.NET Framework Class Library

**IDisposable.Dispose Method**

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

**Namespace:**  [System](http://msdn.microsoft.com/en-us/library/system.aspx)  
**Assembly:**  mscorlib (in mscorlib.dll)

http://i.msdn.microsoft.com/Global/Images/clear.gif Syntax

Visual Basic

Sub Dispose

C#

void Dispose()

Visual C++

void Dispose()

F#

abstract Dispose : unit -> unit

http://i.msdn.microsoft.com/Global/Images/clear.gif 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.

|  |
| --- |
| **Important noteImportant** |
| C++ programmers should read [Destructors and Finalizers in Visual C++](http://msdn.microsoft.com/en-us/library/ms177197.aspx). In the .NET Framework version 2.0, the C++ compiler provides support for implementing deterministic disposal of resources and does not allow direct implementation of the **Dispose** method. |

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. An object must also call the **Dispose** method of its base class if the base class implements [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable.aspx).

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](http://msdn.microsoft.com/en-us/library/system.objectdisposedexception.aspx) 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, objects that implement [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable.aspx) must also implement a finalizer to handle freeing resources when **Dispose** is not called. By default, the garbage collector automatically calls an object's finalizer prior to reclaiming its memory. However, once 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](http://msdn.microsoft.com/en-us/library/system.gc.suppressfinalize.aspx) method.

For more information on implementing finalizers and the **Dispose** method, see the [GC](http://msdn.microsoft.com/en-us/library/system.gc.aspx) class, the [Object..::.Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize.aspx) method, and [Implementing Finalize and Dispose to Clean Up Unmanaged Resources](http://msdn.microsoft.com/en-us/library/b1yfkh5e.aspx).

When you use an object that accesses unmanaged resources, such as a [StreamWriter](http://msdn.microsoft.com/en-us/library/system.io.streamwriter.aspx), 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](http://msdn.microsoft.com/en-us/library/system.io.streamwriter.aspx) class.

http://i.msdn.microsoft.com/Global/Images/clear.gif Examples

The following example shows how you can implement the **Dispose** method.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl00_code');" \o "Copy Code)

Imports System

Imports 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 class that implements IDisposable.

' By implementing IDisposable, you are announcing that

' instances of this type allocate scarce resources.

Public Class MyResource

Implements IDisposable

' Pointer to an external unmanaged resource.

Private handle As IntPtr

' Other managed resource this class uses.

Private component As component

' Track whether Dispose has been called.

Private disposed As Boolean = False

' The class constructor.

Public Sub New(ByVal handle As IntPtr)

Me.handle = handle

End Sub

' Implement IDisposable.

' Do not make this method virtual.

' A derived class should not be able to override this method.

Public Overloads Sub Dispose() Implements IDisposable.Dispose

Dispose(True)

' This object will be cleaned up by the Dispose method.

' Therefore, you should call GC.SupressFinalize to

' take this object off the finalization queue

' and prevent finalization code for this object

' from executing a second time.

GC.SuppressFinalize(Me)

End Sub

' Dispose(bool disposing) executes in two distinct scenarios.

' If disposing equals true, the method has been called directly

' or indirectly by a user's code. Managed and unmanaged resources

' can be disposed.

' If disposing equals false, the method has been called by the

' runtime from inside the finalizer and you should not reference

' other objects. Only unmanaged resources can be disposed.

Protected Overridable Overloads Sub Dispose(ByVal disposing As Boolean)

' Check to see if Dispose has already been called.

If Not Me.disposed Then

' If disposing equals true, dispose all managed

' and unmanaged resources.

If disposing Then

' Dispose managed resources.

component.Dispose()

End If

' 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

End If

End Sub

' Use interop to call the method necessary

' to clean up the unmanaged resource.

<System.Runtime.InteropServices.DllImport("Kernel32")> \_

Private Shared Function CloseHandle(ByVal handle As IntPtr) As [Boolean]

End Function

' This finalizer will run only if the Dispose method

' does not get called.

' It gives your base class the opportunity to finalize.

' Do not provide finalize methods in types derived from this class.

Protected Overrides Sub Finalize()

' Do not re-create Dispose clean-up code here.

' Calling Dispose(false) is optimal in terms of

' readability and maintainability.

Dispose(False)

MyBase.Finalize()

End Sub

End Class

Public Shared Sub Main()

' Insert code here to create

' and use the MyResource object.

End Sub

End Class

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl01_code');" \o "Copy Code)

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

{

// 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.SupressFinalize 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 directly

// or indirectly by a user's code. Managed and unmanaged resources

// can be disposed.

// If disposing equals false, the method has been called by the

// runtime from inside the finalizer and you should not reference

// 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.

}

}

Visual C++

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_ctl00_ctl02_code');" \o "Copy Code)

#using <System.dll>

#using <System.Windows.Forms.dll>

using namespace System;

using namespace System::ComponentModel;

using namespace System::Windows::Forms;

// The following example demonstrates how to create a class that

// implements the IDisposable interface and the IDisposable.Dispose

// method with finalization to clean up unmanaged resources.

//

public ref class MyResource: public IDisposable

{

private:

// Pointer to an external unmanaged resource.

IntPtr handle;

// A managed resource this class uses.

Component^ component;

// Track whether Dispose has been called.

bool disposed;

public:

// The class constructor.

MyResource( IntPtr handle, Component^ component )

{

this->handle = handle;

this->component = component;

disposed = false;

}

// This method is called if the user explicitly disposes of the

// object (by calling the Dispose method in other managed languages,

// or the destructor in C++). The compiler emits as a call to

// GC::SuppressFinalize( this ) for you, so there is no need to

// call it here.

~MyResource()

{

// Dispose of managed resources.

component->~Component();

// Call C++ finalizer to clean up unmanaged resources.

this->!MyResource();

// Mark the class as disposed. This flag allows you to throw an

// exception if a disposed object is accessed.

disposed = true;

}

// Use interop to call the method necessary to clean up the

// unmanaged resource.

//

[System::Runtime::InteropServices::DllImport("Kernel32")]

static Boolean CloseHandle( IntPtr handle );

// The C++ finalizer destructor ensures that unmanaged resources get

// released if the user releases the object without explicitly

// disposing of it.

//

!MyResource()

{

// Call the appropriate methods to clean up unmanaged

// resources here. If disposing is false when Dispose(bool,

// disposing) is called, only the following code is executed.

CloseHandle( handle );

handle = IntPtr::Zero;

}

};

void main()

{

// Insert code here to create and use the MyResource object.

MyResource^ mr = gcnew MyResource((IntPtr) 42, (Component^) gcnew Button());

mr->~MyResource();

}