What happens if I don't call Dispose on the pen object?

Asked 14 years, 1 month ago Modified 5 years, 3 months ago Viewed 17k times

41

What happens if I don't call `Dispose` on the `pen` object in this code snippet?

```
private void panel_Paint(object sender, PaintEventArgs e)
{
    var pen = Pen(Color.White, 1);
    //Do some drawing
}
```

c# winforms dispose

Share Follow

edited Jun 9, 2012 at 11:20

0lukasz0
3,257 ● 1 ● 26 ● 42

asked Nov 24, 2010 at 14:14

Andreas Brinck
52.4k ● 15 ● 86 ● 114

36 Nothing. That's why you should call it. – user395760 Nov 24, 2010 at 14:15

It seems to me that the MSDN documentation on what the method does is pretty clear. msdn.microsoft.com/en-us/library/… – Captain Sensible Nov 24, 2010 at 14:23

Well, I guess I could have phrased the question what *doesn't* happen if I don't call `Dispose`, but I think you get what I mean? – Andreas Brinck Nov 24, 2010 at 14:24

7 .net doesn't do escape analysis. Thus it doesn't know if the reference survives after `pen` falls out of scope. Thus you have to wait until the GC decides to collect the `Pen` which might be much later. – CodesInChaos Nov 24, 2010 at 14:32

1 If you are too lazy to type `myobj.Dispose()`, you must wrap your code with an [`using` statement](msdn.microsoft.com/en-us/library/yh598w02(v=VS.80).aspx). – Cheng Chen Nov 24, 2010 at 15:00

10 Answers

Sorted by: Highest score (default)

35

A couple of corrections should be made here:

Regarding the answer from Phil Devaney:

> "...Calling Dispose allows you to do deterministic cleanup and is highly recommended."

Actually, calling Dispose() does not deterministically cause a GC collection in .NET - i.e. it does NOT trigger a GC immediately just because you called Dispose(). It only indirectly signals to the GC that the object can be cleaned up during the next GC (for the Generation that the object lives in). In other words, if the object lives in Gen 1 then it wouldn't be disposed of until a Gen 1 collection takes place. One of the only ways (though not the only) that you can programmatically and deterministically cause the GC to perform a collection is by calling GC.Collect(). However, doing so is not recommended since the GC "tunes" itself during runtime by collecting metrics about your memory allocations during runtime for your app. Calling GC.Collect() dumps those metrics and causes the GC to start its "tuning" all over again.

Regarding the answer:

> IDisposable is for disposing **unmanaged** resources. This is the pattern in .NET.

This is incomplete. As the GC is non-deterministic, the Dispose Pattern, (How to properly implement the Dispose pattern), is available so that you can release the resources you are using - managed or unmanaged. It has **nothing** to do with what kind of resources you are releasing. The need for implementing a Finalizer does have to do with what kind of resources you are using - i.e. ONLY implement one if you have non-finalizable (i.e. native) resources. Maybe you are confusing the two. BTW, you should avoid implementing a Finalizer by using the SafeHandle class instead which wraps native resources which are marshaled via P/Invoke or COM Interop. If you do end up implementing a Finalizer, you should **always** implement the Dispose Pattern.

One critical note which I haven't seen anyone mention yet is that if disposable object is created and it has a Finalizer (and you never really know whether they do - and you certainly shouldn't make any assumptions about that), then it will get sent directly to the Finalization Queue and live for at least 1 extra GC collection.

If GC.SuppressFinalize() is not ultimately called, then the finalizer for the object will be called on the next GC. Note that a proper implementation of the Dispose pattern should call GC.SuppressFinalize(). Thus, if you call Dispose() on the object, and it has

implemented the pattern properly, you will avoid execution of the Finalizer. If you don't call Dispose() on an object which has a finalizer, the object will have its Finalizer executed by the GC on the next collection. Why is this bad? The Finalizer thread in the CLR up to and including .NET 4.6 is single-threaded. Imagine what happens if you increase the burden on this thread - your app performance goes to you know where.

Calling Dispose on an object provides for the following:

1. reduce strain on the GC for the process;
2. reduce the app's memory pressure;
3. reduce the chance of an OutOfMemoryException (OOM) if the LOH (Large Object Heap) gets fragmented and the object is on the LOH;
4. Keep the object out of the Finalizable and f-reachable Queues if it has a Finalizer;
5. Make sure your resources (managed and unmanaged) are cleaned up.

Edit: I just noticed that the "all knowing and always correct" MSDN documentation on IDisposable (extreme sarcasm here) actually does say

> The primary use of this interface is to release unmanaged resources

As anyone should know, MSDN is far from correct, never mentions or shows 'best practices', sometimes provides examples that don't compile, etc. It is unfortunate that this is documented in those words. However, I know what they were trying to say: in a perfect world the GC will cleanup all *managed* resources for you (how idealistic); it will not, however cleanup *unmanaged* resources. This is absolutely true. That being said, life is not perfect and neither is any application. **The GC will only cleanup resources that have no rooted-references.** This is mostly where the problem lies.

Among about 15-20 different ways that .NET can "leak" (or not free) memory, the one that would most likely bite you if you don't call Dispose() is the failure to unregister/unhook/unwire/detach event handlers/delegates. If you create an object that has delegates wired to it and you don't call Dispose() (and don't detach the delegates yourself) on it, the GC will still see the object as having rooted references - i.e. the delegates. Thus, the GC will never collect it.

@joren's comment/question below (my reply is too long for a comment):

I have a blog post about the Dispose pattern I recommend to use - (How to properly implement the Dispose pattern). There are times when you should null out references and it never hurts to do so. Actually, doing so does do something before GC runs - it removes the rooted reference to that object. The GC later scans its collection of rooted references and collects those that do not have a rooted reference. Think of this

example when it is good to do so: you have an instance of type "ClassA" - let's call it 'X'. X contains an object of type "ClassB" - let's call this 'Y'. Y implements IDisposable, thus, X should do the same to dispose of Y. Let's assume that X is in Generation 2 or the LOH and Y is in Generation 0 or 1. When Dispose() is called on X and that implementation nulls out the reference to Y, the rooted reference to Y is immediately removed. If a GC happens for Gen 0 or Gen 1, the memory/resources for Y is cleaned up but the memory/resources for X is not since X lives in Gen 2 or the LOH.

Share Follow

edited May 30, 2019 at 22:36

answered Apr 5, 2011 at 16:33

Dave Black

7,999 ● 2 ● 56 ● 44

2 With regard to an apparent contradiction, Jason stated "IDisposable is for disposing unmanaged resources". Its sole purpose is to deterministically cleanup resources - whether or not they are managed. Implementing the Dispose pattern correctly provides you to cleanup your resources. I'm not saying that unmanaged resources are not cleaned up, rather that the purpose of IDisposable is orthogonal to the kind of resources you are disposing. One could surmise from his statement that you do not need to implement IDisposable unless you are using a native resource - which is absolutely untrue. – Dave Black Apr 5, 2011 at 17:21

2 I think it's safe to assume if IDisposable is implemented it has resources that need freeing in a deterministic way. Whether or not there unmanaged resources is an implementation detail. – davidcarr May 29, 2019 at 17:53

3 @davidcarr - I think you completely missed the point of my answer. It is NOT ok to avoid calling Dispose in managed code. I've explained in detail exactly why tit's necessary. Bottom line is this: if a class implements IDisposable, you should be calling it. To say in one case you don't need to call Dispose because of the way a certain class is implemented is bad design - you're relying on internal implementation details which should be treated as black boxes - e.g. MemoryStream, and any other .NET Framework (or other external framework) class. Internal implementations change. – Dave Black May 30, 2019 at 22:31

3 @davidcarr - one of the reasons the IDisposable pattern is so misunderstood is people (like yourself) want to concern themselves with whether the underlying resource is using managed or unmanaged resources - as if that is the deciding factor - it should *not* be the deciding factor. If you stick to the idea of "if it implements IDisposable, then I should call Dispose()", you'll be alright 100% of the time. If people followed this advice, there'd be little, if any, confusion on the pattern. – Dave Black May 30, 2019 at 22:35

2 @davidcarr for the record, I meant no disrespect when I said "people like yourself". I was just being descriptive. – Dave Black Jun 4, 2019 at 19:15

29

The `Pen` will be collected by the GC at some indeterminate point in the future, whether or not you call `Dispose`.

However, any unmanaged resources held by the pen (e.g., a GDI+ handle) will not be cleaned up by the GC. The GC only cleans up managed resources. Calling

`Pen.Dispose` allows you to ensure that these unmanaged resources are cleaned up in a timely manner and that you aren't leaking resources.

Now, if the `Pen` has a finalizer and that finalizer cleans up the unmanaged resources then those said unmanaged resources will be cleaned up when the `Pen` is garbage collected. But the point is that:

1. You should call `Dispose` explicitly so that you release your unmanaged resources, and
2. You shouldn't need to worry about the implementation detail of if there is a finalizer and it cleans up the unmanaged resources.

`Pen` implements `IDisposable`. `IDisposable` is for disposing unmanaged resources. This is the pattern in .NET.

For previous comments on the this topic, please see this answer.

Share Follow

edited May 23, 2017 at 12:17

Community Bot

1 ● 1

answered Nov 24, 2010 at 14:31

jason

241k ● 35 ● 434 ● 530

2 This answer is incomplete and slightly misleading. I've described why in my post below. – Dave Black Jun 7, 2012 at 15:06

4 -1 for the assertion that the GC won't clean up unmanaged resources. If the IDisposable implementation does this properly, it will work just fine from the finalization thread, but just rather later. – Dominic Cronin Jun 20, 2012 at 21:53

3 The GC does not release unmanaged memory or resources - regardless of how Dispose() is implemented. What it *can* do is to allow you to call Marshal.ReleaseComObject() and implement a Finalizer which will be called if GC.SuppressFinalize() is not called. Just because the cleanup of unmanaged memory happens during Finalization or in the Dispose implementation, does NOT mean the GC is the one doing the release. The GC is not responsible for cleaning up unmanaged resources - if the programmer never does so, then the memory will be leaked. – Dave Black Jun 21, 2012 at 14:13

@DaveBlack The MSDN Docs (msdn.microsoft.com/en-us/library/...) on Pen disagree with you there. "Always call Dispose before you release your last reference to the Pen. Otherwise, the resources it is using will not be freed until the garbage collector calls the Pen object's Finalize method." Unless badly written, I assume "the resources" are all resources, also unmanaged ones. So the GC indeed does release unmanaged resources in this case. – Tom Aug 9, 2014 at 12:06

@Tom - the GC knows nothing directly about unmanaged resources: not how to allocate them, and not how to release them. What the language does is provide you a means to handle it yourself via Dispose with a Finalizer. Technically speaking the Pen class is not an unmanaged resource; rather it is a managed wrapper over an unmanaged resource. The underlying code in the Pen class implements a Finalizer. The Finalizer in that class does the necessary task of cleanup so when dealing with managed objects, the resources are not typically "leaked" (with very few exceptions). – Dave Black Oct 14, 2016 at 15:00

13

The underlying GDI+ pen handle will not be released until some indeterminate time in the future i.e. when the Pen object is garbage collected and the object's finalizer is called. This might not be until the process terminates, or it might be earlier, but the point is its non-deterministic. Calling Dispose allows you to do deterministic cleanup and is highly recommended.

Share Follow

answered Nov 24, 2010 at 14:28

Phil Devaney

18k ● 6 ● 42 ● 33

This answer is slightly incorrect. I've described why in my post below. – Dave Black Jun 20, 2012 at 21:34

2

The total amount of .Net memory in use is the .Net part + all 'external' data in use. OS objects, open files, database and network connections all take some resources that are not purely .Net objects.

Graphics uses Pens and other objects wich are actually OS objects that are 'quite' expensive to keep around. (You can swap your Pen for a 1000x1000 bitmap file). These OS objects only get removed from the OS memory once you call a specific cleanup function. The Pen and Bitmap Dispose functions do this for you immediatly when you call them.

If you don't call Dispose the garbage collector will come to clean them up 'somewhere in the future*'. (It will actually call the destructor/finalize code that probably calls Dispose())

*on a machine with infinite memory (or more then 1GB) somewhere in the future can be very far into the future. On a machine doing nothing it can be easily longer then 30 minutes to clean up that huge bitmap or very small pen.

Share Follow

answered Nov 24, 2010 at 15:25

CodingBarfield

3,398 ● 2 ● 28 ● 55

2

If you really want to know how bad it is when you don't call Dispose on graphics objects you can use the CLR Profiler, available free for the download here. In the installation folder (defaults to C:\CLRProfiler) is CLRProfiler.doc which has a nice example of what happens when you don't call Dispose on a Brush object. It is very enlightening. The short version is that graphics objects take up a larger chunk of memory than you might expect and they can hang around for a long time unless you call Dispose on them. Once the objects are no longer in use the system will, eventually, clean them up, but that process takes up more CPU time than if you had just called Dispose when you were finished with the objects. You may also want to read up on using IDisposable here and here.

Share Follow

edited Aug 28, 2019 at 20:50

answered Nov 24, 2010 at 14:50

Bill W

1,478 ● 1 ● 20 ● 29

1

It will keep the resources until the garbage collector cleans it up

Share Follow

answered Nov 24, 2010 at 14:15

m.edmondson

30.9k ● 29 ● 126 ● 211

1 Wong! The handle will stay until process is terminated. – Aliostad Nov 24, 2010 at 14:16

2 the pens finalizer should clean up the resources if it runs msdn.microsoft.com/en-us/library/... of course there is no guarantee it will run – jk. Nov 24, 2010 at 14:19

4 @Mitch, @Aliostad, @jk - All objects that extend `Component` implement a finalizer which will call `Dispose`. The finalizer is non-deterministic which means you can't predict when it will run but it will always run eventually. If another finalizer blocks indefinitely or the process is killed then of course it will not run. – ChaosPandion Nov 24, 2010 at 14:26

@ChaosPandion: And a normal finalizer won't be called when the AppDomain is unloaded after an uncaught exception. You need a critical finalizer for that. But I'd guess that a Pen uses a safehandle and thus critical finalization, but I didn't verify that. – CodesInChaos Nov 24, 2010 at 14:29

1

Depends if it implements finalizer and it calls the Dispose on its finalize method. If so, handle will be released at GC.

if not, handle will stay around until process is terminated.

Share Follow

edited Nov 24, 2010 at 14:22

answered Nov 24, 2010 at 14:17

Aliostad
81.6k ● 21 ● 162 ● 209

"Always call Dispose before you release your last reference to the Pen. Otherwise, the resources it is using will not be freed until the garbage collector calls the Pen object's Finalize method." msdn.microsoft.com/en-us/library/... – J D Jun 7, 2012 at 17:44

0

With graphic stuff it can be very bad.

Open the Windows Task Manager. Click "choose columns" and choose column called "GDI Objects".

If you don't dispose certain graphic objects, this number will keep raising and raising.

In older versions of Windows this can crash the whole application (limit was 10000 as far as I remember), not sure about Vista/7 though but it's still a bad thing.

Share Follow

answered Nov 24, 2010 at 14:28

Shadow Wizard
66.3k ● 26 ● 146 ● 209

Does `GDI+` use `GDI` objects? Afaik WinForms uses GDI+ and not GDI for most of it's functionality. – CodesInChaos Nov 24, 2010 at 14:34

@CodeInChaos - quick test confirmed that having Pen objects in the OnPaint event raise the value of GDI Objects, so probably it consists of both GDI and GDI+ objects. – Shadow Wizard Nov 24, 2010 at 14:38

Suppose a program will need to use pens of a lot of different colors. What would be the tradeoffs between (1) having multiple controls each produce and keep a set of pens, `Dispose`ing of them when it was itself `Dispose`d; (2) creating pens always on demand and never caching them; (3) having a global `Dictionary<Color, Pen>` which lasts as long as the application, or (4) having a global `Dictionary<Color, WeakReference>` which would hold a `WeakReference` to each pen? – supercat Jun 11, 2012 at 21:35

@supercat recently I was faced with similar problem in one of my projects, ended up with #3 - global dictionary created once and disposed when the application is destroyed. It works nice and fast but can't say it's the official or best way. – Shadow Wizard Jun 12, 2012 at 6:21

@ShadowWizard: It seems analogous to interning strings. The `WeakReference` approach would seem to have some merit also in cases where a control would hold a graphic object in a field, but many controls might want to hold the same one (so the number of controls holding a particular object might vary from hundreds to zero). – supercat Jun 12, 2012 at 13:34

-1

the garbage collector will collect it anyway BUT it matters WHEN: if you dont call dispose on an object that you dont use it will live longer in memory and gets promoted to higher generations meaning that collecting it has a higher cost.

Share Follow

answered Nov 24, 2010 at 14:17

Ali Tarhini

5,348 ● 6 ● 44 ● 66

-3

in back of my mind first idea came to the surface is that this object will be disposed as soon as the method finishes execution!, i dont know where did i got this info!, is it right?

Share Follow

answered Nov 24, 2010 at 14:22

Rami Alshareef

7,140 ● 13 ● 49 ● 77

1 It does not happen like that in C# (or .NET in general for that matter) – Bryan Nov 24, 2010 at 14:27