**Destructors (C# Programming Guide)**

**Visual Studio 2013**

[Other Versions](javascript:;)

Description: http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [Visual Studio 2012](http://msdn.microsoft.com/en-us/library/66x5fx1b(d=printer,v=vs.110).aspx)
* [Visual Studio 2010](http://msdn.microsoft.com/en-us/library/66x5fx1b(d=printer,v=vs.100).aspx)
* [Visual Studio 2008](http://msdn.microsoft.com/en-us/library/66x5fx1b(d=printer,v=vs.90).aspx)
* [Visual Studio 2005](http://msdn.microsoft.com/en-us/library/66x5fx1b(d=printer,v=vs.80).aspx)
* [Visual Studio .NET 2003](http://msdn.microsoft.com/en-us/library/66x5fx1b(d=printer,v=vs.71).aspx)

Destructors are used to destruct instances of classes.

[Remarks](javascript:void(0))

* Destructors cannot be defined in structs. They are only used with classes.
* A class can only have one destructor.
* Destructors cannot be inherited or overloaded.
* Destructors cannot be called. They are invoked automatically.
* A destructor does not take modifiers or have parameters.

For example, the following is a declaration of a destructor for the class Car:

C#

class Car

{

~Car() // destructor

{

// cleanup statements...

}

}

The destructor implicitly calls [Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize.aspx) on the base class of the object. Therefore, the previous destructor code is implicitly translated to the following code:

protected override void Finalize()

{

try

{

// Cleanup statements...

}

finally

{

base.Finalize();

}

}

This means that the Finalize method is called recursively for all instances in the inheritance chain, from the most-derived to the least-derived.

|  |
| --- |
| **Description: NoteNote** |
| Empty destructors should not be used. When a class contains a destructor, an entry is created in the Finalize queue. When the destructor is called, the garbage collector is invoked to process the queue. If the destructor is empty, this just causes a needless loss of performance. |

The programmer has no control over when the destructor is called because this is determined by the garbage collector. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for destruction, it calls the destructor (if any) and reclaims the memory used to store the object. Destructors are also called when the program exits.

It is possible to force garbage collection by calling [Collect](http://msdn.microsoft.com/en-us/library/system.gc.collect.aspx), but most of the time, this should be avoided because it may create performance issues.

[Using Destructors to Release Resources](javascript:void(0))

In general, C# does not require as much memory management as is needed when you develop with a language that does not target a runtime with garbage collection. This is because the .NET Framework garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources such as windows, files, and network connections, you should use destructors to free those resources. When the object is eligible for destruction, the garbage collector runs the Finalize method of the object.

[Explicit Release of Resources](javascript:void(0))

If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. You do this by implementing a Dispose method from the [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable.aspx) interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the destructor becomes a safeguard to clean up resources if the call to the Dispose method failed.

For more details about cleaning up resources, see the following topics:

* [Cleaning Up Unmanaged Resources](http://msdn.microsoft.com/en-us/library/498928w2.aspx)
* [Implementing a Dispose Method](http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx)
* [using Statement (C# Reference)](http://msdn.microsoft.com/en-us/library/yh598w02.aspx)

[Example](javascript:void(0))

The following example creates three classes that make a chain of inheritance. The class First is the base class, Second is derived from First, and Third is derived from Second. All three have destructors. In Main(), an instance of the most-derived class is created. When the program runs, notice that the destructors for the three classes are called automatically, and in order, from the most-derived to the least-derived.

C#

class First

{

~First()

{

System.Diagnostics.Trace.WriteLine("First's destructor is called.");

}

}

class Second : First

{

~Second()

{

System.Diagnostics.Trace.WriteLine("Second's destructor is called.");

}

}

class Third : Second

{

~Third()

{

System.Diagnostics.Trace.WriteLine("Third's destructor is called.");

}

}

class TestDestructors

{

static void Main()

{

Third t = new Third();

}

}

/\* Output (to VS Output Window):

Third's destructor is called.

Second's destructor is called.

First's destructor is called.

\*/