adds an entry for each instance of the type to an internal structure called the finalization queue. The finalization queue contains entries for all the objects in the managed heap whose finalization code must run before the garbage collector can reclaim their memory. The garbage collector then calls the Finalize method automatically under the following conditions:

- After the garbage collector has discovered that an object is inaccessible, unless the object has been exempted from finalization by a call to the `GC.SuppressFinalize` method.
- On .NET Framework only, during shutdown of an application domain, unless the object is exempt from finalization. During shutdown, even objects that are still accessible are finalized.

Finalize is automatically called only once on a given instance, unless the object is re-registered by using a mechanism such as `GC.ReRegisterForFinalize` and the `GC.SuppressFinalize` method has not been subsequently called.

Finalize operations have the following limitations:

- The exact time when the finalizer executes is undefined. To ensure deterministic release of resources for instances of your class, implement a Close method or provide a `IDisposable.Dispose` implementation.
- The finalizers of two objects are not guaranteed to run in any specific order, even if one object refers to the other. That is, if Object A has a reference to Object B and both have finalizers, Object B might have already been finalized when the finalizer of Object A starts.
- The thread on which the finalizer runs is unspecified.

The Finalize method might not run to completion or might not run at all under the following exceptional circumstances:

- If another finalizer blocks indefinitely (goes into an infinite loop, tries to obtain a lock it can never obtain, and so on). Because the runtime tries to run finalizers to completion, other finalizers might not be called if a finalizer blocks indefinitely.
- If the process terminates without giving the runtime a chance to clean up. In this case, the runtime's first notification of process termination is a DLL_PROCESS_DETACH notification.

The runtime continues to finalize objects during shutdown only while the number of finalizable objects continues to decrease.

If Finalize or an override of Finalize throws an exception, and the runtime is not hosted by an application that overrides the default policy, the runtime terminates the process and no active try/finally blocks or finalizers are executed. This behavior ensures process integrity if the finalizer cannot free or destroy resources.

Note: The C# compiler does not allow you to override the Finalize method. Instead, you provide a finalizer by implementing a destructor for your class. A C# destructor automatically calls the destructor of its base class.