Proper use of the IDisposable interface

Asked 15 years, 10 months ago Modified 1 year, 8 months ago Viewed 445k times

1908

I know from reading Microsoft documentation that the "primary" use of the `IDisposable` interface is to clean up unmanaged resources.

To me, "unmanaged" means things like database connections, sockets, window handles, etc. But, I've seen code where the `Dispose()` method is implemented to free *managed* resources, which seems redundant to me, since the garbage collector should take care of that for you.

For example:

```
public class MyCollection : IDisposable
{
    private List<String> _theList = new List<String>();
    private Dictionary<String, Point> _theDict = new Dictionary<String, Point>
();

    // Die, clear it up! (free unmanaged resources)
    public void Dispose()
    {
        _theList.clear();
        _theDict.clear();
        _theList = null;
        _theDict = null;
    }
}
```

My question is, does this make the garbage collector free memory used by `MyCollection` any faster than it normally would?

Edit: So far people have posted some good examples of using `IDisposable` to clean up unmanaged resources such as database connections and bitmaps. But suppose that `_theList` in the above code contained a million strings, and you wanted to free that memory *now*, rather than waiting for the garbage collector. Would the above code accomplish that?

c# .net garbage-collection idisposable

Share
Improve this question
Follow

edited May 13, 2022 at 11:45
spaleet
926 ●2 ●13 ●25

asked Feb 11, 2009 at 18:12
cwick
26.6k ●12 ●39 ●42

45 I like the accepted answer because it tell you the correct 'pattern' of using IDisposable, but like the OP said in his edit, it does not answer his intended question. IDisposable does not 'call' the GC, it just 'marks' an object as destroyable. But what is the real way to free memory 'right now' instead of waiting for GC to kick in? I think this question deserves more discussion. – Punit Vora Oct 1, 2010 at 17:15

55 `IDisposable` doesn't mark anything. The `Dispose` method does what it has to do to clean up resources used by the instance. This has nothing to do with GC. – John Saunders Oct 1, 2010 at 19:37

5 @John. I do understand `IDisposable`. And which is why I said that the accepted answer does not answer the OP's intended question(and follow-up edit) about whether IDisposable will help in <i>freeing memory</i>. Since `IDisposable` has nothing to do with freeing memory, only resources, then like you said, there is no need to set the managed references to null at all which is what OP was doing in his example. So, the correct answer to his question is "No, it does not help free memory any faster. In fact, it does not help free memory at all, only resources". But anyway, thanks for your input. – Punit Vora Oct 7, 2010 at 15:17

13 @desigeek: if this is the case, then you should not have said "IDisposable does not 'call' the GC, it just 'marks' an object as destroyable" – John Saunders Oct 7, 2010 at 15:31

8 @desigeek: There is no guaranteed way of freeing memory deterministically. You could call GC.Collect(), but that is a polite request, not a demand. All running threads must be suspended for garbage collection to proceed - read up on the concept of .NET safepoints if you want to learn more, e.g. msdn.microsoft.com/en-us/library/678ysw69(v=vs.110).aspx . If a thread cannot be suspended, e.g. because there's a call into unmanaged code, GC.Collect() may do nothing at all. – Concrete Gannet Aug 14, 2015 at 5:15

20 Answers

Sorted by: Highest score (default)

2978

The point of Dispose **is** to free unmanaged resources. It needs to be done at some point, otherwise they will never be cleaned up. The garbage collector doesn't know **how** to call `DeleteHandle()` on a variable of type `IntPtr`, it doesn't know **whether** or not it needs to call `DeleteHandle()`.

> **Note**: What is an *unmanaged resource*? If you found it in the Microsoft .NET Framework: it's managed. If you went poking around MSDN yourself, it's unmanaged. Anything you've used P/Invoke calls to get outside of the nice comfy world of everything available to you in the .NET Framework is unmanaged – and you're now responsible for cleaning it up.

The object that you've created needs to expose *some* method, that the outside world can call, in order to clean up unmanaged resources. The method can be named whatever you like:

```
public void Cleanup()
```

or

```
public void Shutdown()
```

But instead there is a standardized name for this method:

```
public void Dispose()
```

There was even an interface created, `IDisposable`, that has just that one method:

```
public interface IDisposable
{
   void Dispose();
}
```

So you make your object expose the `IDisposable` interface, and that way you promise that you've written that single method to clean up your unmanaged resources:

```
public void Dispose()
{
   Win32.DestroyHandle(this.CursorFileBitmapIconServiceHandle);
}
```

And you're done. **Except you can do better.**

What if your object has allocated a 250MB **System.Drawing.Bitmap** (i.e. the .NET managed Bitmap class) as some sort of frame buffer? Sure, this is a managed .NET object, and the garbage collector will free it. But do you really want to leave 250MB of memory just sitting there – waiting for the garbage collector to *eventually* come along and free it? What if there's an open database connection? Surely we don't want that connection sitting open, waiting for the GC to finalize the object.

If the user has called `Dispose()` (meaning they no longer plan to use the object) why not get rid of those wasteful bitmaps and database connections?

So now we will:

- get rid of unmanaged resources (because we have to), and
- get rid of managed resources (because we want to be helpful)

So let's update our `Dispose()` method to get rid of those managed objects:

```
public void Dispose()
{
   //Free unmanaged resources
   Win32.DestroyHandle(this.CursorFileBitmapIconServiceHandle);

   //Free managed resources too
   if (this.databaseConnection != null)
   {
      this.databaseConnection.Dispose();
      this.databaseConnection = null;
   }
   if (this.frameBufferImage != null)
   {
      this.frameBufferImage.Dispose();
      this.frameBufferImage = null;
   }
}
```

And all is good, **except you can do better**!

What if the person **forgot** to call `Dispose()` on your object? Then they would leak some **unmanaged** resources!

> **Note:** They won't leak **managed** resources, because eventually the garbage collector is going to run, on a background thread, and free the memory associated with any unused objects. This will include your object, and any managed objects you use (e.g. the `Bitmap` and the `DbConnection`).

If the person forgot to call `Dispose()`, we can *still* save their bacon! We still have a way to call it *for* them: when the garbage collector finally gets around to freeing (i.e. finalizing) our object.

> **Note:** The garbage collector will eventually free all managed objects. When it does, it calls the **`Finalize`** method on the object. The GC doesn't know, or care, about *your* **Dispose** method. That was just a name we chose for a method we call when we want to get rid of unmanaged stuff.

The destruction of our object by the Garbage collector is the *perfect* time to free those pesky unmanaged resources. We do this by overriding the `Finalize()` method.

> **Note:** In C#, you don't explicitly override the `Finalize()` method. You write a method that *looks like* a **C++ destructor**, and the compiler takes that to be your implementation of the `Finalize()` method:

```
~MyObject()
{
    //we're being finalized (i.e. destroyed), call Dispose in case the user
forgot to
    Dispose(); //<--Warning: subtle bug! Keep reading!
}
```

But there's a bug in that code. You see, the garbage collector runs on a **background thread**; you don't know the order in which two objects are destroyed. It is entirely possible that in your `Dispose()` code, the **managed** object you're trying to get rid of (because you wanted to be helpful) is no longer there:

```
public void Dispose()
{
   //Free unmanaged resources
   Win32.DestroyHandle(this.gdiCursorBitmapStreamFileHandle);

   //Free managed resources too
   if (this.databaseConnection != null)
   {
      this.databaseConnection.Dispose(); //<-- crash, GC already destroyed it
      this.databaseConnection = null;
   }
   if (this.frameBufferImage != null)
   {
      this.frameBufferImage.Dispose(); //<-- crash, GC already destroyed it
      this.frameBufferImage = null;
   }
}
```

So what you need is a way for `Finalize()` to tell `Dispose()` that it should **not touch any managed** resources (because they *might not be there* anymore), while still freeing unmanaged resources.

The standard pattern to do this is to have `Finalize()` and `Dispose()` both call a **third**(!) method; where you pass a Boolean saying if you're calling it from `Dispose()` (as opposed to `Finalize()`), meaning it's safe to free managed resources.

This *internal* method *could* be given some arbitrary name like "CoreDispose", or "MyInternalDispose", but is tradition to call it `Dispose(Boolean)`:

```
protected void Dispose(Boolean disposing)
```

But a more helpful parameter name might be:

```
protected void Dispose(Boolean itIsSafeToAlsoFreeManagedObjects)
{
   //Free unmanaged resources
   Win32.DestroyHandle(this.CursorFileBitmapIconServiceHandle);

```

```
    //Free managed resources too, but only if I'm being called from Dispose
    //(If I'm being called from Finalize then the objects might not exist
    //anymore
    if (itIsSafeToAlsoFreeManagedObjects)
    {
       if (this.databaseConnection != null)
       {
          this.databaseConnection.Dispose();
          this.databaseConnection = null;
       }
       if (this.frameBufferImage != null)
       {
          this.frameBufferImage.Dispose();
          this.frameBufferImage = null;
       }
    }
}
```

And you change your implementation of the `IDisposable.Dispose()` method to:

```
public void Dispose()
{
   Dispose(true); //I am calling you from Dispose, it's safe
}
```

and your finalizer to:

```
~MyObject()
{
   Dispose(false); //I am *not* calling you from Dispose, it's *not* safe
}
```

> **Note**: If your object descends from an object that implements `Dispose`, then don't forget to call their **base** Dispose method when you override Dispose:

```
public override void Dispose()
{
    try
    {
        Dispose(true); //true: safe to free managed resources
    }
    finally
    {
        base.Dispose();
    }
}
```

And all is good, **except you can do better**!

If the user calls `Dispose()` on your object, then everything has been cleaned up. Later on, when the garbage collector comes along and calls Finalize, it will then call `Dispose` again.

Not only is this wasteful, but if your object has junk references to objects you already disposed of from the **last** call to `Dispose()`, you'll try to dispose them again!

You'll notice in my code I was careful to remove references to objects that I've disposed, so I don't try to call `Dispose` on a junk object reference. But that didn't stop a subtle bug from creeping in.

When the user calls `Dispose()`: the handle **CursorFileBitmapIconServiceHandle** is destroyed. Later when the garbage collector runs, it will try to destroy the same handle again.

```
protected void Dispose(Boolean iAmBeingCalledFromDisposeAndNotFinalize)
{
   //Free unmanaged resources
   Win32.DestroyHandle(this.CursorFileBitmapIconServiceHandle); //<--double
destroy
   ...
}
```

The way you fix this is tell the garbage collector that it doesn't need to bother finalizing the object – its resources have already been cleaned up, and no more work is needed. You do this by calling `GC.SuppressFinalize()` in the `Dispose()` method:

```
public void Dispose()
{
   Dispose(true); //I am calling you from Dispose, it's safe
   GC.SuppressFinalize(this); //Hey, GC: don't bother calling finalize later
}
```

Now that the user has called `Dispose()`, we have:

- freed unmanaged resources
- freed managed resources

There's no point in the GC running the finalizer – everything's taken care of.

Couldn't I use Finalize to cleanup unmanaged resources?

The documentation for `Object.Finalize` says:

> The Finalize method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed.

But the MSDN documentation also says, for `IDisposable.Dispose`:

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

So which is it? Which one is the place for me to cleanup unmanaged resources? The answer is:

> It's your choice! But choose `Dispose`.

You certainly could place your unmanaged cleanup in the finalizer:

```
~MyObject()
{
   //Free unmanaged resources
   Win32.DestroyHandle(this.CursorFileBitmapIconServiceHandle);

   //A C# destructor automatically calls the destructor of its base class.
}
```

The problem with that is you have no idea when the garbage collector will get around to finalizing your object. Your un-managed, un-needed, un-used native resources will stick around until the garbage collector *eventually* runs. Then it will call your finalizer method; cleaning up unmanaged resources. The documentation of **Object.Finalize** points this out:

> 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.

This is the virtue of using `Dispose` to cleanup unmanaged resources; you get to know, and control, when unmanaged resource are cleaned up. Their destruction is *"deterministic"*.

To answer your original question: Why not release memory now, rather than for when the GC decides to do it? I have a facial recognition software that *needs* to get rid of 530 MB of internal images **now**, since they're no longer needed. When we don't: the machine grinds to a swapping halt.

Bonus Reading

For anyone who likes the style of this answer (explaining the *why*, so the *how* becomes obvious), I suggest you read Chapter One of Don Box's Essential COM:

- Direct link: Chapter 1 sample by Pearson Publishing
- magnet: 84bf0b960936d677190a2be355858e80ef7542c0

In 35 pages he explains the problems of using binary objects, and invents COM before your eyes. Once you realize the *why* of COM, the remaining 300 pages are obvious, and just detail Microsoft's implementation.

I think every programmer who has ever dealt with objects or COM should, at the very least, read the first chapter. It is the best explanation of anything ever.

Extra Bonus Reading

When everything you know is wrong archive by Eric Lippert

> It is therefore very difficult indeed to write a correct finalizer, and **the best advice I can give you is to not try**.

Share

Improve this answer

Follow

edited Apr 9, 2023 at 15:03

community wiki
43 revs, 23 users 80%
Ian Boyd

---

48 This is a great answer but I think it would however benefit from a final code listing for a standard case and for a case where the the class derives from a baseclass that already implements Dispose. e.g having read here (msdn.microsoft.com/en-us/library/aa720161%28v=vs.71%29.aspx) as well I have got confused about what I should do when deriving from the class that already implements Dispose (hey I'm new to this). – integra753 Feb 9, 2012 at 12:42

---

3 @Ayce "If you write correct code, you never need the finalizer/Dispose(bool) thingy." I'm not protecting against me; i'm protecing against the dozens, hundreds, thousands, or millions of other developers who might not get it right every time. Sometimes developers forget to call `.Dispose`. Sometimes you can't use `using`. We're following the .NET/WinRT approach of *"the pit of success"*. We pay our developer taxes, and write better and defensive code to make it resilient to these problems. – Ian Boyd Sep 13, 2021 at 13:53

---

3 "But you don't always have to write code for "the public"." But when trying to come up with best practices for a 2k+ upvoted answer, meant for general introduction to unmanaged memory, it's best to provide the best code samples possible. We don't want to leave it all out - and let people stumble into all this the hard way. Because that's the reality - thousands of developers each year learning this nuance about Disposing. No need to make it needlessly harder for them. – Ian Boyd Sep 13, 2021 at 18:55

2 @toha It's two methods. And really you can ignore one of them leaving you with just `Dispose` - where you clean up **unmanaged** resources only. You don't **have** to override the finalizer and try to call `Dispose` during finalization - the developer using your class is **required** (mandatory, no ifs, ands, or buts) to call `.Dispose` themselves. If they didn't call `.Dispose`: well that on them for not following the ground rules. The rest of the answer is for developers who want to try to make their classes more resilient to bad programmers. *(i.e. "you can do better")* – Ian Boyd Feb 22, 2023 at 15:16

2 The best answer I've ever read. @IanBoyd Thanks for your precious time to give such an insightful and resourceful answer You deserve these upvotes man! – Keyrad Mar 19, 2023 at 16:34

85

`IDisposable` is often used to exploit the `using` statement and take advantage of an easy way to do deterministic cleanup of managed objects.

```
public class LoggingContext : IDisposable {
    public Finicky(string name) {
        Log.Write("Entering Log Context {0}", name);
        Log.Indent();
    }
    public void Dispose() {
        Log.Outdent();
    }

    public static void Main() {
        Log.Write("Some initial stuff.");
        try {
            using(new LoggingContext()) {
                Log.Write("Some stuff inside the context.");
                throw new Exception();
            }
        } catch {
            Log.Write("Man, that was a heavy exception caught from inside a
child logging context!");
        } finally {
            Log.Write("Some final stuff.");
        }
    }
}
```

Share Improve this answer Follow

answered Feb 11, 2009 at 18:20

yfeldblum
65.4k ● 12 ● 131 ● 169

54

The purpose of the Dispose pattern is to provide a mechanism to clean up both managed and unmanaged resources and when that occurs depends on how the Dispose method is being called. In your example, the use of Dispose is not actually doing anything related to dispose, since clearing a list has no impact on that collection being disposed. Likewise, the calls to set the variables to null also have no impact on the GC.

You can take a look at this article for more details on how to implement the Dispose pattern, but it basically looks like this:

```
public class SimpleCleanup : IDisposable
{
    // some fields that require cleanup
    private SafeHandle handle;
    private bool disposed = false; // to detect redundant calls

    public SimpleCleanup()
    {
        this.handle = /*...*/;
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose managed resources.
                if (handle != null)
                {
                    handle.Dispose();
                }
            }

            // Dispose unmanaged managed resources.

            disposed = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

The method that is the most important here is the Dispose(bool), which actually runs under two different circumstances:

- disposing == true: the method has been called directly or indirectly by a user's code. Managed and unmanaged resources can be disposed.
- disposing == 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.

The problem with simply letting the GC take care of doing the cleanup is that you have no real control over when the GC will run a collection cycle (you can call GC.Collect(), but you really shouldn't) so resources may stay around longer than needed. Remember, calling Dispose() doesn't actually cause a collection cycle or in any way cause the GC to collect/free the object; it simply provides the means to more deterministicly cleanup the resources used and tell the GC that this cleanup has already been performed.

The whole point of IDisposable and the dispose pattern isn't about immediately freeing memory. The only time a call to Dispose will actually even have a chance of immediately freeing memory is when it is handling the disposing == false scenario and manipulating unmanaged resources. For managed code, the memory won't actually be reclaimed until the GC runs a collection cycle, which you really have no control over (other than calling GC.Collect(), which I've already mentioned is not a good idea).

Your scenario isn't really valid since strings in .NET don't use any unamanged resources and don't implement IDisposable, there is no way to force them to be "cleaned up."

Share

Improve this answer

Follow

edited Feb 11, 2009 at 21:23

answered Feb 11, 2009 at 18:42

Scott Dorman

42.5k ● 12 ● 81 ● 112

what if i change if (handle != null) { handle.Dispose(); } to if (handle != null) { handle = null; } Is that any difference? – toha Feb 22, 2023 at 4:03

24

There should be no further calls to an object's methods after Dispose has been called on it (although an object should tolerate further calls to Dispose). Therefore the example in the question is silly. If Dispose is called, then the object itself can be discarded. So the user should just discard all references to that whole object (set them to null) and all the related objects internal to it will automatically get cleaned up.

As for the general question about managed/unmanaged and the discussion in other answers, I think any answer to this question has to start with a definition of an unmanaged resource.

What it boils down to is that there is a function you can call to put the system into a state, and there's another function you can call to bring it back out of that state. Now, in the typical example, the first one might be a function that returns a file handle, and the second one might be a call to `CloseHandle`.

But - and this is the key - they could be any matching pair of functions. One builds up a state, the other tears it down. If the state has been built but not torn down yet, then an instance of the resource exists. You have to arrange for the teardown to happen at the right time - the resource is not managed by the CLR. The only automatically managed resource type is memory. There are two kinds: the GC, and the stack. Value types are managed by the stack (or by hitching a ride inside reference types), and reference types are managed by the GC.

These functions may cause state changes that can be freely interleaved, or may need to be perfectly nested. The state changes may be threadsafe, or they might not.

Look at the example in Justice's question. Changes to the Log file's indentation must be perfectly nested, or it all goes wrong. Also they are unlikely to be threadsafe.

It is possible to hitch a ride with the garbage collector to get your unmanaged resources cleaned up. But only if the state change functions are threadsafe and two states can have lifetimes that overlap in any way. So Justice's example of a resource must NOT have a finalizer! It just wouldn't help anyone.

For those kinds of resources, you can just implement `IDisposable`, without a finalizer. The finalizer is absolutely optional - it has to be. This is glossed over or not even mentioned in many books.

You then have to use the `using` statement to have any chance of ensuring that `Dispose` is called. This is essentially like hitching a ride with the stack (so as finalizer is to the GC, `using` is to the stack).

The missing part is that you have to manually write Dispose and make it call onto your fields and your base class. C++/CLI programmers don't have to do that. The compiler writes it for them in most cases.

There is an alternative, which I prefer for states that nest perfectly and are not threadsafe (apart from anything else, avoiding IDisposable spares you the problem of having an argument with someone who can't resist adding a finalizer to every class that implements IDisposable).

Instead of writing a class, you write a function. The function accepts a delegate to call back to:

```
public static void Indented(this Log log, Action action)
{
    log.Indent();
    try
    {
        action();
    }
    finally
    {
        log.Outdent();
    }
}
```

And then a simple example would be:

```
Log.Write("Message at the top");
Log.Indented(() =>
{
    Log.Write("And this is indented");

    Log.Indented(() =>
    {
        Log.Write("This is even more indented");
    });
});
Log.Write("Back at the outermost level again");
```

The lambda being passed in serves as a code block, so it's like you make your own control structure to serve the same purpose as `using`, except that you no longer have any danger of the caller abusing it. There's no way they can fail to clean up the resource.

This technique is less useful if the resource is the kind that may have overlapping lifetimes, because then you want to be able to build resource A, then resource B, then kill resource A and then later kill resource B. You can't do that if you've forced the user to perfectly nest like this. But then you need to use `IDisposable` (but still without a finalizer, unless you have implemented threadsafety, which isn't free).

Share

Improve this answer

Follow

edited Feb 23, 2012 at 10:01

answered Feb 11, 2009 at 20:21

Daniel Earwicker
117k ● 38 ● 208 ● 286

18

Scenarios I make use of IDisposable: clean up unmanaged resources, unsubscribe for events, close connections

The idiom I use for implementing IDisposable (*not threadsafe*):

```
class MyClass : IDisposable {
    // ...

    #region IDisposable Members and Helpers
    private bool disposed = false;

    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    private void Dispose(bool disposing) {
        if (!this.disposed) {
            if (disposing) {
                // cleanup code goes here
            }
            disposed = true;
        }
    }
    ~MyClass() {
        Dispose(false);
    }
    #endregion
}
```

Share

Improve this answer

Follow

edited Oct 18, 2013 at 11:33

Jake1164
12.3k ● 6 ● 48 ● 65

answered Feb 11, 2009 at 19:31

olli-MSFT
2,546 ● 19 ● 19

1 This is almost the Microsoft Dispose pattern implementation except you've forgotten to make the DIspose(bool) virtual. The pattern itself is not a very good pattern and should be avoided unless you absolutely have to have dispose as part of an inheritance hierarchy. – MikeJ Jan 25, 2021 at 18:04

13

Yep, that code is completely redundant and unnecessary and it doesn't make the garbage collector do anything it wouldn't otherwise do (once an instance of MyCollection goes out of scope, that is.) Especially the `.Clear()` calls.

Answer to your edit: Sort of. If I do this:

```
public void WasteMemory()
{
    var instance = new MyCollection(); // this one has no Dispose() method
    instance.FillItWithAMillionStrings();
}

// 1 million strings are in memory, but marked for reclamation by the GC
```

It's functionally identical to this for purposes of memory management:

```
public void WasteMemory()
{
    var instance = new MyCollection(); // this one has your Dispose()
    instance.FillItWithAMillionStrings();
    instance.Dispose();
}

// 1 million strings are in memory, but marked for reclamation by the GC
```

If you really really really need to free the memory this very instant, call `GC.Collect()`. There's no reason to do this here, though. The memory will be freed when it's needed.

Share

Improve this answer

Follow

edited Feb 11, 2009 at 21:17

answered Feb 11, 2009 at 18:19

mqp
71.9k ● 14 ● 96 ● 123

3 re: "The memory will be freed when it's needed." Rather say, "when GC decides it's needed." You may see system performance issues before GC decides that memory is *really* needed. Freeing it up *now* may not be essential, but may be useful. – Jesse Chisholm Aug 30, 2012 at 23:40

1 There are some corner cases in which nulling out references within a collection may expedite garbage collection of the items referred to thereby. For example, if a large array is created and filled with references to smaller newly-created items, but it isn't needed for very long after that, abandoning the array may cause those items to be kept around until the next Level 2 GC, while zeroing it out first may make the items eligible for the next level 0 or level 1 GC. To be sure, having big short-lived objects on the Large Object Heap is icky anyway (I dislike the design) but... – supercat Apr 26, 2013 at 15:48

1 ...zeroing out such arrays before abandoning them my sometimes lessen the GC impact. – supercat Apr 26, 2013 at 15:49

In most cases nulling stuff is not required, but some objects may actually keep a bunch of other objects alive too, even when they aren't required anymore. Setting something like a reference to a Thread to null may be beneficial, but nowadays, probably not. Often the more complicated code if the big object could still be called upon in some method of checking if it has been nulled already isn't worth the performance gain. Prefer clean over "I think this is slightly faster". – AyCe Sep 13, 2021 at 12:07

12

If `MyCollection` is going to be garbage collected anyway, then you shouldn't need to dispose it. Doing so will just churn the CPU more than necessary, and may even invalidate some pre-calculated analysis that the garbage collector has already performed.

I use `IDisposable` to do things like ensure threads are disposed correctly, along with unmanaged resources.

EDIT In response to Scott's comment:

> *The only time the GC performance metrics are affected is when a call the [sic] GC.Collect() is made"*

Conceptually, the GC maintains a view of the object reference graph, and all references to it from the stack frames of threads. This heap can be quite large and span many pages of memory. As an optimisation, the GC caches its analysis of pages that are unlikely to change very often to avoid rescanning the page unnecessarily. The GC receives notification from the kernel when data in a page changes, so it knows that the page is dirty and requires a rescan. If the collection is in Gen0 then it's likely that other things in the page are changing too, but this is less likely in Gen1 and Gen2. Anecdotally, these hooks were not available in Mac OS X for the team who ported the GC to Mac in order to get the Silverlight plug-in working on that platform.

Another point against unnecessary disposal of resources: imagine a situation where a process is unloading. Imagine also that the process has been running for some time. Chances are that many of that process's memory pages have been swapped to disk. At the very least they're no longer in L1 or L2 cache. In such a situation there is no point for an application that's unloading to swap all those data and code pages back into memory to 'release' resources that are going to be released by the operating system anyway when the process terminates. This applies to managed and even certain unmanaged resources. Only resources that keep non-background threads alive must be disposed, otherwise the process will remain alive.

Now, during normal execution there are ephemeral resources that must be cleaned up correctly (as @fezmonkey points out *database connections, sockets, window handles*) to avoid unmanaged memory leaks. These are the kinds of things that have to be disposed. If you create some class that owns a thread (and by owns I mean that it created it and therefore is responsible for ensuring it stops, at least by my coding style), then that class most likely must implement `IDisposable` and tear down the thread during `Dispose`.

The .NET framework uses the `IDisposable` interface as a signal, even warning, to developers that the this class *must* be disposed. I can't think of any types in the framework that implement `IDisposable` (excluding explicit interface implementations) where disposal is optional.

Share

Improve this answer

Follow

edited Feb 12, 2009 at 9:34

answered Feb 11, 2009 at 18:20

Drew Noakes

310k ● 168 ● 696 ● 761

Calling Dispose is perfectly valid, legal, and encouraged. Objects that implement IDisposable usually do so for a reason. The only time the GC performance metrics are affected is when a call the GC.Collect() is made. – Scott Dorman Feb 11, 2009 at 21:17

For many .net classes, disposal is "somewhat" optional, meaning that abandoning instances "usually" won't cause any trouble so long as one doesn't go crazy creating new instances and abandoning them. For example, the compiler-generated code for controls seems to create fonts when the controls are instantiated and abandon them when the forms are disposed; if one creates and disposes thousands of controls , this could tie up thousands of GDI handles, but in most cases controls aren't created and destroyed that much. Nonetheless, one should still try to avoid such abandonment. – supercat Aug 2, 2011 at 15:14

1 In the case of fonts, I suspect the problem is that Microsoft never really defined what entity is responsible for disposing the "font" object assigned to a control; in some cases, a controls may share a font with a longer-lived object, so having the control Dispose the font would be bad. In other cases, a font will be assigned to a control and nowhere else, so if the control doesn't dispose it nobody will. Incidentally, this difficulty with fonts could have been avoided had there been a separate non-disposable FontTemplate class, since controls don't seem to use the GDI handle of their Font. – supercat Aug 2, 2011 at 15:23

8

I won't repeat the usual stuff about Using or freeing un-managed resources, that has all been covered. But I would like to point out what seems a common misconception.
Given the following code

```
Public Class LargeStuff
  Implements IDisposable
  Private _Large as string()

  'Some strange code that means _Large now contains several million long
strings.

  Public Sub Dispose() Implements IDisposable.Dispose
    _Large=Nothing
  End Sub
```

I realise that the Disposable implementation does not follow current guidelines, but hopefully you all get the idea.
Now, when Dispose is called, how much memory gets freed?

Answer: None.
Calling Dispose can release unmanaged resources, it CANNOT reclaim managed memory, only the GC can do that. Thats not to say that the above isn't a good idea, following the above pattern is still a good idea in fact. Once Dispose has been run, there is nothing stopping the GC re-claiming the memory that was being used by _Large, even though the instance of LargeStuff may still be in scope. The strings in _Large may also be in gen 0 but the instance of LargeStuff might be gen 2, so again, memory would be re-claimed sooner.
There is no point in adding a finaliser to call the Dispose method shown above though. That will just DELAY the re-claiming of memory to allow the finaliser to run.

Share Improve this answer Follow

answered Feb 11, 2009 at 21:08
pipTheGeek
2,703 ● 17 ● 16

1 If an instance of `LargeStuff` has been around long enough to make it to Generation 2, and if `_Large` holds a reference to a newly-created string which is in Generation 0, then if the the instance of `LargeStuff` is abandoned without nulling out `_Large`, then string referred to by `_Large` will be kept around until the next Gen2 collection. Zeroing out `_Large` may let the string get eliminated at the next Gen0 collection. In most cases, nulling out references is not helpful, but there are cases where it can offer some benefit. – supercat May 9, 2013 at 23:10

In the example you posted, it still doesn't "free the memory now". All memory is garbage collected, but it may allow the memory to be collected in an earlier generation. You'd have to run some tests to be sure.

7

The Framework Design Guidelines are guidelines, and not rules. They tell you what the interface is primarily for, when to use it, how to use it, and when not to use it.

I once read code that was a simple RollBack() on failure utilizing IDisposable. The MiniTx class below would check a flag on Dispose() and if the `Commit` call never happened it would then call `Rollback` on itself. It added a layer of indirection making the calling code a lot easier to understand and maintain. The result looked something like:

```
using( MiniTx tx = new MiniTx() )
{
    // code that might not work.

    tx.Commit();
}
```

I've also seen timing / logging code do the same thing. In this case the Dispose() method stopped the timer and logged that the block had exited.

```
using( LogTimer log = new LogTimer("MyCategory", "Some message") )
{
    // code to time...
}
```

So here are a couple of concrete examples that don't do any unmanaged resource cleanup, but do successfully used IDisposable to create cleaner code.

Share Improve this answer Follow

answered Feb 11, 2009 at 21:07

Robert Paulson
18k ● 6 ● 36 ● 53

Take a look at @Daniel Earwicker's example using higher order functions. For benchmarking, timing, logging etc. It seems much more straightforward. – Aluan Haddad Sep 22, 2016 at 3:15

7

If you want to **delete right now**, use **unmanaged memory**.

See:

- Marshal.AllocHGlobal
- Marshal.FreeHGlobal
- Marshal.DestroyStructure

Share Improve this answer Follow

answered Jun 3, 2013 at 21:07
franckspike
2,204 ● 26 ● 19

5

If anything, I'd expect the code to be *less* efficient than when leaving it out.

Calling the Clear() methods are unnecessary, and the GC probably wouldn't do that if the Dispose didn't do it...

Share Improve this answer Follow

answered Feb 11, 2009 at 20:32
Arjan Einbu
13.7k ● 2 ● 58 ● 59

4

Apart from its primary use as a way to control the **lifetime** of **system resources** (completely covered by the awesome answer of *Ian*, kudos!), the **IDisposable/using** combo can also be used to **scope the state change of (critical) global resources**: the *console*, the *threads*, the *process*, any *global object* like an *application instance*.

I've written an article about this pattern: http://pragmateek.com/c-scope-your-global-state-changes-with-idisposable-and-the-using-statement/

It illustrates how you can protect some often used global state in a **reusable** and **readable** manner: *console colors*, current *thread culture*, *Excel application object properties*...

Share Improve this answer Follow

answered Jun 15, 2013 at 13:56
Pragmateek
13.3k ● 9 ● 76 ● 111

4

I see a lot of answers have shifted to talk about using IDisposable for both managed and unmanaged resources. I'd suggest this article as one of the best explanations that I've found for how IDisposable should actually be used.

https://www.codeproject.com/Articles/29534/IDisposable-What-Your-Mother-Never-Told-You-About

For the actual question; should you use IDisposable to clean up managed objects that are taking up a lot of memory the short answer would be **no**. The reason is that once your object that is holding the memory goes out of scope it is ready for collection. At that point any referenced child objects are also out of scope and will get collected.

The only real exception to this would be if you have a lot of memory tied up in managed objects and you've blocked that thread waiting for some operation to complete. If those objects where not going to be needed after that call completed then setting those references to null might allow the garbage collector to collect them sooner. But that scenario would represent bad code that needed to be refactored - not a use case of IDisposable.

Share

Improve this answer

Follow

edited Jul 2, 2021 at 1:28

answered Oct 3, 2018 at 17:32

MikeJ
1,359 ● 7 ● 12

1 I did't understood why somehone put -1 at your answer – Sebastian Oscar Lopez Aug 23, 2019 at 13:12

One issue with this that I see is people keep thinking that having a file open with a using statement uses Idisposable. when the using statement finishes they do not close because well the GC will garbage collect call dispose, yada yada and the file will get closed. Trust me it does, but not fast enough. Sometimes that same file needs to be reopened immediately. This is what is currently happening in VS 2019 .Net Core 5.0 – Lawrence Thurman Jul 1, 2021 at 19:47

@LawrenceThurman you seem to be describing people using a disposable without a using statement but on a class that has a finalizer. the GC does not call dispose it calls the finalizer. As an example, FIleStream, if wrapped in a using statement, will close the file when disposed. – MikeJ Jul 2, 2021 at 1:21

@MikeJ Try it out - I assure you I know what I am talking about. Open a file WITH a using statement, modify it close and immediately try to reopen the same file and modify it again. Now do this 30 times in a row. I used to deal with 750,000 jpgs an hour to build build pdfs, and converting the original color jpgs into black and white. jpgs. These Jpgs were pages that were scanned from bills, some had 10 pages. GC is to slow, especially when you have a machine with 256 GB of ram. it collects when the Machine needs more ram, – Lawrence Thurman Jul 7, 2021 at 22:00

it only looks for objects that are not being used when it does look. you need to call file.Close() before the end of the using statement. Oh yeah try it with a database connection too, with real numbers, 800,000 connections, you know like a large bank might use, this is why people use connection pooling. – Lawrence Thurman Jul 7, 2021 at 22:00

3

`IDisposable` is good for unsubscribing from events.

Share

Improve this answer

Follow

edited Nov 22, 2012 at 15:16

bluish
27.2k ● 28 ● 125 ● 184

answered Feb 20, 2012 at 16:35

Adam Speight
732 ● 1 ● 9 ● 22

3

Your given code sample is not a good example for `IDisposable` usage. Dictionary clearing *normally* shouldn't go to the `Dispose` method. Dictionary items will be cleared and disposed when it goes out of scope. `IDisposable` implementation is required to free some memory/handlers that will not release/free even after they out of scope.

The following example shows a good example for IDisposable pattern with some code and comments.

```
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.
    }
}
```

Share Improve this answer Follow

answered May 26, 2017 at 5:41

CharithJ
47.4k ● 20 ● 124 ● 136

2

There are things that the `Dispose()` operation does in the example code that *might* have an effect that would not occur due to a normal GC of the `MyCollection` object.

If the objects referenced by `_theList` or `_theDict` are referred to by other objects, then that `List<>` or `Dictionary<>` object will not be subject to collection but will suddenly have no contents. If there were no Dispose() operation as in the example, those collections would still contain their contents.

Of course, if this were the situation I would call it a broken design - I'm just pointing out (pedantically, I suppose) that the `Dispose()` operation might not be completely redundant, depending on whether there are other uses of the `List<>` or `Dictionary<>` that are not shown in the fragment.

Share Improve this answer Follow

answered Feb 11, 2009 at 18:45

Michael Burr
340k ●52 ●548 ●768

They're private fields, so I think it's fair to assume the OP isn't giving out references to them. – mqp Feb 11, 2009 at 21:20

1) the code fragment is just example code, so I'm just pointing out that there may be a side-effect that is easy to overlook; 2) private fields are often the target of a getter property/method - maybe too much (getter/setters are considered by some people to be a bit of an anti-pattern). – Michael Burr Feb 11, 2009 at 22:27

2

One problem with most discussions of "unmanaged resources" is that they don't really define the term, but seem to imply that it has something to do with unmanaged code. While it is true that many types of unmanaged resources do interface with unmanaged code, thinking of unmanaged resources in such terms isn't helpful.

Instead, one should recognize what all managed resources have in common: they all entail an object asking some outside 'thing' to do something on its behalf, to the detriment of some other 'things', and the other entity agreeing to do so until further notice. If the object were to be abandoned and vanish without a trace, nothing would ever tell that outside 'thing' that it no longer needed to alter its behavior on behalf of the object that no longer existed; consequently, the 'thing's usefulness would be permanently diminished.

An unmanaged resource, then, represents an agreement by some outside 'thing' to alter its behavior on behalf of an object, which would useless impair the usefulness of that outside 'thing' if the object were abandoned and ceased to exist. A managed resource is an object which is the beneficiary of such an agreement, but which has signed up to receive notification if it is abandoned, and which will use such notification to put its affairs in order before it is destroyed.

Share Improve this answer Follow

answered Feb 22, 2012 at 6:38

supercat
80.8k ●9 ●174 ●220

Well, IMO, definition of unmanaged object is clear; *any non-GC object*. – eonil May 26, 2014 at 21:30

1 @Eonil: Unmanaged Object != Unmanaged Resource. Things like events can be implemented entirely using managed objects, but still constitute unmanaged resources because--at least in the case of short-lived objects subscribing to long-lived-objects' events--the GC knows nothing about how to clean them up. – supercat May 27, 2014 at 13:35

2

First of definition. For me unmanaged resource means some class, which implements IDisposable interface or something created with usage of calls to dll. GC doesn't know how to deal with such objects. If class has for example only value types, then I don't consider this class as class with unmanaged resources. For my code I follow next practices:

1. If created by me class uses some unmanaged resources then it means that I should also implement IDisposable interface in order to clean memory.
2. Clean objects as soon as I finished usage of it.
3. In my dispose method I iterate over all IDisposable members of class and call Dispose.
4. In my Dispose method call GC.SuppressFinalize(this) in order to notify garbage collector that my object was already cleaned up. I do it because calling of GC is expensive operation.
5. As additional precaution I try to make possible calling of Dispose() multiple times.
6. Sometime I add private member _disposed and check in method calls did object was cleaned up. And if it was cleaned up then generate ObjectDisposedException
   Following template demonstrates what I described in words as sample of code:

```
public class SomeClass : IDisposable
    {
        /// <summary>
        /// As usually I don't care was object disposed or not
        /// </summary>
        public void SomeMethod()
        {
            if (_disposed)
                throw new ObjectDisposedException("SomeClass instance been
disposed");
        }

        public void Dispose()
        {
            Dispose(true);
        }

        private bool _disposed;

        protected virtual void Dispose(bool disposing)
        {
            if (_disposed)
```

```
                return;
            if (disposing)//we are in the first call
            {
            }
            _disposed = true;
        }
    }
```

Share

Improve this answer

Follow

edited Aug 31, 2015 at 12:18

shA.t
16.9k ●5 ●57 ●119

answered Aug 31, 2015 at 9:45

Yuriy Zaletskyy
5,151 ●5 ●36 ●56

1 "For me unmanaged resource means some class, which implements IDisposable interface or something created with usage of calls to dll." So you are saying that any type which `is IDisposable` should itself be considered an unmanaged resource? That doesn't seem correct. Also if the implmenting type is a pure value type you seem to suggest that it does not need to be disposed. That also seems wrong. – Aluan Haddad Sep 22, 2016 at 3:08

Everybody judges by himself. I don't like to add to mine code something just for the sake of addition. It means if I add IDisposable, it means I've created some kind of functionality that GC can't manage or I suppose it will not be able to manage it's lifetime properly. – Yuriy Zaletskyy Apr 20, 2018 at 22:22

1

The most justifiable use case for disposal of managed resources, is preparation for the GC to reclaim resources that would otherwise never be collected.

A prime example is circular references.

Whilst it's best practice to use patterns that avoid circular references, if you do end up with (for example) a 'child' object that has a reference back to its 'parent', this can stop GC collection of the parent if you just abandon the reference and rely on GC - plus if you have implemented a finalizer, it'll never be called.

The only way round this is to manually break the circular references by setting the Parent references to null on the children.

Implementing IDisposable on parent and children is the best way to do this. When Dispose is called on the Parent, call Dispose on all Children, and in the child Dispose method, set the Parent references to null.

Share Improve this answer Follow

answered Sep 14, 2016 at 5:40

controlbox
542 ●4 ●13

4 For the most part, the GC doesn't work by identifying dead objects, but rather by identifying live ones. After each gc cycle, for each object that has registered for finallization, is stored on the large object heap, or is the target of a live `WeakReference`, the system will check a flag

which indicates that a live rooted reference was found in the last GC cycle, and will either add the object to a queue of objects needing immediate finalization, release the object from the large object heap, or invalidate the weak reference. Circular refs will not keep objects alive if no other refs exist. – supercat Sep 15, 2016 at 2:39

0

I think people are conflating the **PATTERN** of IDisposable with the primary purpose of IDisposable which was meant to help clean up unmanaged resources. We all know this. Some think the pattern has some sort of magical powers that clears memory and frees resources. The PATTERN does NOT do this. But the usage of the pattern with the methods that are implemented DO clear memory and free resources.

The pattern is simply a built in **try{} finally{}** block. Nothing more. Nothing less. So what does that mean? You can create a block of code that lets you do something at the end without having to do extra code for it. It provides a **CUSTOM** block you can use to segment code and scope.

My example:

```
//My way
using (var _ = new Metric("My Test"))
{
    DoSomething();  //You now know all work in your block is being timed.
}

//MS mockup from memory
var sw = new Stopwatch();
sw.Start();
DoSomething();  //something fails? I never get the elapsed time this way
sw.Stop();
```

Metric class

```
    public class Metric : IDisposable
    {
        private string _identifier;
        private DateTime _start;

        public Metric(string identifier)
        {
            _identifier = identifier;
            _start = DateTime.Now;
        }

        public void Dispose()
        {
            Console.WriteLine(_identifier + " - " + (DateTime.Now -
_start).TotalMilliseconds)
        }
    }
```

Share Improve this answer Follow

answered Feb 18, 2023 at 9:20

T Brown
1,533 ● 14 ● 11

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.