How to dispose a class in .net?

Asked 16 years, 4 months ago Modified 1 year, 1 month ago Viewed 94k times

46

The .NET garbage collector will eventually free up memory, but what if you want that memory back immediately? What code do you need to use in a class `MyClass` to call

```
MyClass.Dispose()
```

and free up all the used space by variables and objects in `MyClass`?

.net memory dispose

Share

Improve this question

Follow

edited Jun 22, 2016 at 15:16

community wiki
8 revs, 7 users 44%
Jon Galloway

20 Answers

Sorted by: Highest score (default)

103

IDisposable has nothing to do with freeing memory. IDisposable is a pattern for freeing *unmanaged* resources -- and memory is quite definitely a managed resource.

The links pointing to GC.Collect() are the correct answer, though use of this function is generally discouraged by the Microsoft .NET documentation.

Edit: Having earned a substantial amount of karma for this answer, I feel a certain responsibility to elaborate on it, lest a newcomer to .NET resource management get the wrong impression.

Inside a .NET process, there are two kinds of resource -- managed and unmanaged. "Managed" means that the runtime is in control of the resource, while "unmanaged" means that it's the programmer's responsibility. And there really is only one kind of managed resource that we care about in .NET today -- memory. The programmer tells the runtime to allocate memory and after that it's up to the runtime to figure out when the memory can freed. The mechanism that .NET uses for this purpose is called garbage collection and you can find plenty of information about GC on the internet simply by using Google.

For the other kinds of resources, .NET doesn't know anything about cleaning them up so it has to rely on the programmer to do the right thing. To this end, the platform

gives the programmer three tools:

1. The IDisposable interface and the "using" statement in VB and C#
2. Finalizers
3. The IDisposable pattern as implemented by many BCL classes

The first of these allows the programmer to efficiently acquire a resource, use it and then release it all within the same method.

```
using (DisposableObject tmp = DisposableObject.AcquireResource()) {
    // Do something with tmp
}
// At this point, tmp.Dispose() will automatically have been called
// BUT, tmp may still a perfectly valid object that still takes up memory
```

If "AcquireResource" is a factory method that (for instance) opens a file and "Dispose" automatically closes the file, then this code cannot leak a file resource. But the memory for the "tmp" object itself may well still be allocated. That's because the IDisposable interface has absolutely no connection to the garbage collector. If you *did* want to ensure that the memory was freed, your only option would be to call `GC.Collect()` to force a garbage collection.

However, it cannot be stressed enough that this is probably not a good idea. It's generally much better to let the garbage collector do what it was designed to do, which is to manage memory.

What happens if the resource is being used for a longer period of time, such that its lifespan crosses several methods? Clearly, the "using" statement is no longer applicable, so the programmer would have to manually call "Dispose" when he or she is done with the resource. And what happens if the programmer forgets? If there's no fallback, then the process or computer may eventually run out of whichever resource isn't being properly freed.

That's where finalizers come in. A finalizer is a method on your class that has a special relationship with the garbage collector. The GC promises that -- before freeing the memory for any object of that type -- it will first give the finalizer a chance to do some kind of cleanup.

So in the case of a file, we theoretically don't need to close the file manually at all. We can just wait until the garbage collector gets to it and then let the finalizer do the work. Unfortunately, this doesn't work well in practice because the garbage collector runs non-deterministically. The file may stay open considerably longer than the programmer expects. And if enough files are kept open, the system may fail when trying to open an additional file.

For most resources, we want both of these things. We want a convention to be able to say "we're done with this resource now" and we want to make sure that there's at least some chance for the cleanup to happen automatically if we forget to do it manually. That's where the "IDisposable" pattern comes into play. This is a convention that allows IDispose and a finalizer to play nicely together. You can see how the pattern works by looking at the official documentation for IDisposable.

Bottom line: If what you really want to do is to just make sure that memory is freed, then IDisposable and finalizers will not help you. But the IDisposable interface is part of an extremely important pattern that all .NET programmers should understand.

Share

Improve this answer

Follow

edited Sep 2, 2008 at 1:44

answered Aug 15, 2008 at 15:40

Curt Hagenlocher

20.9k ● 8 ● 62 ● 50

I just wanted to point out that there's a terminology issue with this question and its answers. You are all talking about freeing/disposing objects; i.e., instances of a class. A class is the type itself, the associated code -procedures, functions, properties, etc.- and shared (static) class data. Releasing memory or other resources from a class itself is much more difficult than an instance's, as it involves unloading the entire AppDomain the class lives into, along with its assembly and other related classes. Of course that's not something one would normally want to do. – Guillermo Prandi Sep 22, 2015 at 1:12

@Curl, it was nice if you had included some code-examples. – T.Todua Sep 28, 2017 at 15:01

25

You can only dispose instances that implement the IDisposable interface.

To force a garbage collect to free up the (unmanaged) memory immediately:

```
GC.Collect();
GC.WaitForPendingFinalizers();
```

This is normally bad practice, but there is for example a bug in the x64-version of the .NET framework that makes the GC behave strange in some scenarios, and then you might want to do this. I don't know if the bug have been resolved yet. Does anyone know?

To dispose a class you do this:

```
instance.Dispose();
```

or like this:

```
using(MyClass instance = new MyClass())
{
    // Your cool code.
}
```

that will translate at compile-time to:

```
MyClass instance = null;

try
{
    instance = new MyClass();
    // Your cool code.
}
finally
{
    if(instance != null)
        instance.Dispose();
}
```

You can implement the IDisposable interface like this:

```
public class MyClass : IDisposable
{
    private bool disposed;

    /// <summary>
    /// Construction
    /// </summary>
    public MyClass()
    {
    }

    /// <summary>
    /// Destructor
    /// </summary>
    ~MyClass()
    {
        this.Dispose(false);
    }

    /// <summary>
    /// The dispose method that implements IDisposable.
    /// </summary>
    public void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }

    /// <summary>
    /// The virtual dispose method that allows
    /// classes inherithed from this one to dispose their resources.
    /// </summary>
    /// <param name="disposing"></param>
    protected virtual void Dispose(bool disposing)
```

```
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose managed resources here.
            }

            // Dispose unmanaged resources here.
        }

        disposed = true;
    }
}
```

Share

Improve this answer

Follow

edited Aug 26, 2008 at 9:31

Keith

155k ● 82 ● 306 ● 446

answered Aug 15, 2008 at 15:48

Patrik Svensson

13.8k ● 8 ● 58 ● 77

This is along the lines of what i'm trying to work out. If you have managed code... `MyClass` which you want to indicate your done with. Do you simply assign as null? which you saying is the same as implementing IDisposable? Its not easy explaining what i'm trying to asking. Basically I have a loop which news up a class on each loop I want to indicate that i'm done with it. Do I leave it as is or assign to null, or implement this. `uisng` statement without all the other bits (finiliser). empty dispose method. Or is this wrong. – Seabizkit Aug 6, 2017 at 8:15

@Seabizkit you should only implement IDisposable if your class is holding on to other things that are IDisposable (and invoke their Dispose method), and when you're utlilizing things like unmanaged resources. If you're worrying about GC cost in a loop, perhaps you should look into making it a value type (struct) or somehow reusing the instance by changing how it works. – Patrik Svensson Aug 7, 2017 at 2:41

20

The responses to this question have got more than a little confused.

The title asks about disposal, but then says that they want memory back immediately.

.Net is *managed*, which means that when you write .Net apps you don't need to worry about memory directly, the cost is that you don't have direct control over memory either.

.Net decides when it's best to clean up and free memory, not you as the .Net coder.

The `Dispose` is a way to tell .Net that you're done with something, but it won't actually free up the memory until it's the best time to do so.

Basically .Net will actually collect the memory back when it's easiest for it to do so - it's very good at deciding when. Unless you're writing something very memory

intensive you normally don't need to overrule it (this is part of the reason games aren't often written in .Net yet - they need complete control)

In .Net you can use `GC.Collect()` to force it to immediately, but that is almost always bad practise. If .Net hasn't cleaned it up yet that means it isn't a particularly good time for it to do so.

`GC.Collect()` picks up the objects that .Net identifies as done with. If you haven't disposed an object that needs it .Net may decide to keep that object. This means that `GC.Collect()` is only effective if you correctly implement your disposable instances.

`GC.Collect()` is **not** a replacement for correctly using IDisposable.

So Dispose and memory are not directly related, but they don't need to be. Correctly disposing will make your .Net apps more efficient and therefore use less memory though.

99% of the time in .Net the following is best practice:

Rule 1: If you don't deal with anything *unmanaged* or that implements `IDisposable` then don't worry about Dispose.

Rule 2: If you have a local variable that implements IDisposable make sure that you get rid of it in the current scope:

```
//using is best practice
using( SqlConnection con = new SqlConnection("my con str" ) )
{
    //do stuff
}

//this is what 'using' actually compiles to:
SqlConnection con = new SqlConnection("my con str" ) ;
try
{
    //do stuff
}
finally
{
    con.Dispose();
}
```

Rule 3: If a class has a property or member variable that implements IDisposable then that class should implement IDisposable too. In that class's Dispose method you can also dispose of your IDisposable properties:

```
//rather basic example
public sealed MyClass :
   IDisposable
```

```
{
    //this connection is disposable
    public SqlConnection MyConnection { get; set; }

    //make sure this gets rid of it too
    public Dispose()
    {
        //if we still have a connection dispose it
        if( MyConnection != null )
            MyConnection.Dispose();

        //note that the connection might have already been disposed
        //always write disposals so that they can be called again
    }
}
```

This isn't really complete, which is why the example is sealed. Inheriting classes may need to observe the next rule...

Rule 4: If a class uses an *unmanaged* resource then implement IDispose *and* add a finaliser.

.Net can't do anything with the *unmanaged* resource, so now we are talking about memory. If you don't clean it up you can get a memory leak.

The Dispose method needs to deal with both *managed* and *unmanaged* resources.

The finaliser is a safety catch - it ensures that if someone else creates and instance of your class and fails to dispose it the 'dangerous' *unmanaged* resources can still be cleaned up by .Net.

```
~MyClass()
{
    //calls a protected method
    //the false tells this method
    //not to bother with managed
    //resources
    this.Dispose(false);
}

public void Dispose()
{
    //calls the same method
    //passed true to tell it to
    //clean up managed and unmanaged
    this.Dispose(true);

    //as dispose has been correctly
    //called we don't need the

    //'backup' finaliser
    GC.SuppressFinalize(this);
}
```

Finally this overload of Dispose that takes a boolean flag:

```
protected virtual void Dispose(bool disposing)
{
    //check this hasn't been called already
    //remember that Dispose can be called again
    if (!disposed)
    {
        //this is passed true in the regular Dispose
        if (disposing)
        {
            // Dispose managed resources here.
        }

        //both regular Dispose and the finaliser
        //will hit this code
        // Dispose unmanaged resources here.
    }

    disposed = true;
}
```

Note that once this is all in place other managed code creating an instance of your class can just treat it like any other IDisposable (Rules 2 and 3).

Share

Improve this answer

Follow

edited Sep 1, 2008 at 9:09

answered Aug 31, 2008 at 9:56

Keith

155k ● 82 ● 306 ● 446

14

Would it be appropriate to also mention that dispose doesn't always refer to memory? I dispose resources such a references to files more often than memory. GC.Collect() directly relates to the CLR garbage collector and may or may not free memory (in Task Manager). It will likely impact your application in negative ways (eg performance).

At the end of the day why do you want the memory back immediately? If there is memory pressure from elsewhere the OS will get you memory in most cases.

Share Improve this answer Follow

answered Aug 15, 2008 at 15:41

Brian Lyttle

14.6k ● 15 ● 69 ● 106

6

Take a look at this article

Implementing the Dispose pattern, IDisposable, and/or a finalizer has absolutely nothing to do with when memory gets reclaimed; instead, it has everything to do with telling the GC ***how*** to reclaim that memory. When you call Dispose() you are in no way interacting with the GC.

The GC will only run when it determines the need to (called memory pressure) and then (and only then) will it deallocate memory for unused objects and compact the memory space.

You *could* call GC.Collect() but you really shouldn't unless there is a **very** good reason to (which is almost always "Never"). When you force an out-of-band collection cycle like this you actually cause the GC to do more work and ultimately can end up hurting your applications performance. For the duration of the GC collection cycle your application is actually in a frozen state...the more GC cycles that run, the more time your application spends frozen.

There are also some native Win32 API calls you can make to free your working set, but even those should be avoided unless there is a **very** good reason to do it.

The whole premise behind a gargbage collected runtime is that you don't need to worry (as much) about when the runtime allocates/deallocates actual memory; you only need to worry about making sure the your object knows how to clean up after itself when asked.

Share
Improve this answer
Follow

edited Aug 24, 2011 at 9:53
user906780

answered Aug 17, 2008 at 14:22

Scott Dorman
42.5k ● 12 ● 81 ● 112

5

I wrote a summary of Destructors and Dispose and Garbage collection on http://codingcraftsman.wordpress.com/2012/04/25/to-dispose-or-not-to-dispose/

To answer the original question:

1. Don't try to manage your memory
2. Dispose is not about memory management, it's about unmanaged resource management
3. Finalizers are an innate part of the Dispose pattern and actually slow down memory freeing of managed objects (as they have to go into the Finalization queue unless already Dispose d)

4. GC.Collect is bad as it makes some short-lived objects appear to be required for longer and so slows them down from being collected.

However, GC.Collect could be useful if you had a performance critical section of code and wanted to reduce the likelihood of Garbage Collection slowing it down. You call that before.

On top of that, there is an argument in favour of this pattern:

```
var myBigObject = new MyBigObject(1);
// something happens
myBigObject = new MyBigObject(2);
// at the above line, there are temporarily two big objects in memory and
neither can be collected
```

vs

```
myBigObject = null; // so it could now be collected
myBigObject = new MyBigObject(2);
```

But the main answer is that Garbage Collection just works unless you mess around with it!

Share

Improve this answer

Follow

answered May 2, 2012 at 22:13

community wiki
Ashley Frieze

3

```
public class MyClass : IDisposable
{
    public void Dispose()
    {
       // cleanup here
    }
}
```

then you can do something like this

```
MyClass todispose = new MyClass();
todispose.Dispose(); // instance is disposed right here
```

or

```
using (MyClass instance = new MyClass())
{
```

```
}
// instance will be disposed right here as it goes out of scope
```

Share Improve this answer Follow

answered Aug 15, 2008 at 15:32

Darren Kopp

77.6k ● 9 ● 80 ● 93

3

Complete explanation by Joe Duffy on "Dispose, Finalization, and Resource Management":

> Earlier in the .NET Framework's lifetime, finalizers were consistently referred to as destructors by C# programmers. As we become smarter over time, we are trying to come to terms with the fact that the **Dispose method is really more equivalent to a C++ destructor (deterministic)**, while the **finalizer is something entirely separate (nondeterministic)**. The fact that C# borrowed the C++ destructor syntax (i.e. ~T()) surely had at least a little to do with the development of this misnomer.

Share Improve this answer Follow

answered Aug 26, 2008 at 9:36

jfs

16.7k ● 13 ● 63 ● 90

2

You can't really force a GC to clean up an object when you want, although there are ways to force it to run, nothing says it's clean up the all the object you want/expect. It's best to call dispose in a try catch ex finally dispose end try (VB.NET rulz) way. But Dispose is for cleaning up system resources (memory, handles, db connections, etc. allocated by the object in deterministic way. Dispose doesn't (and can't) clean up the memory used by the object itself, only the the GC can do that.

Share Improve this answer Follow

answered Aug 15, 2008 at 15:42

Booji Boy

4,582 ● 4 ● 41 ● 45

1

This article has a pretty straightforward walkthrough. However, *having* to call the GC instead of letting it take its natural course is generally a sign of bad design/memory management, **especially** if no limited resources are being consumed (connections, handles, anything else that typically leads to implementing IDisposable).

What's causing you to need to do this?

Share

Improve this answer

Follow

edited Aug 15, 2008 at 15:34

answered Aug 15, 2008 at 15:30

Greg Hurlman

17.8k ● 6 ● 55 ● 87

1

@Keith,

I agree with all of your rules except #4. Adding a finalizer should only be done under very specific circumstances. If a class uses unmanaged resources, those should be cleaned up in your Dispose(bool) function. This same function should only cleanup managed resources when bool is true. Adding a finalizer adds a complexity cost to using your object as each time you create a new instance it must also be placed on the finalization queue, which is checked each time the GC runs a collection cycle. Effectively, this means that your object survives one cycle/generation longer than it should so the finalizer can be run. The finalizer should not be thought of as a "safety net".

The GC will only run a collection cycle when it determines that there is not enough available memory in the Gen0 heap to perform the next allocation, unless you "help" it by calling GC.Collect() to force an out-of-band collection.

The bottom line is that, no matter what, the GC only knows how to release resources by calling the Dispose method (and possibly the finalizer if one is implemented). It is up to that method to "do the right thing" and clean up any unmanaged resources used and instruct any other managed resources to call their Dispose method. It is very efficient at what it does and can self-optimize to a large extent as long as it isn't helped by out-of-band collection cycles. That being said, short of calling GC.Collect explicitly you have no control over when and in what order objects will be disposed of and memory released.

Share Improve this answer Follow

answered Aug 31, 2008 at 10:26

Scott Dorman

42.5k ● 12 ● 81 ● 112

1

The selected answer here is incorrect. As a few people have stated subsequently Dispose and implementing IDisposable has nothing to do with freeing the memory associated with a .NET class. It is mainly and traditionally used to free unmanaged resources such as file handles etc.

While your application can call GC.Collect() to try to force a collection by the garbage collector this will only really have an effect on those items that are at the correct generation level in the freachable queue. So it is possible that if you have cleared all references to the object it might still be a couple of calls to GC.Collect() before the actual memory is freed.

You don't say in your question WHY you feel the need to free up memory immediately. I understand that sometimes there can be unusual circumstances but seriously, in managed code it is almost always best to let the runtime deal with memory management.

Probably the best advice if you think your code is using up memory quicker than the GC is freeing it then you should review your code to ensure that no objects that are no longer needed are referenced in any data structures you have lying around in static members etc. Also try to avoid situations where you have circular object references as it is possible that these may not be freed either.

Share
Improve this answer
Follow

edited Nov 1, 2023 at 2:47

Nimantha
6,438 ● 6 ● 30 ● 75

answered Aug 25, 2008 at 21:29

Shaun Austin
3,822 ● 3 ● 24 ● 27

0

If MyClass implements IDisposable you can do just that.

```
MyClass.Dispose();
```

Best practice in C# is:

```
using( MyClass x = new MyClass() ) {
    //do stuff
}
```

As that wraps up the dispose in a try-finally and makes sure that it's never missed.

Share Improve this answer Follow

answered Aug 15, 2008 at 15:32

Keith
155k ● 82 ● 306 ● 446

0

If you don't want to (or can't) implement IDisposable on your class, you can force garbage collection like this (but it's slow) -

```
GC.Collect();
```

Share Improve this answer Follow

answered Aug 15, 2008 at 15:33

Seibar
70.2k ● 38 ● 93 ● 100

IDisposable interface is really for classes that contain unmanaged resources. If your class doesn't contain unmanaged resources, why do you **need** to free up resources before the garbage collector does it? Otherwise, just ensure your object is instantiated as late as possible and goes out of scope as soon as possible.

Share Improve this answer Follow

answered Aug 15, 2008 at 16:48

Tundey
2,965 ● 1 ● 24 ● 28

You can have deterministic object destruction in c++

You never want to call GC.Collect, it messes with the self tuning of the garbage collector to detect memory pressure and in some cases do nothing other than increase the current generation of every object on the heap.

For those posting IDisposable answers. Calling a Dispose method doesn't destroy an object as the asker describes.

Share

Improve this answer

Follow

edited Aug 25, 2008 at 21:18

answered Aug 21, 2008 at 8:05

Brian Leahy
35.4k ● 12 ● 46 ● 60

@Keith:

> IDisposable is for managed resources.
>
> Finalisers are for unmanaged resources.

Sorry but that's just wrong. Normally, the finalizer does nothing at all. However, if the dispose pattern has been correctly implemented, the finalizer tries to invoke `Dispose`.

`Dispose` has two jobs:

- Free unmanaged resources, and
- free nested managed resources.

And here your statement comes into play because it's true that while finalizing, an object should never try to free nested managed resources as these may have already been freed. It must still free unmanaged resources though.

Still, finalizers have no job other than to call `Dispose` and tell it not to touch managed objects. `Dispose`, when called manually (or via `Using`), shall free all unmanaged

resources and pass the `Dispose` message on to nested objects (and base class methods) but this will *never* free any (managed) memory.

Share
Improve this answer
Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 • 1

answered Aug 26, 2008 at 9:11

Konrad Rudolph
545k • 139 • 956 • 1.2k

0

Konrad Rudolph - yup, normally the finaliser does nothing at all. You shouldn't implement it unless you are dealing with unmanaged resources.

Then, when you do implement it, you use Microsoft's dispose pattern (as already described)

- `public Dispose()` calls `protected Dispose(true)` - deals with both managed and unmanaged resources. Calling `Dispose()` should suppress finalisation.
- `~Finalize` calls `protected Dispose(false)` - deals with unmanaged resources only. This prevents unmanaged memory leaks if you fail to call the `public Dispose()`

`~Finalize` is slow, and shouldn't be used unless you do have unmanaged resources to deal with.

Managed resources can't memory leak, they can only waste resources for the current application and slow its garbage collection. Unmanaged resources can leak, and `~Finalize` is best practice to ensure that they don't.

In either case `using` is best practice.

Share
Improve this answer
Follow

edited Aug 28, 2008 at 16:06

answered Aug 26, 2008 at 9:30

Keith
155k • 82 • 306 • 446

0

@Curt Hagenlocher - that's back to front. I've no idea why so many have voted it up when it's wrong.

`IDisposable` is for *managed* resources.

Finalisers are for *unmanaged* resources.

As long as you only use managed resources both @Jon Limjap and myself are entirely correct.

For classes that use unmanaged resources (and bear in mind that the vast majority of .Net classes don't) Patrik's answer is comprehensive and best practice.

Avoid using GC.Collect - it is a slow way to deal with managed resources, and doesn't do anything with unmanaged ones unless you have correctly built your ~Finalizers.

Share

Improve this answer

Follow

edited Nov 1, 2023 at 2:48

Nimantha

6,438 ● 6 ● 30 ● 75

answered Aug 26, 2008 at 9:01

Keith

155k ● 82 ● 306 ● 446

-1

In answer to the original question, with the information given so far by the original poster, it is 100% certain that he does not know enough about programming in .NET to even be given the answer: use GC.Collect(). I would say it is 99.99% likely that he really doesn't need to use GC.Collect() at all, as most posters have pointed out.

The correct answer boils down to 'Let the GC do its job. Period. You have other stuff to worry about. But you might want to consider whether and when you should dispose of or clean up specific objects, and whether you need to implement IDisposable and possibly Finalize in your class.'

Regarding Keith's post and his Rule #4:

Some posters are confusing rule 3 and rule 4. Keith's rule 4 is absolutely correct, unequivocately. It's the one rule of the four that needs no editing at all. I would slightly rephrase some of his other rules to make them clearer, but they are essentially correct if you parse them correctly, and actually read the whole post to see how he expands on them.

1. If your class doesn't use an unmanaged resource AND it also never instantiates another object of a class that itself uses, directly or ultimately, an unmanaged object (i.e., a class that implements IDisposable), then there would be no need for your class to either implement IDisposable itself, or even call .dispose on anything. (In such a case, it is silly to think you actually NEED to immediately free up memory with a forced GC, anyway.)
2. If your class uses an unmanaged resource, OR instantiates another object that itself implements IDisposable, then your class should either:

   a) dispose/release these immediately in a local context in which they were created, OR...

   b) implement IDisposable in the pattern recommended within Keith's post, or a few thousand places on the internet, or in literally about 300 books by now.

   b.1) Furthermore, if (b), and it is an unmanaged resource that has been opened, both IDisposable AND Finalize SHOULD ALWAYS be implemented, per Keith's

Rule #4.
In this context, Finalize absolutely IS a safety net in one sense: if someone instantiates YOUR IDisposable object that uses an unmanaged resource, and they fail to call dispose, then Finalize is the last chance for YOUR object to close the unmanaged resource properly.
(Finalize should do this by calling Dispose in such a way that the Dispose method skips over releasing anything BUT the unmanaged resource. Alternatively, if your object's Dispose method IS called properly by whatever instantiated your object, then it BOTH passes on the Dispose call to all IDisposable objects it has instantiated, AND releases the unmanaged resources properly, ending with a call to suppress the Finalize on your object, which means that the impact of using Finalize is reduced if your object is disposed properly by the caller. All of these points are included in Keith's post, BTW.)

b.2) IF your class is only implementing IDisposable because it needs to essentially pass on a Dispose to an IDisposable object it has instantiated, then don't implement a Finalize method in your class in that case. Finalize is for handling the case that BOTH Dispose was never called by whatever instantiated your object, AND an unmanaged resource was utilized that's still unreleased.

In short, regarding Keith's post, he is completely correct, and that post is the most correct and complete answer, in my opinion. He may use some short-hand statements that some find 'wrong' or object to, but his full post expands on the usage of Finalize completely, and he is absolutely correct. Be sure to read his post completely before jumping on one of the rules or preliminary statements in his post.

edited Mar 23, 2012 at 1:25

community wiki
3 revs
Jaycephus

3 Thanks for pointing out I do not know enough about programming in .Net and therefore should not be given an answer (even without taking into account feeding stackoverflow while it was still starting up)! I gather it was late at night that you wrote the answer and normally do better. – Jorrit Reedijk May 9, 2012 at 21:40

@Jorrit Is your name ALSO Jon Galloway or OP? – Jaycephus Oct 16, 2018 at 21:44