

IDisposable.Dispose Method

Namespace: [System](#)

Assembly: System.Runtime.dll

In this article

[Definition](#)


[Examples](#)

[Remarks](#)

[Applies to](#)


[See also](#)

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

C#	 Copy
<pre>public void Dispose ();</pre>	

Examples

The following example shows how you can implement the [Dispose](#) method.

C#	 Copy
<pre>using System; using System.ComponentModel; // The following example demonstrates how to create // a resource class that implements the IDisposable interface // and the IDisposable.Dispose method. public class DisposeExample { // A base class that implements IDisposable. // By implementing IDisposable, you are announcing that // instances of this type allocate scarce resources. public class MyResource: IDisposable {</pre>	

```

// Pointer to an external unmanaged resource.
private IntPtr handle;
// Other managed resource this class uses.
private Component component = new Component();
// Track whether Dispose has been called.
private bool disposed = false;

// The class constructor.
public MyResource(IntPtr handle)
{
    this.handle = handle;
}

// Implement IDisposable.
// Do not make this method virtual.
// A derived class should not be able to override this method.
public void Dispose()
{
    Dispose(true);
    // This object will be cleaned up by the Dispose method.
    // Therefore, you should call GC.SuppressFinalize to
    // take this object off the finalization queue
    // and prevent finalization code for this object
    // from executing a second time.
    GC.SuppressFinalize(this);
}

// Dispose(bool disposing) executes in two distinct scenarios.
// If disposing equals true, the method has been called di-
rectly
// or indirectly by a user's code. Managed and unmanaged re-
sources
// can be disposed.
// If disposing equals false, the method has been called by the
// runtime from inside the finalizer and you should not refer-
ence
// other objects. Only unmanaged resources can be disposed.
protected virtual void Dispose(bool disposing)
{
    // Check to see if Dispose has already been called.
    if(!this.disposed)
    {
        // If disposing equals true, dispose all managed
        // and unmanaged resources.
        if(disposing)
        {
            // Dispose managed resources.
            component.Dispose();
        }

        // Call the appropriate methods to clean up

```

```

        // unmanaged resources here.
        // If disposing is false,
        // only the following code is executed.
        CloseHandle(handle);
        handle = IntPtr.Zero;

        // Note disposing has been done.
        disposed = true;
    }
}

// Use interop to call the method necessary
// to clean up the unmanaged resource.
[System.Runtime.InteropServices.DllImport("Kernel32")]
private extern static Boolean CloseHandle(IntPtr handle);

// Use C# destructor syntax for finalization code.
// This destructor will run only if the Dispose method
// does not get called.
// It gives your base class the opportunity to finalize.
// Do not provide destructors in types derived from this class.
~MyResource()
{
    // Do not re-create Dispose clean-up code here.
    // Calling Dispose(false) is optimal in terms of
    // readability and maintainability.
    Dispose(false);
}
}

public static void Main()
{
    // Insert code here to create
    // and use the MyResource object.
}
}

```

Remarks

Use this method to close or release unmanaged resources such as files, streams, and handles held by an instance of the class that implements this interface. By convention, this method is used for all tasks associated with freeing resources held by an object, or preparing an object for reuse.

Warning

If you are using a class that implements the **IDisposable** interface, you should call its **Dispose** implementation when you are finished using the class. For more information, see the "Using an object that implements IDisposable" section in the **IDisposable** topic.

When implementing this method, 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 object B allocates an object C, then A's **Dispose** implementation must call **Dispose** on B, which must in turn call **Dispose** on C.

i Important

The C++ compiler supports deterministic disposal of resources and does not allow direct implementation of the **Dispose** method.

An object must also call the **Dispose** method of its base class if the base class implements **IDisposable**. For more information about implementing **IDisposable** on a base class and its subclasses, see the "IDisposable and the inheritance hierarchy" section in the **IDisposable** topic.

If an object's **Dispose** method is called more than once, the object must ignore all calls after the first one. The object must not throw an exception if its **Dispose** method is called multiple times. Instance methods other than **Dispose** can throw an **ObjectDisposedException** when resources are already disposed.

Users might expect a resource type to 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. The implementer of a class that has such a convention might choose to implement a public method with a customized name, such as `Close`, that calls the **Dispose** method.

Because the **Dispose** method must be called explicitly, there is always a danger that the unmanaged resources will not be released, because the consumer of an object fails to call its **Dispose** method. There are two ways to avoid this:

- Wrap the managed resource in an object derived from **System.Runtime.InteropServices.SafeHandle**. Your **Dispose** implementation then calls the **Dispose** method of the **System.Runtime.InteropServices.SafeHandle** instances. For more information, see "The SafeHandle alternative" section in the **Object.Finalize** topic.

- Implement a finalizer to free resources when [Dispose](#) is not called. By default, the garbage collector automatically calls an object's finalizer before reclaiming its memory. However, if the [Dispose](#) method has been called, it is typically unnecessary for the garbage collector to call the disposed object's finalizer. To prevent automatic finalization, [Dispose](#) implementations can call the [GC.SuppressFinalize](#) method.

When you use an object that accesses unmanaged resources, such as a [StreamWriter](#), a good practice is to create the instance with a `using` statement. The `using` statement automatically closes the stream and calls [Dispose](#) on the object when the code that is using it has completed. For an example, see the [StreamWriter](#) class.

Applies to

.NET

5.0 RC1

.NET Core

3.1, 3.0, 2.2, 2.1, 2.0, 1.1, 1.0

.NET Framework

4.8, 4.7.2, 4.7.1, 4.7, 4.6.2, 4.6.1, 4.6, 4.5.2, 4.5.1, 4.5, 4.0, 3.5, 3.0, 2.0, 1.1

.NET Standard

2.1, 2.0, 1.6, 1.5, 1.4, 1.3, 1.2, 1.1, 1.0

UWP

10.0

Xamarin.Android

7.1

Xamarin.iOS

10.8

Xamarin.Mac

3.0

See also

- [Implementing a Dispose Method](#)

Is this page helpful?

 Yes  No
