IDisposable Best Practices for C# Developers

Course Overview

Course Overview

Hey, how you doing? My name's Elton, and this is IDisposable Best Practices for C# Developers. I'm a Microsoft Azure MVP and a freelance consultant, and I've been using C# since the first release of .NET in 2002. That's nearly 20 years. And in all that experience, some of the most common problems I've seen are from developers not having a good understanding of IDisposable. This course, IDisposable Best Practices for C# Developers, will help you avoid those problems and write code which is high performing, low on resources, and free of hard-to-find functional issues. First, you'll learn what the IDisposable interface is, what it's for, and what kind of problems you get if you don't use it correctly. And these aren't edged case issues. You'll see problems with everyday objects like SQL connections and file streams. Next, we'll have a closer look at memory management in .NET, understanding the garbage collector and how it tidies up managed and unmanaged resources. You'll learn how to use the tools in Visual Studio to diagnose problems, and you'll see how to implement IDisposable correctly to avoid them. Then, you'll see how IDisposable fits with modern .NET architecture where you use dependency injection to manage object lifetimes. And you'll learn how you need to work with common objects like DbContexts and HTTP types in that architecture. By the end of the course, you'll have a good understanding of all the best practices for working with IDisposable, and we'll finish up with guidance of what you need to do every day and what you need to remember for less frequent situations. So join me and take your .NET apps to the next level of reliability with IDisposable Best Practices for C# Developers right here, on Pluralsight.

Introducing IDisposable

Understanding IDisposable

.NET is a managed platform. That means your code executes in a runtime, which takes care of managing resources like memory and network connections. So you'll never get a situation where you run out of resources and your application crashes, right? Well, not quite. Hey, how you doing? I'm Elton, and this is IDisposable Best Practices for C# Developers here, on Pluralsight. This course will

teach you all about the IDisposable interface, how it helps you manage resources efficiently and why you need to use it in your .NET applications. A good understanding of IDisposable is critical for writing high-quality apps that run efficiently and play nicely in a world with finite computing resources that we all use, like memory, database connections, and file handles. It's the one part of writing C# code where .NET asks for your help. Everything else in the platform is there to make it easy for you to solve problems. But IDisposable is where .NET has its own problem, and it needs your help to solve it. And we're talking about all versions of .NET here from the original .NET Framework that only ran on Windows to the cross-platform evolution of .NET Core and the latest releases, .NET 5 and 6. The problem that .NET has is resource management. And to start with, we'll look at memory. Any program on any platform needs to allocate memory while it's doing work. Typically, there's a stack where small objects go, and larger objects go on the heap with a pointer in the stack. The heap grows as the program uses more memory, and then it shrinks when memory is released. When the application ends, all the memory gets made available again. But while the program is running, it needs to make sure it releases memory that it isn't using. In some platforms, you have to do that manually with method calls to allocate and free memory, and it's easy to get that wrong and have a program with a memory leak where it just uses more and more memory until eventually it crashes because there's no available memory left on the machine. In .NET, we have it easy because memory management is taken care of in the platform. All the .NET runtimes have a garbage collector, the GC, that runs in the background, finds any objects that the app is no longer using, and releases those objects, freeing up any resources that they have allocated. But it can only do that if it knows the object isn't being used, and that's where IDisposable comes in. Your app could be using expensive resources, like database connections. And if you don't use IDisposable correctly, then you're not taking control of clean up and you're relying on the garbage collector to clean them up for you. The GC decides when to trigger a collection, so your expensive objects may live a lot longer than you intended. And if they have lingering references, they'll never get cleaned up at all. The garbage collector can let objects live even when they're out of scope and you can't reach them in your code. So in the simplest app, you could end up with thousands of objects you're no longer using still in memory. So the memory profile of your app grows and grows until there's no memory left and your application crashes. And those objects that aren't disposed could be hogging other resources, file handles or network connections. So even though you finish using them, they're still alive in memory and will keep hold of those resources until the owner frees them up or your application ends. And it's not just about resource management. The objects that are still alive may cause functional failures too. If you have event handlers registered for objects which you've finished with, but you don't remove the handlers and dispose the object, those handlers will keep on firing, which can make your app behave strangely. And some implementers take advantage of IDisposable for internal optimizations. If they know they should be notified when their object is finished with, they may

code it so the results of work stay in memory until the point where it gets disposed and then they get committed. If you don't use that IDisposable object the way it was meant to be used, you can find your app only writes half of the content of an output file or produces an encrypted string which is incomplete and corrupt. By the end of this course, you'll understand why these things happen and how IDisposable helps you to avoid them. Using IDisposable is straightforward. And in this course, you'll learn the valuable best practices that you'll need in all of your applications and that are easy to apply. In this first module, we'll look at the IDisposable interface, how it's defined and why it is used. Then we'll look at how you use IDisposable objects correctly and how you can implement IDisposable yourself. Next, we'll walk through a very simple demo solution that uses some universal .NET features, working with the file system, a SQL database, and an HTTP service. The app doesn't use IDisposable correctly, and by the end of this module, we'll see that by ignoring IDisposable, our app has got a lot of problems. We'll use the latest version of .NET for all these demos, but the techniques apply to all versions of .NET, and they apply equally for all types of .NET applications, desktop and server, running on Windows or Mac or Linux. In the rest of the course, we'll go about fixing our demo application by getting a good understanding of IDisposable and how to use it correctly. We'll look at what actually happens when the garbage collector runs to understand why this is so important, and we'll see what happens when you don't dispose, fixing each of the problems that we find just by using IDisposable correctly. This is a really focused course with the aim of making sure you know how and when to use IDisposable so you can build your applications confident that there are no lurking hard-to-find defects. So let's get straight on and start working with IDisposable.

Using Disposable Objects

IDisposable is a very simple interface. The simplest interface you can write in C# looks like this. And to implement IDoNothing, you don't need to do anything. You'll hear this called a marker interface, which is a pattern for physically associating classes which aren't logically linked. Sometimes it's an antipattern, but sometimes it's very useful. For a real interface which does something, this is as simple as it gets. To implement IDoOneThing, you need to write one method which takes no arguments and returns nothing. And IDisposable is just about as simple as that. It has one method, Dispose, with no arguments and no return. The difference between IDon'tDoAnything, IDoOneThing, and IDisposable is that IDisposable is defined in the System namespace, in the System.RuntimeAssembly. And that means any .NET class in the base library or any NuGet package or in your own custom code can implement IDisposable, and you need to know if you're using an instance of a class which is disposable so that you can use it properly. IDisposable is not tied to the GC mechanism. The garbage collector does not call dispose, and it doesn't care if the objects it's

looking at are disposable. That's an important point that we'll look at more as we go through the course. So if it's not about garbage collection, this is what IDisposable is for. It provides a mechanism for releasing unmanaged resources, and that dispose method is for your own application code to free those unmanaged resources. So if you implement IDisposable, you're telling users of your class that you have unmanaged resources, and you expect them to tell you when they're finished using an instance of your class, and then you'll take care of cleaning up those unmanaged resources. And that's the real problem with IDisposable. We've seen that it's a simple interface definition, and we'll soon see that it's easy to implement and use. But the documentation consistently refers to unmanaged resources when it talks about IDisposable, so it's very easy to think that it doesn't concern you. When you're working with a .NET solution, you may think that everything you're doing is managed code so you don't need to worry about IDisposable. Unmanaged code is where you need to call something outside of .NET, and it's pretty rare in most applications. I've worked with .NET since 2002 on hundreds of apps and tens of millions of lines of code, and I've only worked explicitly with unmanaged resources a handful of times, runtime-callable wrappers in the old days on Windows, which you used to work with COM components, and platform invoke, which you used to call native code which is cross-platform. Those explicit uses of unmanaged code are pretty obvious. You'll see DllImport attributes, or you'll be using types like IntPtr. And if you're working in that space, then you need to understand IDisposable and finalizers to free up those resources which .NET doesn't manage for you. But whenever our .NET apps interact with the outside world, we're leaving the managed sandbox, and so pretty much every application that talks to a file or database or a REST API is using unmanaged resources, but they're wrapped inside managed code. If you connect to a SQL Server database, you use managed code in the System.Data assembly, but the actual connection to SQL Server isn't managed code. When you fetch data, you leave your application domain and the managed space. The same when you call an HTTP service, whether it's local or remote, or write to a file on the local machine or across the network. Your application can be using all managed code in System and Microsoft assemblies, but those implementations are ultimately reaching outside of the app so there are unmanaged resources somewhere. And that's why the key classes in those scenarios, they all implement IDisposable, and they expect you to tell them when you're finished using them so they can clean up those unmanaged resources. So IDisposable isn't just about unmanaged resources that you use explicitly. If you use any class that implements IDisposable, you need to use it correctly, even though you're building a fully managed application and you're not working with unmanaged code at all. Now that we've seen IDisposable doesn't only apply when you're using a legacy COM DLL on Windows or a native C library on Linux, let's see how it works. If you create an instance of a class which implements IDisposable, then you should actively manage the lifetime of that object. Here's my custom class that implements IDisposable. And if I want to use an instance of that class, I should declare it inside a using block. So this is the pattern. Using blocks are for

working with IDisposable objects. When the block ends, dispose gets called on the object automatically, and then it can free up whatever resources it had allocated without you having to get involved with them. And this is the standard way to work with disposable objects, declaring them inside the using statement and working with them inside the block. It has the advantage that it's very clear to read and to see when the object goes out of scope and it's going to be disposed. There is an alternative syntax which does away with the braces, and that's useful if you have lots of nested using statements. But I don't like it for a single using because I think it's less clear to the reader, but you can use whichever you prefer. Functionally, they all do the same thing. Using is a C# construct that the compiler actually renders as a try finally block, calling dispose on the object inside the finally. So the using block is equivalent to creating the object, using it inside the try block, and then calling dispose inside the finally block. So I've said we should have using statements, but what happens if you don't call dispose or if you don't use a using block? You won't get a compile time error. You may not get any runtime errors. But you have got a problem in your code, which may just be biding it's time. The nature of the problem depends on what the class is doing and how hard the app is working. But we're not telling the object to clean up its resources, so we're not releasing those resources as soon as we finish with them. Depending on the situation, that may mean the resources take longer to clean up as they wait for the next garbage collection event, or the object may still have references pointing to it. So the resources will never be released until your application exits or until it throws an OutOfMemoryException or another error. And that's enough of the theory. Next, we'll have a demo is showing you what goes wrong if you don't use IDisposable in practice.

Demo: Working with Disposable Objects

In this demo, we'll work with a common disposable object, the SqlConnection class in the Microsoft.Data NuGet package, which lets us connect to SQL Server databases, and it implements IDisposable. We'll use instances of the class with and without disposing them, and we'll see what happens to our database connections and to our managed code. If you've seen my other courses on Pluralsight, you'll know that I like to use markdown for my demos, and all the documentation is in the download folder for the course. This one is slightly different because we'll be spending most of our time in Visual Studio, but I'll still use VS Code for the notes. You can follow along with my demos here. All you need is a SQL Server database, and I'm using Docker to run my database in a container, so you don't even need SQL Server installed on your machine. I've already started my database container, so I'll just checked that is still running. That's looking good, and it's publishing to the standard SQL Server ports, so I can use it from within Visual Studio. And that's all the setup I need, so now I'll switch to the full Visual Studio. In my SqlConsumer solution, I've got a really simple class called DatabaseState, which I can use to run some SQL

queries. This class keeps an instance of the SqlConnection class as a local field, and that's from the standard Microsoft.Data.SqlClient NuGet package. In the GetDate method down here, it instantiates that SqlConnection object if it's currently null, then it opens the connection and uses it to fire off a getdate query. Keeping the SqlConnection object alive like this is actually a flawed attempt at optimization, which you do see a lot in real projects. The thinking is it's relatively expensive to get my SqlConnection, so once it's opened I'll keep it and keep reusing it. And that may give your app a small optimization when it runs, but it means it doesn't scale well. If you run multiple instances of this code concurrently, they all keep hold of connections, and eventually they'll starve the connection pool. But this class does implement IDisposable, and in the Dispose method down here, it closes the connection and then calls Dispose on it to free up the database resources when this object is no longer used. So as long as we use the DatabaseState class correctly, we're not going to see any problems. The Dispose method also writes a debug log with a timestamp when it gets called so we can see when these resources get freed up. I have a set of unit tests here to work with that class, and in the simplest test, I'm doing things properly with a using block. So I create that DatabaseState object inside my using statement, I call the GetDate method inside the using block, and then when this block exits, my object's going to get disposed. The Wait method, that just waits for a few seconds and writes another timestamp, and that will help us work out the sequence of events when we run these tests. So I'm going to execute this test in debug mode. And when that's finished, in the Output window, we can see all of the log entries from the test runner, and then here's my code. There's the response from the GetDate call, and then immediately after that, there's my Disposing log, and that's because I had a using block when I worked with the object. Then there's the Wait log a few seconds later, and after that the test ends, so the app domain gets unloaded. I have the same test here without the using block. So this test just instantiates a new object, calls the GetDate method, and then waits for 5 seconds, so there is no Dispose call within this test. And when I run this in debug, we'll see a big difference. The test still passes, the GetDate output is still there, but there's no Dispose call. There's the Wait log, but even when the app domain exits, Dispose never got called. So when I don't call Dispose, my test here is holding onto a connection from the SQL Server connection pool for all the time during that wait period, and connections are a finite resource. The connection will get closed when the application exits, but if this was a long-running server app, then that connection would stay allocated. But still the test passes, so it can't be that big of an issue, right? Well, next we'll run the same code in a loop and see where not disposing gets us.

Demo: Resource Exhaustion from Not Disposing

In this clip, we'll carry on with the simple SqlConnection tests and have a look at the consequences of not disposing when you're running at higher load. We'll see

that freeing up resources isn't just a nice to have, and even in a simple application you can exhaust resources and cause exceptions. So back to the test suite with that same DatabaseState class. This test fixture makes the GetDate call inside a loop, running for 1000 iterations. Each iteration has a using statement, so every object will be disposed after it's used. I've got the alternative using syntax here without the curly braces, but remember it's functionally exactly the same. So then I'll run this test in debug, and we can check the output. So we can see every GetDate log is followed by Disposing log, so every object is getting disposed. This number is the hash code for the SqlConnection object, and it's different every time, so we can see that these are definitely different objects. And now here's the big upset coming with this code. So this test works in exactly the same way with a loop with 1000 iterations. In each iteration, we create a new DatabaseState object, and we call the GetDate method. And the big difference is we don't have a using statement. This DatabaseState object is newed up inside the loop, so it's a local variable, and you might think it will still get cleaned up at the end of each iteration, but we'll run this and we'll see. So it starts off well. We get lots of GetDate logs. There are no Dispose logs, of course, because we're not disposing the objects. But after a while, bang, we get this InvalidOperationException. We can see this exception comes from the SqlConnection object, and it's a timeout from trying to connect to SQL Server. And that happens because all the connections in the connection pool are already being used by those objects that ran earlier in the loop. They made the connection, ran the call, but never got disposed, so they still have an open connection. And we'll get to the point where there are no free connections in the pool, and then we get this error. In my connection string for the tests, I specify an application name, which means in SQL Server I can query the sysprocesses table to find out all the open connections with that program name, and I'll query that when I run my next test. So this test fixture here, it runs that same loop, but inside a try/catch block. So I've got my 1000 iteration loop without a using statement, I call the Wait statement after the for loop completes, but I'll put my breakpoint in this catch block here, and then that's where we'll execute our SQL statement. So if I debug this test here, we've hit the same exception as before with the timeout trying to connect SQL Server. If I go to my SQL statement and run this, we can see that I've got 100 open connections to SQL Server. Now if I go back to my test here and step over the next line, which forces a garbage collection, and then I go back and run my query again, I still have 100 open connections. And it's only when I let the test carry on running. And I'll check the output window here, and I can see the program has exited, which unloads the managed app domain and frees up all the memory. And now if I run my query, my connections have all been closed. So why did the app break when there were 100 open connections? Well, that's the default connection pool size in SQL Server. Instead of using the defaults, I can specify my own value, and that's what I have in this second connection string here. I'm specifying my own maximum pool size of 400, and in both my connection strings, I'm using a very short connection timeout of 2 seconds so my tests fail quickly and we can see what's happening. So I'll change my test to use

this db-max connection string instead, and I'll run that same test with a loop. I'll put an extra breakpoint here, and up at the top I'll change the connection string to use. Now if I run this test in debug, this time my breakpoint hits at the Wait call. So the for loop has executed correctly. If I check SQL Server over here, I've got 390 open connections. So even though my for loop has finished and I've still got a lot of connections open, I didn't hit that magic 400 connection limit, which is the pool size, so my application didn't break. Now that doesn't seem to make sense because the pool size is less than 1000, and we expect to exhaust that when we take a connection from the pool and don't release it in our loop of 1000 iterations. The reason that this does work is down to finalizers, which we'll cover in the next module. But for now, we can say that the garbage collector is doing some cleanup for us, so connections are being made available again. But because we're not calling IDisposable, we don't know when that will happen. And the point when we exhaust the pool size before the garbage collector can free up the used connections depends on the speed of the machine and the resources available at runtime. If you run this test yourself, you may find the new version still errors, so by not using IDisposable, we're giving our code the potential to fail, but we don't know when that will happen. It will be different in different environments. So if I switch to the smaller connection pool size again, which means that it's less likely that the garbage collector will run before I exhaust the pool, and then I'll run the loop test again without the using statement, I get the same error again telling me the timeout has expired because the connection pool has been exhausted. But if I run that same size loop with the using statement, then I can be confident it's going to pass no matter what the pool size is because I'm not relying on a separate process to do the cleaning up for me. In the next clip, we'll recap what we've seen in these demos and talk about our first IDisposable best practice.

Best Practice #1

So that's the basics of IDisposable, and we can already see that it's easy to use and it's dangerous when you don't use it. I have my DatabaseState class, which implements IDisposable, releasing the SqlConnection object inside its own Dispose method. Because it implements IDisposable, if I want to use an instance of this class for a short time, then I can work with that instance inside a using block. The syntax for using is really simple, create the object as the argument to the using statement, do what you need to inside the block, and when the block ends, Dispose gets called on the object so it can immediately release all of its resources. Implementing IDisposable is pretty simple too. You will have local fields which are themselves disposable, and in your Dispose method you call their Dispose method. So we set up a chain where each object in the graph releases its resources until ultimately whoever owns the unmanaged resources can release them. This is actually not a best practice implementation of IDisposable. We'll see later there are a couple more things we need to do for a proper implementation,

but this is okay as a starting point. One thing to note, if you're not familiar with the using statement, you can return from a method inside the using block and your code still operates correctly. We did that inside the disposable class with the SqlCommand object, which is itself disposable. Running the execute call inside a using block and returning still in that block is perfectly valid, and it still ensures Dispose gets called after the method returns. So in that quick demo, we've seen the first of the best practices. If you're making use of an instance of a class which implements IDisposable, call Dispose as soon as you're finished with it. It doesn't matter if you think your app or the class you're using isn't dealing with unmanaged resources. If it implements IDisposable, it does it for a reason, so you need to use it the way it expects to be used. Working with objects in a using block is the simplest way to follow that practice, and it's also semantically very clear for anyone reading your code. I'm using this object for a short period, and now I'm done. There are some rare cases where the using block isn't a good fit. Maybe an object could throw an exception in the Dispose method and you want to manage that. The alternative here is to use a try block and call Dispose in the finally yourself. That still makes it clear that you always want Dispose to be called. But if you have a temporary reference to an IDisposable object, then make sure you call Dispose on it as soon as you're done, whether you do it explicitly or inside a using block, and that way, you'll be making sure that your app releases resources as soon as it can. The other best practices will unfold in the course. But before we go too much further, it's going to be useful to look at some real code. Over the next couple of modules, we're going to look at a poorly built application, which has been implemented in complete ignorance of IDisposable. It's a simple app for counting the number of words in a book. At the moment, it's a prototype, and what we have is an HTTP service that counts the number of words in a line of text and a console application which monitors a folder. When you drop a text file into the folder, the console app picks it up, records some data about the file in a SQL database, then calls the web service for each line of text. It does that using parallel tasks. So for any one file, we can be calling the service for dozens of lines concurrently. Each task writes the number of words for the line in the database. And then when all the tasks complete, the app updates the database with the total word count for the book. So that's a pretty simple architecture. Obviously, before we go to market, we'll be running the whole thing in the cloud with a RESTful API, saving files to blob-storage, kicking off processing with messages sent to a queue, running the whole thing in Kubernetes, and we'll deliver a mobile app that lets users upload a text file and get told how many words are in that book. I can't think of anyone who doesn't need that application, but for this course, we're going to keep it simple and stick with the console prototype. If I add some overlays to this architecture, we can see that our console application is managed code, but there's an awful lot going on outside the application, which is not managed code. Actually, the service is a managed .NET application too, but we're not going to focus on that service. And as far as my application is concerned, it could be a REST API written in Java or a gRPC service written in Go, it's all HTTP calls under

the hood. And as we have lines going from the managed bits to the unmanaged bits, that means at some point our app is dealing with unmanaged resources. And we already know that because of that, it should be using IDisposable objects correctly or it's going to hit problems. Next, we'll have a demo showing how the app works and think through some of the problems that it might have.

Demo: Consuming Resources without Disposing

Let's see how this prototype app gets along without IDisposable. We only have one feature, which is to calculate the number of words in a text file and store the totals in a SQL database. We'll run the app, drop some files in, and see what happens so we can get a feel for the flow and prove that the thing actually works. Then I'll walk you through the code and flag up where we should be thinking about IDisposable, and I'll show you how easy it is to break our application just because it doesn't follow IDisposable best practices. The setup for this demo is pretty similar. I have a Docker image prepared which runs SQL server, and it has my database schema already deployed. If you want to see how that's done, it's all in the Docker folder in the course downloads. But if you don't care, you can just run the container, and you'll have that database running locally. I'll switch to the solution in Visual Studio and start it running, which will run the REST API and the console application. So here's my app. It's doing nothing at the moment, but if I hit S, then it starts monitoring the books folder on my drive for new files. I'll shift the windows around a bit, open a terminal in Visual Studio, and get ready to copy in some files. I'm using Project Gutenberg for sample files, and pg98 is an ASCII text file with the complete text of a Tale of Two Cities. I also have an excerpt file and a file which just has the contents page, so we can try with files of different sizes. I'll start by copying the smallest file to that drops folder, and we'll see in my Application window, there's a whole load of Console output, which is telling me when each line in the file gets read and sent off to the API for counting. That's finished. It just took a few seconds, and I can run some SQL queries over my database. I'm using the SQL Server VS Code extension, but you can use any sqlClient. The BookFeed table records the file being processed, and that shows the total size, and then the BookLine table, that shows the count for each line, and all the data is there. So this is looking good. But if I open my terminal again and try with a slightly larger file, it's going to take a little bit longer to complete, but the application still runs, the count is going to complete successfully, and now I'll have two files in my BookFeed table, and there we go. So let's check out the code. I'm not interested in the web API at this point, just with the console application, which is interacting with the managed world. All the work is done in the Program class, which starts with some setup to load application configuration and to create the directories the application is going to use. When I hit s in the console, it runs this start method, which uses this FolderWatcher class. Calling start passing the path of the folder to watch and an action to execute when any text file gets created. That FolderWatcher class, it

creates a FileSystemWatcher, which is from the system.io namespace, and effectively, this class is just a wrapper that executes the given action whenever the FileSystemWatcher's created event gets fired, and it starts the watcher by enabling events. You often see the FileSystemWatcher getting wrapped like this precisely for this bit of code. If you want to be notified when a file is created, the watcher can be a little bit too zealous. So the event fires when a file write starts and with large files, or files being copied across a slow network, the write could still be happening when the event fires. So when your handler tries to do something with the file, it finds the file is locked. There are better ways of dealing with that, but here, I just have a Sleep to let the file write finish. Back in the program, when we see a new file, we run this ProcessFile method, and we start by copying the input file to an archive location using this FileArchiver class, and all that class does is open streams for the input and the output files and then copy the input to the output. So very simple, three lines of code, not much to go wrong here. Next, we save the file details to the book database using this sqlClient class, which has a very basic approach, opening a SqlConnection, creating a command, and firing off an insert statement. And then back in the Program, we actually count the words in the file. We read all the lines in the file, and for each line, we run a task to call the GetWordCount method, and what that method does, it uses the apiClient to get the count of words in that line and then uses the sqlClient class to save the number of words in that line. So we've already seen the sqlClient, which uses raw SQL strings, and we'll see the apiClient is pretty simple too. It's just a wrapper around the GeneratedService class from Visual Studio, which creates an instance of the Client class and then returns the response from the service call. Everything uses async methods, which get awaited so the runtime can make efficient use of threads, and the cancellationToken stuff is to let me deal gracefully with any failures. If the WordCount service isn't available, I'll find that out during task execution, and there's no point in sending off another 40,000 tasks to a dead service. So I pass the CancellationTokenSource to each task, and GetWordCount only proceeds if the token isn't set, and if any call to the service errors, then the token gets canceled so all the outstanding tasks get canceled, and I won't try and create anymore. So once all the tasks are created, we wait for them all to finish, and if they all finish successfully, we sum all of their results to get the total WordCount for the book, and then we use the sqlClient again to save that total to the BookFeed table. It's fairly straightforward, and I'm using parallel tasks to get maximum performance, but I don't call Dispose anywhere, I don't have any using blocks, and I don't implement IDisposable in any of my classes, but it's not a problem because the app actually works. So is IDisposable just a nice-to-have that you only really need in enterprise scenarios? Well, I expect you're thinking no. And in the next demo clip, we'll see if you're right.

Demo: Functional Defects from Not Disposing

In this clip, we'll carry on with the WordCounter demo application and see that we really can't get by without IDisposable. We'll find a few nasty problems using up all of the available resources and causing bugs in the application. I've cleared my database and restarted the app, so we're running from fresh. I'll start the FolderWatcher in the console application, and then I'll drop in a full-size copy of a Tale of Two Cities, which is pg98.txt. We'll see the app starts in the same way, processes a whole bunch of lines, and then we're going to get a whole lot of exceptions. And that's the same error we saw in the first demo. The SqlConnection pool has been starved because the app didn't call Dispose on the SqlConnection or the SqlCommand. If we look at the code here in my connection string, I've set my pool size to 250, so the first set of tasks work correctly, and we can see that in the database. I've got 366 lines in there, so 366 tasks are processed correctly. Not exactly 250 because the garbage collector is freeing up some of those resources. But mostly, I've got un disposed connections hogging the connection pool. Even though the SqlClient is a local variable in a task which runs on a separate thread and then ends, so all those SqlClients are no longer in scope and they can't possibly be referenced from my application, they're still each keeping hold of their SQL connection. The point where this application crashes depends on how powerful your processor is. So long as the number of concurrent tasks being executed by .NET is less than the size of the connection pool, you won't see this error. So ironically, if you have a slower machine, this application works better because it will process larger files without crashing. That means you could be happily running this in dev and test environments, sign it all off, and when you get to production it goes bang because you're running on a 32 processor core machine, and the number of tasks .NET runs concurrently will exceed the size of the connection pool. There are other problems lurking in my application too. I'll go back and clear down my database again. So now we're starting with an empty state, and I'll run up a new instance of my application. So imagine this app is running as a background service or in a Docker container instead of in a console, and it's been set up to allow pausing and resuming. I can simulate that by hitting S to start with watcher process and then hitting S again for the resume. I get my listening on output twice, so I'm sure you can guess what's going to happen. Now when I drop a file, and I can use a short file here, I get a lot of duplicated console output. And in my database, I'm going to have two sets of results for the same file. So my event handler, for when files get created, has fired for each instance of my FileSystemWatcher. And if we check the code in the FolderWatcher here, the FileSystemWatcher gets replaced each time I call Start, so then the previous object goes out of scope, but the event handler is still keeping your reference to them, so they won't get picked up by the garbage collector, and the events still fire. So this demo app is not as solid as we thought, and there are a couple of other issues that we'll see as we go through the course. Looking at the Program class again, you may think that this is a pretty naïve implementation with objects being created on demand and with that SqlClient class executing SQL statements directly, but that's just so we can focus on the issues. Later in the course, we'll see an alternative implementation that uses dependency injection

and Entity Framework, but that doesn't get rid of all the disposing issues. But for now, we've set the scene. We have an app that mostly works, but which could fail in different environments while behaving correctly, and we may have a hard time replicating those issues before we can fix them.

Module Summary

We've had a lap around our sample application, and if we ignore all the lurking problems, then this is probably good enough to take looking for a first round of funding, 20 million should probably do it, but the problems are there.
Our application has the potential to max out the database connection pool and to process files multiple times. And we haven't seen it yet, but there is probably an issue with calling the word-count HTTP Service too. The triggers for those defects will be different across different infrastructures and different input files and that's going to make defect fixing very hard. We will knock this demo out into shape over the rest of the course and you'll see that these problems are all from not disposing of resources correctly. In this module, we've had an introduction to IDisposable. We've seen the interface definition and how you use it to manage the lifetime of objects and their resources. We saw that the docs talk about freeing unmanaged resources, but that can be misleading because you might not use them explicitly in your code, but a lot of the system classes that you use all the time are dealing with unmanaged resources themselves. You might think you're building in all-managed application without using native libraries or old fashioned windows com interop so IDisposable doesn't matter to you, but we showed that wasn't the case with a quick demo that queries SQL Server using all brand new managed code, but it didn't call dispose on the necessary objects so it kept connections to SQL open when the object using them had long gone out of scope. And we introduced our demo application that suffers from the same problems with SQL and also has issues with the file system and calling HTTP services. Failing to free resources means you're building problems into your application, which can be hard to replicate and fix. And that's all for the introduction. In the next module, we're going to take a step back and look at what happens when the garbage collector runs, and we'll cover finalizers and the best practice implementation of IDisposable so we'll be ready to go on and fix our application. So stay right here for what happens when the garbage collector runs the next module in IDisposable Best Practices for C# Developers, on Pluralsight.

What Happens When the Garbage Collector Runs?

Introducing the Garbage Collector

Hey, how you doing? I'm Elton, and this is What Happens When the Garbage Collector Runs?, the next module in IDisposable Best Practices for C# Developers, here on Pluralsight. In this module, we're going to get a better understanding of what the garbage collector does, what happens when the garbage collector finds objects which aren't in use, and how using IDisposable correctly means our code works efficiently without relying on the garbage collector to do our cleanup for us. We'll also cover the best practice implementation of IDisposable, which allows you to build a class hierarchy where derived classes use different mixtures of managed and unmanaged resources, but they all get cleaned up correctly by following a simple pattern. A garbage collector is the component of a software runtime which takes care of the memory your app uses, more correctly, it manages the allocation and release of memory. Let's find out how that looks in practice. When you run a .NET application, the runtime creates an app domain, which is an isolated environment for that application. Each app domain can use memory in the stack and in the heap. Whenever you create an instance of a class in your code, a portion of memory is reserved for that object. That's virtual memory, which the operating system makes available to .NET, and it may be space in the physical RAM on the box or in the swap space on disk if the OS is moving things around. That memory allocation for the object holds its state, the current values of any member variables, and its behavior, the methods that you can call on the object. The memory allocation is exclusively for that object, so no other object in that program, or any other .NET program, or any other program from another language can use that piece of memory until it's deallocated. Deallocating the memory happens in two ways, firstly, when the program ends or the address space that the operating system reserved for it becomes free again, so when you are exits, gracefully or because of a crash, or the memory that had reserved gets deallocated. It doesn't matter how many objects are live or whether they've been disposed or finalized, the address space the OS gave to the .NET runtime for this instance of your app is reclaimed. So no matter how inefficient your app is or what state it's in, when it ends, it won't hang on to any memory. The second way memory is deallocated happens while the app is running, and it's done via the garbage collector. The garbage collector looks at every piece of memory the app has allocated and checks to see if the object using that memory is still reachable by the code. In the case of local variables which were used in a method and are no longer accessible, they'll be flagged as dead. But any objects which are reachable, like an instance field being used in the entry point to your app, that's flagged as still alive. After it's checked all the objects in the memory used by the app, the GC deallocates all the memory used by dead objects, so that memory becomes available again, and it compresses the heap, so available memory can be used more efficiently. Now, actually, that's a simplified version of what the GC does. And in the next clip, we'll go down to another level of detail because to write efficient .NET apps, this is something that you need to understand.

How and When the GC Runs

Running a garbage collection is expensive because it requires the GC to walk the whole object graph in your app and check every object's usage to see if there are any live references to it. So the .NET garbage collector uses a more complex mechanism to make sure it runs efficiently. The GC groups objects into different generations. The basic assumption is that old objects, those which have already survived a previous collection, are long-lived, and they don't need to be checked every time the GC runs, whereas objects that haven't been seen by the GC before are new, and they could be short-lived, so they do get checked every time. The .NET GC has 3 collections, Gen 0, Gen 1, and Gen 2. If an object survives Gen 0 collection because it's still being referenced in the app, it gets moved to Gen 1, which is collected less frequently. If it's not being referenced, it gets flagged for removal. Next time the GC runs, there will be new objects in Gen 0 to check, and again, if they're still in use, they'll be moved to Gen 1, and if not, they'll be flagged for removal. But now the Gen 1 objects are also checked, and if they're still in use, they'll survive Gen 1 collection and get moved to Gen 2; otherwise, they get flagged for removal. So during the typical running of your application, you'll have lots of objects that come in and out of Gen 0, some objects in Gen 1, and a few long-lived objects in Gen 2. And the GC only does a full collection, checking all objects in all generations, when the app is running low on available memory. And there's still more that the .NET garbage collector does. It actually manages two memory heaps, the standard heap used for most objects and an additional heap used specifically for large objects that take up a lot of memory. When the GC runs, it can check both heaps for dead objects and flag them for cleanup. When it has deallocated dead objects after a collection, it compresses the ordinary heap, removing gaps to make the remaining memory available in one block, which makes allocating memory for new objects more efficient than trying to allocate across many small pieces of available memory. The large object heap doesn't get compressed automatically, so the GC doesn't spend time moving bigger chunks of memory around. So when the garbage collector runs, it does a lot of work, and the main job of the GC is to get a balance between keeping memory available for the app to use, ideally, contiguous memory so allocation is faster, but also not slowing down the application and soaking up process of time. It does that by running collections efficiently and by running them infrequently. So in principle, that's what the GC does. I've simplified some of the details, and I'll go on to describe where IDisposable comes in during the module, but in brief, we've seen what happens when the GC collects. But when does the GC actually run a collection? Well, there are different trigger points for different types of collection. When you allocate a new object in code, before the allocation happens, the GC checks to see how much memory the app is using. If the GC doesn't like the amount of memory the app has still available, it considers the app to be under memory pressure, and it will trigger a collection. Whether it's a full collection or a partial collection depends on how

much pressure the app is under. The GC has thresholds to decide if it can run a quick Gen 0 collection, or both Gen 0 and Gen 1, or if it needs to run a full, expensive generation 2 collection. When the GC decides that it needs to run a collection, it kicks it off in the background. There are variations of this that you can configure, but generally, the GC runs its collection in a separate thread while the rest of your app carries on running in the main thread. If the collection has been triggered by your app, allocating some memory for a new object, your allocation will happen, and the work will carry on running on the main thread while the GC is checking allocated objects for liveness in the background thread. So the GC tries not to impact your running application, but it still needs to use compute resources for every run. So it aims to run as infrequently and as efficiently as possible, which is where you can help by correctly using IDisposable. Well, that's enough theory for the moment. So next, we'll have a look at what actually happens when the GC runs. I've got a demo coming up where we'll go back to using the SQLConnection class, and I've put that get date code from the module introducing IDisposable into a console app that we can monitor as we're running.

Demo: Understanding Garbage Collection

In this first demo, I'll be using static variables in my program class and I won't dispose anything. I'll run the app in debug mode with Visual Studio's memory profiler which we can use to collect statistics about allocated objects and memory so we can profile the app at different times. And then I'll force the garbage collector to run and see what happens with the object graph after collection. If you want to follow along with the demos for this module, you just need to have a local SQL Server instance running and I'm using a Docker container for that with an image that I've already packaged up and put on Docker Hub. So the easiest way to follow along is to pick up the README file in the downloads and run this Docker command. Now, I've already got Docker running so if I open my terminal, I'll just make sure my database is still there and my container is there, it's listening on port 1433, which is the default SQL Server port. So then I'll switch over to the full Visual Studio, and we'll have a look at this SqlConsumer solution. So here is the code for that database state class, it does the same thing that we've seen in other modules. It opens up a SqlConnection, and when you call this GetDate method, it just executes a sql command and returns the value of the current date from the database. The sql command is a disposable object and that gets used inside a using block, but the SqlConnection object is a local field and that doesn't get disposed. So there is one big implementation difference with this version of the class. I'm not implementing IDisposable so I'm going to have issues when I run it. In the Program class, for this first iteration, I have a static field for an instance of the DatabaseState class. When the application runs, it loads the configuration settings from JSON using the standard .NET configuration libraries, it prints out some instructions, and when

the user hits G, it runs this GetDate method. In the GetDate method, that static database state field gets instantiated if it hasn't been created already, and then we make a call to the GetDate method and print out the results to the console. So the first thing I'll do is I'll start my app in debug mode and then we go back to Visual Studio and I'm going to go to the Debug menu and open the window for the diagnostic tools. There are lots of ways to profile .NET applications, but this view is built right into all versions of Visual Studio, including the free community edition. It's super easy to use and it gives you all the details you need. My app is running and currently is not doing anything, but I'll take a snapshot of memory right now, but we won't analyze it just yet and I'm going to hide the details that Visual Studio is showing me. So first, I'm going to get the app to do something. So I'll hit G, which instantiates my database state instance, calls GetDate using your SQL command and a SQL connection, and it doesn't dispose of them, so my call has worked. And I'll take another snapshot now and I'll hide those details just for a second. We've had one run of the method. We know we'll have an instance of the DatabaseState class, which has its own instance of the SqlConnection class, and they'll both be live because the app can still use them. The Program class is still active and that DatabaseState object is a static field so it still has a live reference. We'll have more than two live objects in the heap though because I have a console that .NET has drawn, there is some framework code to write to the console, and to read the configuration from JSON. So how many live objects do you think we'll have, 10, 100? Let's see. Actually, it's over 3,000, which sounds like a huge amount and it is. We've only allocated a few 100 KB of memory in the heap, but we have lots of objects in there. If I follow the link here, I can see all the classes that have instances and it's sorted by the number of objects I have over 1,000 strings here. I can drill down into the instances and see that a lot of these strings are static variables, and if I scroll down, I'll see more static variables to do with the SQL Client classes. So I can guess that these are optimizations in .NET which mean loading and reading configuration and making SQL calls puts a whole load of strings into memory. Up here, I can filter by type names, so I'll search for my own SQL consumer namespace and I'll see just one object there which is my database state object. It's using a tiny amount of memory by itself, just 32 bytes, but the inclusive size here includes all the child objects and that's more like 13 KB. The Paths to Root here shows me how the object is still alive, and if I open this up, we can see that static field reference inside my program class, and if I look at the reference types, it will show me the objects that this object still owns. So I've got my SQL connection in there, and inside there, I can see an internal connection and my database pool and some strings in there. So I've really only executed one line of code that I care about and we can see there are already lots of objects that the garbage collector is going to need to manage. And next, we'll go back to that program class and we'll change it to use a local variable, instead of a static field, and see how that compares.

Demo: Memory and Object Profiling

In this clip, I'll continue with the same demo application, but I'll change from using a static field to using local variables. I still won't Dispose any objects, and we'll use profiling to check for object leaks. That's why memory consumption keeps increasing because we've got unused objects that the garbage collector won't clean up. So my app is still running here, and if we compare our snapshot with the 3000 allocations to the very first snapshot, which was taken after the app had written to the console, but before we did anything, there were originally 570 objects using about 150 KB of memory. So running our code once has allocated another 2700 objects and around 250k of memory, but that's not necessarily an issue unless it does that every time we run the method. Let's check. So if I go back to my application here, and I'll run the GetDate method another 10 times. Now, I'll take another object snapshot, and we can see there's a little bit of variation there. There's 57 new objects and 4 more kilobytes of memory, but we still have around 3000 objects and 400 KB, so we don't have a big object link in our code. If I go back to the app and force a garbage collection, now that runs this code in the Program class, that does a full generation 2 garbage collection, so it's the most expensive and most thorough collection, and the app has waited until the collection is complete. Now I'll take another snapshot. The Process Memory graph here shows me that the garbage collection has happened. That's with that little yellow marker, but the results are much the same, 3000 are objects, 400 KB of memory. So we can expect that all of those 3000 objects are still live, and so they're not being collected. Now, let's see what happens if I use a local variable instead of the static field. So I'll comment out this code here, and I'll create a new variable using the same configuration that I've already loaded. And in my Console line, I'll call GetDate on that new local variable. So as soon as this method is completed my variable s is going to grow out of scope, so it should be viable for the garbage collector to clean that up for me. Now, remember, my DatabaseState class in this version doesn't implement IDisposable, so my Best Practice #1, Dispose of disposable objects as soon as possible doesn't apply here. So I'll debug that code again, and I'll open the diagnostic tools with Ctrl+Alt+F2. And one important thing here is that these diagnostic tools only work in debug mode in Visual Studio, but if you're doing more detailed analysis on your app, then you should build it and profile it in release mode. In debug mode, there are some differences in how .NET references live objects, so the instance counts that we see won't be quite correct, but they'll be fine for making a comparison. So let's see if my changes made a difference. I'll take a snapshot now, and I'll go back to my app and run GetDate once, and then I'll take another snapshot, and this looks very similar to the original application. The first snapshot has about 150k and 570 objects. By the second snapshot, we've allocated 2700 more objects and another 250k of memory, so the profile is very similar. I'll go back to my app, and I'll run GetDate another 10 times, and we can see back in the profiler that the Memory graph is creeping up. It's only a small increase, but there is an upward trend. And if I check that by taking another snapshot now, I now have over 4000 objects. So this is definitely a problem. My memory usage is only creeping up a small amount, but

this is a tiny piece of code. If I were doing a lot more work and running it in a tight loop, I could soon be using hundreds of megabytes, and once allocated, they wouldn't be quickly released. Looking at the objects in use here, I have 1000-odd strings, which isn't very different from the original version, but there's a SqlConnectionTimeout object high up in the list here with around 70 instances, and there are the same number of instances of the Stopwatch class, which suggests that those two have been created together every time I create a SqlConnection, and they're not being cleaned up. If I go back to my app again and force that level 2 garbage collection. I'm back in the diagnostic tools. I can see the collection has happened, and I take another snapshot. The number of objects doesn't really change. Certainly, it doesn't go back to the baseline of 3000 objects, so this code is doing pretty badly at cleaning up. And next, we'll talk about what's actually happening when we run this code.

Managing the Garbage Collector's Workload

Now we've seen the garbage collector in practice and we've looked at what happens when IDisposable objects are not used correctly, in the first run of the code, our console app allocated a single database state instance as a static field, which in turn, instantiated a single SQL connection instance for talking to SQL Server, nothing gets disposed. When we ran the app and took some snapshots with the memory profiler just running the console without doing any work allocated around 500 objects, mostly in the system namespace and their objects .NET users to manage the app domain and the console. When we ran a call to get data from SQL Server, the object count leapt up to 3,000, and again, most of those were in the system namespace, their strings, and other objects used by the data and configuration libraries. Calling GetDate multiple times in the console didn't appreciably changed the number of objects allocated as we're reusing the same objects to make that database call. And when we forced a garbage collection, the object count stayed the same because all those objects can be referenced by the running app so the whole object graph is still alive and there is nothing to be cleaned up. In the second version of the app, we use a new DatabaseState instance every time the user ran GetDate from the console, but we didn't follow the IDisposable best practices. The DatabaseState instance uses a SQL connection, but doesn't dispose it. So each time we run, there is a stack of objects from the DatabaseState instance and all the allocations that it made in the call tree which aren't being cleaned up as soon as we finish with them. When we ran this version of the app, we got the same 500 dot-object allocations, and when we ran GetDate, that jumped to 3,000. When we ran GetDate repeatedly, the object count kept increasing, adding a few 100 objects every time we make the call. Quickly, we had 4 and a half 1000 objects allocated, and although most of those were local variables which had gone out of scope, the garbage collector didn't clean them up, that's because there was no trigger to run a collection. The app wasn't under memory pressure next time we allocated a

bunch of objects so the GC didn't need to run. When we forced the GC to run, the objects in the graph for those unused DatabaseState instances didn't get cleaned up, but that's because we were running in a debug build where objects get tracked slightly differently. In a release build, we would expect those unreferenced objects to get collected, but only when the garbage collector decided the app was under memory pressure, then we will be back to the baseline, those won't ever get cleared up because .NET is deliberately keeping them in memory for optimization. So in the second version of our console application, after running for a while we have this scenario with 3000 objects that .NET wants to keep in memory and 1000 objects from our GetDate calls which we're not using anymore. This app was running with a minimal memory footprint, so there is no immediate need for the GC to run and clean up those 1000 unused objects. How much memory your app can use depends upon your hardware and operating system. Compiled for 32-bits, you could theoretically access 2 GB of memory, and for 64-bit, the limit is practically all the memory in your server. The GC has thresholds to decide when to run a collection. Those thresholds get adjusted as the app is running and the GC learns more about its behavior, but the defaults range from 16 MB of memory allocated for 32-bit processes up to 4 GB for 64-bit processes. If this was a long-running app and assuming that it could keep running without exhausting the SQL connection pool, then we could easily get to hundreds of thousands of objects allocated before the GC ran, that's hundreds of thousands of objects that the GC needs to check for liveness and it could need to clean up and compact hundreds of megabytes. And again, it doesn't sound like a big deal when you're running on a server with gigabytes of RAM, but .NET apps running in Linux containers are using a really lightweight environment and they might only have a few 100 MB of memory made available to them, but even so, the garbage collector is built to be fast and efficient, so even with lots of work to do, you may not see much performance impact in your app, but leaving the GC to work out that objects aren't being used at the point when your app is under pressure is just unnecessary when you know in your code that you finished using the object yourself and you could free up the resources immediately, saving the GC a lot of work. And we know that we should clean up disposable objects as soon as we can, but our DatabaseState class currently doesn't implement IDisposable so we can't, and in the next demo, we're going to go on and fix that.

Demo: Implementing IDisposable

In this demo, we're going to implement IDisposable on the DatabaseState class so consumers of it, like our console application, can clean up resources as soon as they're not needed rather than waiting for the GC to do it for them. We'll follow the standard pattern for implementing IDisposable in a class which could be extended to allow for subclasses to join in, disposing their own IDisposable members, and also, to dispose of any unmanaged resources that they may

have. So back in Visual Studio, if I add the IDisposable interface to my class and I let Visual Studio generate a stub implementation, I get a couple of options here. If I choose to use the Dispose pattern, I'm going to get this whole big chunk of code with all the best practices that you're going to learn about in this module. So that's the option that you should really choose, but I'm going to start with the basic interface implementation and then build up to the Dispose pattern so you can see why it's useful. So I have a public Dispose method here, and I could do my disposing in there and clean up the SqlConnection, but that's not a good idea. What I should do is have a second Dispose method, protected this time and virtual so that it can be overridden, and this takes a Boolean flag saying whether or not the object is being disposed. Then in the public method, I'm going to call the protected method, passing it the value true so we know that we're in a disposed call. And then on the garbage collector, I'm going to call SuppressFinalize to let the garbage collector know that this instance has cleaned up all of its own resources and that it doesn't need to be finalized. We'll look at finalizers shortly, but for now we just need to know that finalizers are expensive, and this line of code saves us that expensive call, which we don't need because we'll have done our own cleanup. In the protected Dispose method, I'm going to check the disposing flag. The flag is there to help me work out what I need to dispose if I've got managed and unmanaged resources. So in this case, if I'm in a disposing call, I need to check if my SqlConnection object is not null, and if it isn't, then I call Dispose on that object and then set that object to be null. Your Dispose code shouldn't throw any exceptions, and you should allow consumers to call Dispose multiple times without any failures. If I had a lot to do in my Dispose method, I could maintain a private Boolean field, telling me whether this instance has been disposed, and then I'll check that flag in the Dispose call, so I only do the disposal if Dispose hasn't already been called. And when I have Dispose, I'll set the flag. This flag is also a useful thing to have to protect against someone really misusing your class. The .NET guidelines say, an object which has been disposed should throw an ObjectDisposedException if consumers try to access any members. So in the GetDate method here, I can check that disposed flag, and if it's been set, then I can throw an ObjectDisposedException. Although this is good practice, it's far more common to see objects not disposed correctly rather than objects disposed and then used again, so you don't typically need to do this unless someone using your class after it's been disposed is going to cause unwanted behavior. Now, we have a correct implementation of IDisposable, and in the next clip, we'll see what happens with our object profiling when we start disposing of our objects correctly.

Demo: The Full Dispose Pattern

In this demo, we'll continue with our new IDisposable implementation. We'll use a local variable in our console app, but we'll dispose of it as soon as we've used it and see how our memory profile looks and look at the number of objects that the

GC needs to manage. So now that my database state object implements IDisposable back in the Program class, I can replace my straight object initializer with a using block. And now we can start using our best practices for IDisposable and not relying on some future garbage collection run to clean up objects which I know that I'm not using. Implementing and using IDisposable was pretty easy even taking the time to implement it correctly. Now I'll run that revised app in debug mode again with the profiler and we'll see what a difference that change has made. So the same procedure as before, I'll open the Diagnostic Tools and take the initial snapshot before we do any work. Then I'll run GetDate once, take another snapshot, and now we're up to that 3,000. objects which looks like it's being the baseline state for our application once it's warmed up. So now I'll run GetDate another 10 times and now I'll take another snapshot. So we have a couple more K allocated in the heap and the object count has gone up a small amount, but this is much better than the 800 plus new objects that we had when we weren't disposing of our local variables. So if I repeat now just to be sure, I'll run GetDate another 10 times and take another snapshot and here we can see the difference with the previous code where the object wasn't disposing of the SQL connection, but the GC hadn't run because the app still had plenty of memory available. We would have had 4 or 5,000 objects by now. With the new code, I still have the same 3,000. objects, and if I look at the latest snapshot and I search for my namespace, we can see there are no objects there. So I have none of my own objects still live, which means they've been cleared up and any objects they had allocated have also been cleared up without me having to wait for the GC to run. If I clear the filter, then I'll see those SQL connection and stopwatch objects aren't there anymore, and although there are still thousands of objects, these are the things which .NET is deliberately keeping alive to optimize performance, not objects which my own code has explicitly used and failed to clean up. If I go back to the app now, force a garbage collection, and then take another snapshot, the memory profile hasn't really changed because there was no big allocation of unused objects just waiting to be collected. In ordinary usage when the GC does run, it will have a lot less to do 3,000 objects to check, instead of 5,000 so it will run more quickly, which is good for the baseline speed of our application. And because we're disposing of the SQL connection when we finished using it, the connection returns to the pool as soon as we're done so that's good for the potential to scale our application. Next, we'll talk about those changes that we've made and some more best practices for your own applications.

How the GC Uses Finalizers

Our SqlConsumer application is running a lot more smoothly now. We're using one instance of the DatabaseState class each time the user gets the date and the class implements IDisposable, so we can follow the first best practice and dispose of instances as soon as we're finished with them. In the implementation of

Dispose, the database class first checks that it hasn't already been disposed, and then it disposes of the SQL connection instance, which closes the connection, returns it to the pool, cleans up the unmanaged resources, and basically, frees up the stack of objects in that allocation tree. This means our app doesn't have any lurking issues with starting the connection pool and we're not building up a huge amount of work for the garbage collector. We're taking ownership of clean up ourselves, so we can do it as soon as possible, which is the second best practice, if you write a class which has IDisposable objects as instance fields, implement IDisposable in that class, and when your object is disposing, call Dispose on those instance fields. My DatabaseState class has a SQL connection as an instance field. SQL connection implements IDisposable, so I need to implement IDisposable as well. This is the standard Dispose pattern, which we'll look at more closely in the next demo. But when my object is disposing, I call Dispose on any live disposable fields, in this case, the SQL connection object. I'm also being very careful in my Dispose implementation to make sure that this method is safe. You shouldn't assume that Dispose will only be called once on an object, so you need to check that any objects you do Dispose are still in a valid state to be used, which is the third best practice, allow Dispose to be called multiple times and don't throw exceptions when you are disposing. Now, actually, these three are the most important practices. So if you follow them, then your code will be in pretty good shape. But the protected Dispose method and the GC SuppressFinalize call are both there for good reasons, so we should look at what they're doing before we finish with the garbage collector. When the GC runs and finds an object that isn't referenced, it cleans it up, but the GC doesn't know or care about IDisposable, so the .NET runtime doesn't call Dispose for you. IDisposable is there for the user of the class to explicitly run the cleanup when an object is no longer in use. In the original demo app for this module, the DatabaseState class used a SQL connection instance and never called Dispose, but we know you need to call Dispose to close the connection and return it to the pool. So if we didn't ever call Dispose and the GC doesn't call Dispose, what happens to the open connections when the GC runs and cleans up the unreferenced DatabaseState objects and their SQL connection objects. In .NET, we have finalizers for this exact scenario. When a class is using unmanaged resources and it implements IDisposable, it should also define a finalizer, which will clean up only the unmanaged resources in case the consumer of the class doesn't call Dispose. When the GC sees a finalizer defined on a class and it hasn't been told not to finalize that instance, it will run the finalizer. The finalizer is the last chance for unmanaged resources to be safely cleaned up. But having a finalizer on a class means it gets treated differently by the garbage collector, and objects with finalizers can live a lot longer in memory, so you only need to finalizer for classes that are using unmanaged objects. And next, we'll have a demo to see how that works.

Demo: IDisposable and Finalizers

In this demo, we're going to use a class which derives from our DatabaseState class and uses some additional resources of its own, which need to be cleaned up. We'll implement a finalizer to deal with these unmanaged objects, and we'll do our cleanup in the protected Dispose method so we retain the disposing object from the parent class. We'll see how this pattern works with the memory profiler to look at object allocations. So here's a subclass of DatabaseState called UnmanagedDatabaseState. It has two member fields, which we need to do something with, a SqlCommand object, which implements IDisposable and an IntPtr, which is a pointer to memory, and that's an unmanaged resource. I've made the GetDate method in the BaseClass virtual and overridden it in this class. I'm not actually adding anything here. But just to demonstrate, when the method gets called, I allocate the SqlCommand object by getting a command from the connection, which I know has been allocated in the BaseClass by this point, and then I allocate the pointer using the AllocHGlobal method on the Marshal class, which is how I reserve a chunk of unmanaged memory and get a pointer to it. That might be something I need to do to interrupt with some unmanaged code. But here, I'm just initializing and using that block of memory. My BaseClass implements IDisposable, which means consumers can use instances of UnmanagedDatabaseState disposable objects too. But if they call Dispose, then they'll call the public method of the BaseClass, which only disposes of the SqlConnection and not the other resources that my UnmanagedDatabaseState class is using. To do that, we'll go back to the derived class, and we'll override the virtual Dispose method from the BaseClass. That lets me add code to Dispose of my own local object, so in this case, my SqlCommand object. So if we're disposing and my SqlCommand object is not null, then we dispose of it and set it to null so we know that it's been dealt with, and we'll move this call to Dispose on the BaseClass after this one. So the BaseClass method will run and clean up the other resources, but inside here, I also need to deal with my unmanaged resource. I need to clean that up whether I'm disposing or not. So I'll check if the pointer has been allocated. And if not, then I need to run another Marshal call to free that block of memory, and that's the memory referenced by my pointer, and then I can set the Pointer to 0 so that I know it's been dealt with, and this is where the disposing flag comes in. If a consumer forgets to call Dispose on this class, the SQL objects won't be cleaned up until the garbage collector fires, but the unmanaged memory that we've allocated won't be freed up at all. The GC can't release it as it's just a Pointer object, and it doesn't know what it needs to do to clean it up. In the next clip, we'll add the code that we need for the GC to clean up this unmanaged resource using a finalizer.

Demo: Cleaning up Unmanaged Resources

In this demo, we'll finish up our look at finalizers with unmanaged object cleanup. We'll see how to implement a finalizer and how that works with the

dispose method to ensure that all our objects get cleaned up without writing duplicated code. We'll run the profiler for the last time to see that everything is cleaned up the way we expect. This unmanaged DatabaseState class does need a finalizer to deal with the unmanaged memory. In C#, finalizers are defined with a tilde and then the class name. So this is the finalizer for my class which will run once for each object when it gets cleaned up by the GC and not if they've already been disposed as we're calling GC suppress finalize in the base class when we're disposing. So here, I'm going to use the existing disposed method, but pass false so that only the unmanaged objects get cleaned up. That's important because I want to have one set of cleanup code in my dispose method, but it needs to do different things in different circumstances. When the object is being disposed, that means the consumer is finished with it and we're not running as part of a GC collection. All the objects used by this instance are still accessible so we can call dispose on the SQL command and then we can clean up the pointer, but if this method gets called as part of the finalizer, then the consumer hasn't called dispose. We're running in a GC collection and the GC has called finalize. At this point, the managed objects may have already been dealt with so we can't necessarily access them, so we skip the managed resources letting the GC deal with the SQL command, and we only clean up the pointer because the GC isn't going to do that for us. So now we go back to my Program class and use the UnmanagedDatabaseState object rather than the base class and run my app again under the profiler. Like before, I'll take a snapshot now the application is loaded, then I'll call GetDate and take another snapshot, and GetDate 10 more times, and take 1 more snapshot. Every call to the GetDate method uses an instance of the new class in a using statement so they all get disposed. And if I check the snapshot after making all those calls, there are still around 3000 objects so we're about the normal baseline for this application. We can see the memory in the graph isn't spiking. We don't see the 100 MB that the new objects allocate because that's unmanaged memory. The managed memory that my app uses on the heap still doesn't go above about 400 K. If I go back to the app and force a collection, take a snapshot now, and the profile is still much the same. So our disposable implementation dealing with inheritance and unmanaged resources is still working nicely in our app and there are no problems waiting to emerge. And next, we'll recap what we've seen here and talk about what's coming in the next module.

Module Summary

In that demo, we worked through an implementation of IDisposable, which follows best practices in an inheritance hierarchy. Our derived class contained an additional disposable object as an instance field and an unmanaged resource, a pointer to a block of unmanaged memory. When instances of this class are disposing, we need to clean up those resources. We override the protected Dispose method from the BaseClass, the one with the disposing flag, and we

leave the public Dispose method unchanged. In the child class, when the object is being disposed, we dispose of the SQL command object if it's been initialized. And whether we're disposing or not, we're going to clean up the unmanaged resources. In this case, we do that with a Marshal call to free the memory that we've allocated and set the pointer so that we know we've done the clean up. This is also where you would finish up any interrupt work that you've done with unmanaged resources and then call Dispose on the BaseClass, passing the disposing flag so the disposal continues through the whole hierarchy. In case consumers of this class haven't watched this course and don't call Dispose, we need a way to clean up that unmanaged memory, which we do by having a finalizer on the class. If a class has a finalizer defined, the garbage collector creates a reference to the object when it's collecting an instance so it knows there's a finalizer to call. That prolongs the life of the object, but it means we will always be able to clean up unmanaged resources. So in our finalizer, we call Dispose, passing false, so we use the same disposing logic but we only clean up the managed resources. At this point in the object's lifecycle, we can't be sure that any managed resources that it references still exist. They may have already been cleaned up, so we only want to work with the unmanaged resources. And that's a full implementation of the Dispose pattern, which has given us a couple more best practices. Best Practice #4, if you implement IDisposable in a class which can be inherited, do your disposal logic in a protected virtual method, which child classes can override with their own cleanup logic. By following this, you ensure that each class in the hierarchy cleans up only its own resources. Derived classes clean up any fields that they define and call Dispose on the BaseClass for it to clean up its own fields. So every class is responsible for its own cleanup, and you don't risk having objects which are only partly cleaned up because they override the BaseClass Dispose metho2d, but don't call the BaseClass to do its disposal. And the last one for this module, Best Practice #5, only define a finalizer on your class if you're using unmanaged resources and only release unmanaged resources in the finalizer. The finalizer should call the protected Dispose method, passing false, so that method ignores any managed resources and only releases unmanaged fields. This pattern means if consumers do use your class correctly and dispose of instances when they finish with them, the managed and unmanaged resources will be cleaned up straight away. If consumers don't call Dispose, then if the object is out of scope the next time the GC runs a collection, the GC will take care of cleaning up the managed resources, and your finalizer will run, eventually, to clean up the unmanaged resources. So that's what happens when the garbage collector runs and how IDisposable fits in. We started off looking at the garbage collector's role, which is to ensure your app has sufficient memory available to keep running and how that can be a costly operation in terms of processing. So the GC only runs a collection when it's needed. So if your app is using disposable objects, but not disposing them, they will sit around taking up memory until the GC finds the app is under memory pressure and runs a collection to clean them up. We demonstrated how that works with a very simple app that we ran with the Visual

Studio profiler. Our app used the .NET configuration system to get a database connection string from config and then made a call to GetDate on a configured SQL Server database. When we ran our app, we saw it was allocating about 3000 objects to run that method. Without disposing the objects we used, that memory usage went up and up, causing the app to use more and more memory and meaning the GC would have more and more work to do when it did run. By simply following the best practice from module one and disposing the object as soon as we had used it, we kept the memory profile of our application flat, and we saw how to implement IDisposable correctly, calling Dispose on any managed resources that are disposable, allowing Dispose to be called multiple times without failing, doing the actual Dispose in a protected virtual method. So if your class is derived, then the child class can also dispose of its resources and only using your finalizer if your class has unmanaged resources. Finalizers are expensive, so you should only use them if they're needed, and in your public Dispose method, call SuppressFinalize on the garbage collector so it knows this instance doesn't need to be finalized, and it doesn't need special management. In the next module, we'll revisit the demo WordCounter application, looking at what happens if you don't Dispose and applying all of our best practices to fix the problems. That's in What Happens If You Don't Dispose, the next module in IDisposable Best Practices for C# Developers, here on Pluralsight.

What Happens if You Don't Dispose?

Approaches for Finding and Fixing Disposable Issues

Hey, how you doing? I'm Elton, and this is What Happens if You Don't Dispose, the next module in iDisposable Best Practices for C# Developers. In this module, we're going to return to our WordCounter application and look at how we find the issues that we've got from not using iDisposable correctly. We'll also look at a more modern alternative implementation of the app, which uses Entity Framework for data access and Microsoft Dependency Injection Library, and show that this more typical architecture has some different problems. We'll fix those problems in both stacks by drilling into the code and applying the best practices that we've seen already in this course together with a couple of new ones. We've already seen several examples of problems that you get when you don't use iDisposable correctly. If you build a solution without any awareness of iDisposable, you're relying on the garbage collector to do everything for you, and you'll be building in issues. They can be nonfunctional problems like excessive memory usage, performance hits when the GC runs a collection, and holding onto external resources. That can lead to functional problems, crashes when your app runs out of memory, or database connections, or maxes out the TCP sockets for a service call. And they can be straight defects like files being locked which aren't anymore in use. We've also seen that some of those issues only occur in

certain circumstances. The GC doesn't run predictably, so how do you go about finding problems caused by undisposed resources? With any defect, the later it's found, the more expensive it is to fix, but that's especially true with these sorts of issues, which can take a lot of effort to track down even when you know they're happening. If you build a problem into your app by failing to use iDisposable and it only gets flagged in testing, it may be difficult or impossible to replicate in a different environment, so your defect fix may just be a process of trial and error. Alternatively, you may have to do root cause analysis using a memory profiler, taking snapshots of your app before and after running the journey that exposes the defect. That sort of analysis can be very time consuming and frustrating. It's often a cyclical process. You find objects are still alive that you think should have been cleaned up, you find the code, fix it to follow the iDisposable best practices, and you find you still have a problem with a different set of objects in a different part of the code. So finding issues post build at runtime is difficult and expensive, so you're much better off trying not to create problems by using iDisposable correctly at build time. Static code analysis can help tell you where objects aren't being disposed. We'll do that in this module, but we'll see that it's not 100% reliable. In some situations, you get warnings that aren't helpful, and worse, in other situations, you don't get warnings when objects aren't being disposed. When you're writing the code, you can try and identify the most likely areas to cause maximum pain by looking at the touch points where your managed code interacts with the outside world and focusing on any classes that enable those parts of the stack, like database connections and file and network I/O. But the best way really is just to know your domain, get familiar with the classes that you use regularly, and understand how you need to handle them. Code reviews also fit into this last category. So peers may be able to identify iDisposable classes where the lifetime isn't being managed properly, and correcting these issues at design time is much more efficient. In the demos coming up next, we'll use a combination of those techniques to find and fix up issues with our WordCounting application.

Demo: Finding Problems with Object Lifetimes

In this demo, we'll run through our original WordCounter app, looking to zone in on the problems we have from not disposing. We'll do a quick sweep for defects, especially looking at the file.io, and then we'll run the app under the memory profiler with a full-sized input file to see if that flushes out any more issues. I'll be running the WordCounter console app that we saw in the module introducing iDisposable, that uses a SQL Server database and a REST API, and I have both of those running in Docker containers. All the source code for the applications and the containers is in the course downloads, and you can follow along by running these dependencies with Docker Desktop. And I've already started my containers, and we'll just check that they're running here, and there I've got my SQL Server database, which is listening on the standard port 1433, and my REST

API, which is listening on port 8080. So now I'll switch to the full Visual Studio, and we can run our console application. I'll start by running the app in debug mode, and I'll start it listening for input files, and I'll open up some Explorer windows here, and I'll start by copying in the smallest file. We'll see all the lines get processed, the file gets removed from the input folder, and a copy of the original input file is put in the archive folder. And I can open that with Notepad and see that it contains all the original text. Now we're trying to find issues, which may be caused by not using dispose, and those issues may only emerge in certain circumstances, like before the GC runs a collection. Because the GC decides when to run, it's going to be hard to identify those scenarios, which is why it's hard to replicate certain defects, so I'm going to cheat, I'm running in debug mode, and I'll put a break point in the code that processes the file just here. Now that's after the input file has been copied to the archive directory, and the initial data has been written to the database. I'll drop another copy of that same file to the input folder, and my break point gets hit, and now the application is paused. The archive path here uses a random file name, and I can see this one starts with 873, so if I open my archive folder here, that 873 file exists, although it's showing as 0 KB. And if I try and open it with Notepad, you'll see I get an error saying that I can't access the file because it's being used by another process. But my archiving code has finished, so the app thinks the file has been copied, and it's moved on to the next stage, but it hasn't released the right lock on the file, so no other processes can access that file. I'll put another break point down here, and I'll let the app carry on to the point where it started all the HTTP service calls. We're there now, so we've read all the lines from the original file, started all the tasks, but the archive file here is still 0 KB, and I still can't open it with Notepad because of the file lock is still there. So now I'll let the app run on until the whole processing is complete, and now we can see the file does have a size, and I can open it with Notepad, and that's all good, so the file lock has been released. So at some point during the execution of all the HTTP tasks, the garbage collector ran, and it cleared up the file handle for the archive, but we don't know when that happened, and it wasn't something that we controlled in our own code. Now I've had to artificially run with a break point to expose that issue, but it doesn't mean that it wouldn't happen in production. The garbage collector is constantly adjusting thresholds, so we could have a situation where it decides to run less frequently, so our file handle gets locked for longer, and if we had a scheduled job to back up the archive files, it wouldn't be able to access the ones that were locked. And the next thing I want to do in this demo is run the app with the memory profiler. I'm going to switch to Release mode here because when you're running in Debug, object lifetimes are slightly different. So I'll build in Release mode, and then under the Debug window here, I'm going to open the Performance Profiler, and this is an alternative way of doing your memory profiling. Instead of running the app in Debug mode through Visual Studio, and opening the Diagnostics window, you can use the profiler to launch the application independently of Visual Studio, and Visual Studio just tracks the performance. So I'm going to launch an external exe here, which is from

the Release folders, so this would be the published code in my app that I'm going to use in production, and I'm just going to run profiling for .NET Object Allocation, which gives me a lot of detail about how much memory is being used, how many objects are allocated, and what happens when the GC runs. So I'll start my application here, that launches my console application, and back in Visual Studio here, I've got some headline information about how many objects there are, there are 2,500 objects right now. So I'll start the application watching, and then just like before, I'll go back to my books folder, and I'll drop the smallest file first, and that's being processed now, so we'll switch by to Visual Studio. You can see there's a big spike in my object allocation, we're up to 50,000 live objects at the peak here, and then there's a sudden drop back down to about 20,000. And the delta here shows me there's a spike in objects when the file is being processed, a lot of them get cleared up when the application is done processing that file. So what this will show me is the overall trend of the memory. So if I open up my application again, and copy in a slightly larger file this time, we'll see what happens with our overall memory usage. So that gets picked up here, we'll see there's another spike, we're up to 100,000 objects now, and after a little while it goes back down again after the processing has happened, but this time we're down to 43,000 objects. So we haven't gone back to our original baseline, which was only a few 1000 objects. There's still a lot of stuff hanging around here. And the last thing we'll do, and we'll copying a full-sized text, so this is the entire text of the novel, but it's still a very small file, it's under 1 MB in size here. We'll see the processing here will take a little while, so we'll go back to Visual Studio, there's a spike, and we're up to 300,000 objects now. You can see there's lots of allocations and deallocations happening inside here, so we can guess the garbage collector is running a lot and doing a lot of cleanup for us. We're spiking up to nearly half-a-million objects at this point. And if we switch back to the console application, we'll see what's happening here, and now we've hit a whole bunch of errors. And if I grab one of these errors, we'll see it's our favorite error that we've seen lots of times in this course, and we've exhausted the SQL connection pool, and the application is just hanging. It will never fix itself at this point, it's going to keep failing, and eventually it will bomb out. And what we can see here is that it's still trying to do the processing, so it's still allocating objects, and using memory, and other resources, even though the processing has failed. So we've got functional issues in this app, which we'd hope to find in testing like this, but that's pretty late in the cycle. So in the next demo, we'll see what we can do at design time using the analysis tools that we get in Visual Studio.

Demo: Fixing Problems with Static Analysis and Profiling

In this demo, we'll look at some approaches for finding and fixing disposable problems before we get to running our application. We'll add static code analysis to the build of our console application and look at the rules that

get executed to help us use IDisposable correctly. And then we'll walk through the code base looking at each class we use, seeing if it is disposable, and changing the code to use it correctly. We'll run the fixed app under the profiler again to see what kind of difference we get with our fixes. Static code analysis used to be a feature of Visual Studio, but now it's a feature of the compiler, so you can enable it in your project settings. So if I open my WordCounter app here, I can see I've got this property commented out, which is my AnalysisMode. If I uncomment it, that will enable static code analysis with a default set of rules. Now all I need to do is build my console application. And in the Error List here, I'll see a whole bunch of warnings, which come from my static code analysis. If I filter the list looking for dispose, then I see the main rule which we care about, which is CA2000 telling you to call dispose on an object before it goes out of scope. There are several of these failures in my FileArchiver, in my Program class, on my SqlClient, which is telling me that I'm not disposing of my objects correctly. So this stuff now should be pretty easy to fix. I'll start with my FileArchiver. And we know there's a functional problem with file logs being held open longer than they need to. And if I hover over this warning here, I can see why. That's because my file streams are disposable objects, but I'm not calling them with the using block, so I'm not disposing of them when I should be. I've got two streams here, one for read and one for write, and the easiest fix here is to add a using block for both of these. And because they're nested like this, I can use one using block with curly braces for the output stream and have a simpler using statement for the input stream. So you can stack your using statements like this. And what it means when you exit the using block here, both of those objects will get disposed. You can see the squiggly warnings have gone, so I've fixed my CA2000 for my FileArchiver. And that's actually going to fix my functional problem with my file logs being held open for longer than they need to. As soon as the file has been copied to the archive location, the streams get disposed and the file logs get released. So back to my Error List here. We're now down to 4 of these CA2000 problems, so let's have a look at the SqlClient. And I'm getting my error here on my OpenConnection call. So if I scroll down and have a look at this, this creates a new SqlConnection object which is disposable, and it returns it from the method. And the static analysis knows that even though this came from a method call, I own the lifetime of this object so I should be disposing it correctly. So I can fix this again with the using statement, and that will make that error go away. I've got another error here, CA2007, asking me to think about calling ConfigureAwait on the task, but this course is not about async await so I'm not going to focus on that. But I do want to look at the next line where I'm creating a SqlCommand object. Now there's no CA2000 error here, but if I go and look at the definition of this class, I've got a SqlCommand object here which extends from DbCommand. And DbCommand is IDisposable, so my SqlCommand object is also IDisposable. And this is one of those cases where the static code analysis is a bit too cautious. It knows the newly-initialized object should be disposed, but in this case the compiler can't be sure who owns this object. The SqlCommand object gets created by the SqlConnection object, but the compiler

can't be sure whether the SqlConnection object is giving you a whole new object or whether it wants to reuse one. And this is where the domain knowledge comes in because I know that I can safely dispose of this object too. So I'll put this inside a using block. And so I've got stacked using blocks here for my SqlCnnection and my SqlCommand, and they'll both get tidied up as soon as my query has run. And I can repeat this pattern for the next method too. And then for the final method in my SqlClient class, I can take a slightly different approach because my SqlConnection gets used by multiple commands. So in this case, I'm going to nest my using block. So I'll create a using statement for the sqlConnection, and then I'll create another using block from my select statement, and a final using block from my insert statement. So in this case, I've got two separate commands. My select command gets disposed when I find out the ID of my book, and the insert command gets disposed when I've inserted the new data. And then when both those commands have run, the SqlConnection itself gets disposed. I've got other warnings here telling me not to use plain text from my SqlCommands because of a potential SQL injection attack. But again, this course is not focused on security, so I won't go into that here. If I open up my Error List again now, I've just got one more issue to fix, which is in the Program class. And my cancellationTokenSource, which is what I used to stop all of the tasks running if there's a problem accessing the API, that implements IDispose too. So let's put this within a using block. And there's quite a bit of code that gets run that uses this cancellation token source. So the block for this will extend all the way down into the completion of the processing for the file. So that's my last disposable that I need to fix, and all of my dispose warnings have gone away now. And the one other thing I need to think about is my API call. So I've got my ApiClient here that makes the calls to my REST API. There are no disposable issues that get flagged here, but I am creating a new HttpClient object for every call to the API. And if I make use of IntelliSense here, and I have a look at this class to see if it's got a Dispose method, indeed it has. So this is an IDisposable class, which I'm instantiating here then not cleaning up. And again, this doesn't get picked up by code analysis because the rules are quite cautious. The HttpClient is passed in the constructor to my WordCounterApiClient object, and the compiler can't be sure how that object wants to use the HttpClient. And this is another case where experience and domain knowledge really come in because the HttpClient class is meant to be reusable. You're not meant to dispose it after every single call you make because it's more efficient to reuse the same object. So what I'll do here is I'll change this local variable into being a member variable, so then at least I can reuse this HttpClient for every service call that this object makes. So down in the method call here, I'll replace that with my local member variable, my HttpClient class. And now that I have a better understanding of the lifetime of my HttpClient, I can implement IDisposable in this class, and I can dispose of my HttpClient when this class is finished with it. So I'll implement that interface using the dispose pattern that we've already seen. And then Visual Studio gives me the boilerplate code for the correct pattern for disposing. Now inside here, I don't have any unmanaged resources, so I can get rid of the

finalizer. I'll leave the Dispose method as it is, and then the protected Dispose method, that's where I'm going to check that my HttpClient isn't already null. And if it is, that's when I'll dispose, and I'll set the object to null so we don't try to dispose it again. And again, there's no unmanaged resources here, so I'll get rid of this comment to remind me to clean them up. And now I have this class which implements IDisposable. So when I build again, if I check my Error List here, I've introduced a new IDisposable error because I'm not disposing of my ApiClient correctly when I call it inside the Program class. So if I have a look at this here, this is a good example of how using disposable correctly changes how you work with your objects. So this method here gets called for each line in the incoming file. And right now it instantiates a new ApiClient, calls the method to invoke the service, and ultimately returns the count of words in that line. But now I know that my ApiClient will reuse its internal HttpClient to be more efficient. What I want to do instead is I want to take this ApiClient as a parameter to this method. So instead of doing it up here, I'll take it in as a parameter. Then in my loop I need to provide that ApiClient instance. So inside here, I'll create my new ApiClient, passing in the same configuration setup that I've used before, like I said into the method call for my task. And I've still got my CA2000 error here, but I'm not going to fix it in the usual way by applying a using statement like I did here with my cancellation token source because that object gets used in lots of different places. So I need to either make sure that my tasks have finished before I clear up that client, and I should also clean up when there's been an exception. So the best place to put my cleanup code is inside the finally block. And remember that the using statement is just syntactic sugar to give you a try and finally block. And so here, I can call apiClient.Dispose, and I can set my apiClient to null. So when I build again now, I've got no disposable warnings, and everything looks good. But in the back of my mind, I'm thinking about that half a million objects that I saw when I was running my load tests in the last demo. And the only place where I'm creating a large number of objects is in this loop here. So for each line, I create a new task with the Task.Run statement, and I add it to my list of apiTasks here. So really, this is the only part of my code that could be responsible for that big spike in objects when I drop the larger files. So let's have a look at this task definition here. And I can see the task implements IDisposable, but I don't get any errors about clearing up my tasks. And again, this is where understanding the domain you're working with really comes into play. So tasks are disposable, but typically you're working with them inside an await statement, and then the runtime will take care of cleaning them all up for you. But in this case, I'm using Task.Run, which will add that method call to the scheduler. And then I'll call Task.WaitAll to wait for all of my API calls to finish. So in this case, I do know the lifetime of those task objects, and so I can dispose of all those too. So in my finally block here where my apiClient is disposed, at this point either all the tasks will have been completed or I've hit an exception. So I can say for all of my apiTasks with a ForEach block here to dispose that task and then set it to null. And just for good measure, I'll clear down my list so I don't have any references to those task objects. So this is something you would not normally do

unless you were using this pattern where you're creating a whole bunch of tasks and you're waiting for them all to finish and you own the lifetime. Okay, so let's build on this again. I don't have any disposed errors now. I've just built in release mode, so I'll close all these windows here and open up my profiler. And that's going to run the new version of my exe that has all of my fixes. And I'll start that running now. So just like before, when the application starts it has a couple of thousand objects. I'll start listening in my folder, and there's a little bump in the number of objects here. And then I'll do the same as before, I'll copy in the smallest file, and we'll see this starts to get processed. There's a bump in the object count to about 32,000, and then it flatlines without returning to a smaller value. And the reason for that is the garbage collector is running less frequently. I've got fewer object allocations, I'm doing more tidy up myself so the GC doesn't need to run. If I go back to my app and drop in the medium-sized file, now it starts getting processed. We see a similar thing, there's a spike in my object count up to about 50,000, and then we return to about 32,000 again. And then if I copy in the final file, which is the full-size file, we'll see all the processing starts up here. We get a spike to about 200,000 objects, and we'll see how this grows. So we're up to about 300,000 at this point. And at this point, my whole processing is finished. I didn't get any errors, it took 27 seconds to process that entire file, and my object count is now at about 200,000. So we've got a lot less allocation of objects going on, and we fixed all of our functional defects. And the fact that I've still got a lot of objects allocated isn't necessarily an issue here. Because I'm taking the ownership of those objects, the garbage collector has less work to do. If I force the garbage collection now, we'll see the memory profile suddenly plummets back down to about 30,000 objects, which is looking like the baseline for this application. And it's going to return to that point eventually when the garbage collector does its runs. But because we're disposing of our objects, were asking the garbage collector to do less work, and so our whole application is running more efficiently. So we've seen how it looks to find and fix issues with our object lifetimes, and next we'll talk about the changes we made and look at the alternative version of our application that's going to use Entity Framework and dependency injection.

Using Static Analysis to Find IDisposable Issues

We've had a close look at our WordCounting app and found runtime issues, and they're particularly tricky to investigate because you need to test and debug the code or run repeated memory profiler sessions. We also looked at using static code analysis and understanding which classes you use need to be managed correctly. That design-time work is much less effort, and it gets you better quality code before you even start to run your app. Static analysis is a .NET Compiler feature, which you enable in your project settings, and Visual Studio hooks in and shows you the results. Code analysis can find a lot of issues and save a lot of fixing time, but you need to know that static code analysis isn't enabled by

default, that's a performance thing. Running those rules adds a lot of time to the build, but when you do turn it on, the default set of rules includes the main iDisposable rule, CA2000. That identifies cases where you're using iDisposable instances and not disposing them, and that found most of the issues in the WordCounter code. There are some finer details on how this rule works, which you can read about in the docs here. We fixed up the CA2000 issues by wrapping local variables in using blocks, like with this example in the FileArchiver, which is a good use of stacked using blocks where the scope of the objects is still clear. But the checks for rule CA2000 don't find every issue. If you create a new instance, then the rule knows that you own the object, and you should be taking care of disposing it. But if you get an object as the result from a method call, the Compiler can't be certain that you own the object. When you call create command on sqlConnection instance to get a command, the Compiler doesn't know if sqlConnection is creating a new object for you, which you should dispose, or giving you an existing object to reuse, in which case you shouldn't dispose it. So in this scenario, you don't get a CA2000 warning even though you should be disposing of the object. Static analysis catches a lot of problems, but not all of them, so you need to know how the classes you're using expect to be managed, and a lot of that just comes down to experience and understanding the domain that you're working in. We looked at a couple of extra scenarios there too. The HttpClient class implements iDisposable, but it's intended to have a long lifetime. HTTP sockets are expensive to work with, so it optimizes them for reuse. That means a bigger change than just adding a using statement. I changed the ApiClient class to use a local member for the HttpClient and implemented iDisposable with the proper dispose pattern in that class. Then in the Program class, I used the same ApiClient instance for every service call, and I dispose of it in a finally block when all the work is done. And tasks are a special case too. Typically you work with them implicitly with await statements, or you schedule them for background work, and then you shouldn't dispose them because you're not always sure of their lifetime. But in this case, I explicitly await all of my tasks, so I know when they're done, and then I can dispose them all. This is an optimization you usually don't need, but it's good to know that tasks are disposable if you create a lot of them over a short space of time like in this example. Those changes fix the defects in the app and also gave us some important performance gains. In the first run of the code, the smallest files each took around 3 seconds to process, and the object allocations rose from 50,000 to 100,000 objects. The biggest file jumped to 650,000 object allocations before it started to hit errors with a maxed out sqlConnection pool. The fixes, and these were all about managing disposable object lifetimes, reduced the file processing time from 3 seconds to sub second for the smaller files, and it reduced object allocations by about 50% across the board, which had the side effect of reducing the amount of garbage collections. So this is the next best practice, enable static analysis with rule CA2000, but don't rely on it exclusively. If you have a large code base, enabling code analysis will slow down your development pace as every bill takes much longer to run, but in the majority of cases, CA2000

will find problems and help you stick to best practice number 1, disposing of resources as soon as you finish with them. But in some cases, this rule doesn't help, so you need to be aware that there's still manual work to be done, which comes with your domain knowledge. So we've got a more reliable version of the WordCounter app here, but the design is still pretty bad, and I have promised you a look at managing disposable objects within a more modern architecture, and we'll do that in the next demo. We'll look at a better, more solid version of the app where the work is split out between lots of smaller classes, and we use standard .NET libraries for dependency injection and data access.

Demo: Managing Object Lifetime in Modern Apps

In this demo, we'll look at disposable objects in modern .NET apps which use dependency injection. That moves the responsibility for managing object lifetimes to the DI container. We'll have a lap around the new code base and run the new version of the app under the profiler to see how it performs. So in the same WordCounter solution for the source downloads, I've got this WordCounter.ModernApp, which has my more modern architecture. Inside the Program class here, you can see I'm using the standard DependencyInjection library from Microsoft.Extensions. That's a .NET standard library, so you can use it with .NET 5, .NET Core, and .NET Framework applications. In the setup for the application, I create this ServiceCollection, and I'll use that to register all of the dependencies that my application needs to use. For my basic dependencies, I'm adding a singleton instance for the application configuration, and I'm adding logging, which means each of my classes can get their own logger, which has been configured using the centralized configuration. Then I'm adding each of my own classes, so the FolderWatcher and the FileArchiver we've already seen and some new processor classes, which are going to do the actual work of processing the books when they get copied into the folder. And instead of using a custom SQL client that fires off manual SQL statement, I'm using Entity Framework here, and I'm adding my database context as a service to the dependency injection container so objects which need to use the database can request the DbContext from the dependency injection framework. The rest of the code of the Program class doesn't do very much. It fetches the FolderWatcher object from the ServiceProvider and configures it to start watching. So the FolderWatcher here is the entry point for my application now. If I open this up, the FolderWatcher class is using dependency injection, and then the constructor, it receives a BookFeedProcessor and a logger, so we don't have these Console.WriteLines anymore. We're using the consistent logging framework throughout. The code to start watching the folder and to sleep when files are created, that's exactly the same. But now when the file gets created, the FolderWatcher class calls on the BookFeedProcessor to process the file. So I've broken out the logic for my application into smaller classes that each have a specific concern. The

BookFeedProcessor manages reading all the contents from the file, and then it uses this BookLineProcessor, which processes each line in the book and calls the REST API to get the WordCount, and it saves the data for that line in the database. This class uses dependency injection to get the objects that it needs to use, that's the database context and the ApiClient, and the code to process the line of text is very similar from what we originally had in the Program class in the older application, except that I'm using my database context here, so I can work with an object model instead of building up SQL statements. So this is all fairly standard stuff, but where it gets interesting is in the BookFeedProcessor itself. So this is the class that gets used by the FolderWatcher whenever a new file gets dropped. But in the dependency injection for this class, the requirements it has is the database context and a FileArchiver, but it doesn't receive an instance of the BookLineProcessor as a dependency. Instead, it uses this ServiceScopeFactory, which you can use to resolve an instance of the BookLineProcessor on demand. So what we'll see if we go and have a look at the processing logic inside here, when a file comes in, it uses the FileArchiver class to copy it to the archive destination, then it saves the basic details about the book to the database using the DbContext, then it sets up the CancellationTokenSource, reads all the files, and kicks off all of those tasks. When each task runs, it's going to call the GetWordCount method on the BookLineProcessor, but objects of that class use a DbContext, and the DbContext isn't meant to be shared between different threads. So when I have multiple tasks all running simultaneously and the methods running inside those tasks use DbContext, I've got to make sure that each one has its own database context instance, and that's what the ServiceScopeFactory is for. So this is a built-in part of the dependency injection library. Any class can request a ServiceScopeFactory, and then it can create a new scope and fetch a dependency within that scope. And then each scope has its own object graph. So for each of the tasks that run, there w ill be a new BookLineProcessor object that gets its own database context object. Ultimately, the BookLineProcessor uses an ApiClient class. And we can see, this will probably need a bit of work because it uses an Http_Client as a local member, but this class itself doesn't implement IDisposable. And then the FileArchiver class, that's got the same original code without the using statements. So the first thing we'll do is switch to Release mode, we'll build this new console out, and then we'll run it through the profiler. So my profiler here is set up to launch the ModernApp.exe in the release folder, and again, we're just doing the Object Allocation Tracking so we can see what happens with the memory. So I'll start the application running, and I'll kick off the FolderWatcher, and the first thing that we see is that even when the application isn't doing anything, it uses a lot more objects. So we're up to nearly 14,000 objects already, even though the application hasn't done anything yet. And then I'll do the same thing I did before. I'll start by copying the smallest file to my books location, so we can see the processing happens in the same way. The logging is slightly different because I'm not doing Console.WriteLines. I'm using the logging framework. Now, if I go back and look at the object allocation

here, it's significantly more than we had with our older application. So at the peak, there are 120,000 objects being allocated, and we can see, there's lots of allocations and deallocations happening, which suggests the garbage collector is doing quite a lot of work. So I'll carry on and drop the second largest file, and that's finished now. That took about 3 seconds to run. And again, there's been a big jump in memory usage, so we've peaked at about 280,000 objects. And even though some objects have been reclaimed here, we're still running with 230,000 even though the processing of that file has finished. And the last thing I'll do is I'll drop that full-sized file. And because we're using dependency injection and Entity Framework, we shouldn't expect to see any issues this time. This was the file that caused the SQL connection exhaustion when I was running with my older application before I had my disposable fixes. So it's running along nicely here. If I look, we're using an awful lot of objects, so we're up to half a million object allocations here. There's a lot of allocating and deallocating happening, so the garbage collector is doing a lot of work. The application certainly seems to be running a lot more slowly. And I'll pause the video here to let this run along, and we'll see what happens when it gets towards the end of a file. Okay, so that's all worked correctly. It took 170 seconds to complete the processing. And if I look at my diagnostics here, I can see we've peaked at nearly a million objects, and there's been lots of deallocations happening. So it looks like there is something wrong with our application. And actually, sometimes when you run this with the full-size file, you will still see SQL connection errors where the connection pool gets maxed out because of the way we're using our database contexts. So this version of the app looks like it's a lot harder on resources than the original, but we haven't fixed up the disposable objects yet, and we'll do that in the next demo.

Demo: Fixing Lifetime Problems with Dependency Injection

In this demo, we'll fix up the object lifetime management in the modern application using static analysis to find disposable issues and following that up with a look at how we're scoping object lifetimes. Some of the issues we'll see need special treatment with dependency injection, so we'll look at how that works too. Back in Visual Studio, I'm ready to start fixing up the app. So let's do what we did before, and we'll open up our project file, and we'll switch static analysis on with that AnalysisMode, and we'll rebuild this application. So we'll look at the Error List here. We're still searching on dispose, and I see I've got three instances of CA2000, two of them in my FileArchiver. So we know how this works. I'll go back to my FileArchiver here, I'll put my using for the inputStream and a using block around the outputStream, and that's going to fix those issues. I've got my CancellationTokenSource issue here, which we knew about from the previous iteration of the application, so again I'll put all this code within a using block. This is in the BookFeedProcessor class now, as opposed to the Program class, but it's the same issue where my CancellationTokenSource gets

used to stop on my task if there's a problem with the processing. But that's it, I don't have any more dispose warnings, which is strange, because when I did my performance run, we saw those huge peaks and troughs in the object allocation where .NET was allocating hundreds of thousands of objects and then clearing them up periodically, which suggests that the GC was working really hard. And again, the only way to really work out what's going wrong here is to understand the domain that you're working in, and the real problem is this line of code here. So this gets me a service scope, and I can instantiate a whole object graph inside there using dependency injection, but the scope itself is a disposable instance. So every time I'm processing a line at the moment, I'm creating this whole object graph and not disposing of it, so that's stacking up all this work for the garbage collector to do. And again, CA2000 doesn't find it because the object is being created by an external method, so we can't be sure that the scope object is safe to dispose, but I know that it is, so I'll change this to be a using. And that's going to fix a lot of my object allocation peaks and also my connection pool exhaustion, which we didn't see in this run, but you may see if you run this yourself. And we know there's another fix that I can put in here because I'm managing the lifetime of my tasks, so after this try catch block I can put a finally inside here with the same code that we've seen before saying that we can dispose all of these tasks because we own the lifetime and we're sure that they're done with. So that's everything we need to do with the BookFeedProcessor. The main problem was around creating those service scopes. The BookLineProcessor it takes all its dependencies via injection, and now they're all part of that scope which is being disposed. That should all be good. For my ApiClient, I need to think about how I'm using the HttpClient, and again, this just really comes with knowledge. You want your HttpClient to be reused, but you don't want to have to manage the lifetime of that yourself. So I'll replace my HttpClient here with an HttpClientFactory, and that requires an additional NuGet package, so I'll install the Microsoft.Extensions.Http package. And then I can get an instance of that factory using dependency injection. I want to make my method calls down here. Instead of using a member field for the HttpClient, I'm going to use a local variable, and I'll paste this code in here to get my variable from the HttpClientFactory with a helpful comment there saying that I don't need to dispose this object even though it is disposable, because the factory is going to manage the lifetime for me. To make that HttpClientFactory available, I need to go back to my Program class, and when I'm registering my services here I'll add one more, which is my HttpClient. I'm also going to register the HttpClientFactory so my BookLineProcessor can reuse the same instance of the HttpClient and the Factory will manage the lifetime. And there's one more thing that I may want to do here, which is to do the lifetime of my dependencies. The default for the AddDbContext is to use a scoped lifetime, and although I am using service scopes, my DbContext doesn't span across multiple tasks, so I can switch this to being a Transient object. I will add a comment here again saying that my DbContext is going to manage my SQL connections for me. So, again, DbContext

is a disposable object, but I don't need to dispose of it myself because my dependency injection lifetime is going to take care of that for me. So now the last class I need to look at is my FolderWatcher, and this has an instance field which is a FileSystemWatcher, and that class implements IDisposable. Now the way that I work with my FolderWatcher, I only have a single instance, and it's open for the lifetime of the application, so technically I can leave this as it is. But this is a public class, and other people might use it in a different way, so really I should make this disposable too. And again, I'll implement that with the Dispose pattern, and when I scroll down here, all I need to do is I can get rid of the finalizer because I don't have unmanaged code. I just have that FileSystemWatcher that I need to worry about. So if I'm disposing, and the watcher is not null, then I can dispose of it and I can set it to null and get rid of this comment here, and now my Modern application is going to work much more nicely. So if I rebuild this again in release mode, I'll close all these windows and I'll open my Profiler, and we'll get this running again for the final tests. And now I'll start the FolderWatcher, and again, we can see we've gone up to using kind of 14,000 objects already without doing anything, and that's just because we're using a much more complex domain to represent our application. And I'll open my folders here, and we'll see how we get on with the smallest file, which is my contents file. Everything starts processing. That took 5 seconds for that run, and we peaked at 140,000 objects, and then we returned quite quickly to a baseline of about 47,000 objects and that's staying at that level now. So if I try it with a larger file, the excerpt file, it gets picked up and processed pretty quickly. It took 3 seconds this time, and again, we had a peak where the applications spun up to about 220,000 objects. And after the processing it hasn't returned so much. It's still using about 180,000. And now for the final test we'll run the full-size file inside here now. And if we check how this is looking this time, we're up to about half a million objects. And there's obviously a garbage collection here that has taken us down from about 680,000 objects down to about 270 as the file is being processed. We can see the trend there, the object allocation is going up and up, but we hit the point where the garbage collector does its collection, and that's at about 580,000 objects, so we're not hitting those million object allocations that we had before we made the fixes to our application. So I'll pause the video here while I let this carry on processing. And we're back. The process is completed. This time it took about 140 seconds, and if we look at our allocations here we've still got these peaks and troughs, but they're much smaller. So we're looking at 500,000 allocations here, dipping down to 270,000, and now we've reached a baseline of about 380,000. If I run my garbage collector here, and we'll check back, there's been a big collection here and now we're running at about 85,000, which is going to be the minimum the application needs once it's warmed up and run for a few files. So next we'll talk about the changes we made to that modern application and the domain knowledge that you need to have when you're managing disposable objects using modern .NET libraries.

Disposable Objects and Dependency Injection Special Cases

In those demos, we worked with a more modern design of our application where the logic is broken down among lots of small classes, which use dependency injection to get the objects that they need to work with. Mostly, that's all transparent with Microsoft's dependency injection extensions populating objects in the constructor. But with the FeedProcessor, we explicitly create a new ServiceScope so each LineProcessor gets its own database context. You need to do that so the DbContext on different threads don't end up clashing. So this snippet of code is quite advanced, but it's the sort of thing you will find if you're processing lots of work in parallel. We're calling Task.Run to schedule some work on a background thread, and that's going to call an asynchronous method, which uses this scopeFactory to create a ServiceScope. The scope has a ServiceProvider that we can use to get our BookLineProcessor, and then we can call a method on that processor object, and it will have all of its dependencies already set up by the ServiceProvider. And we needed to do that in this case because the BookLineProcessor uses a DbContext class through dependency injection. DbContext is not thread safe, so you can have multiple tasks all trying to use the same DbContext object, which means you need to have one DbContext instance for each task, either with a lifetime that scoped to that class or with a transient lifetime, and DbContext is one of those special cases that you don't dispose. You leave it to the ServiceProvider or the ServiceScope to work out, to manage the lifetime of those objects. My LineProcessor class also uses an ApiClient with dependency injection, and that class has another special case, which is httpClient. HttpClient is also disposable, but it's meant to be reused for performance reasons. And so rather than manage the lifetime of those objects yourself, you should use an httpClientFactory, which you get via dependency injection, and then you can create a client using the factory whenever you need to, and the factory decides whether to reuse an existing client or create a new one. All those services are registered in the Program class, and this is using that same dependency injection library that you'll use with ASP.NET applications. There's a lot of integration with that dependency injection framework, so I can add logging and my httpClient with a nice method overload. And I can do the same with my DbContext, so I don't have to worry about how my DbContext is created because this effectively sets up a factory to create my instances with whatever lifetime I require. I'm specifying transient objects here, but if you remove that ServiceLifetime, the default will be scoped. And all those special cases bring us to the next best practice, which is know your domain. It's not a very satisfactory piece of advice, but there are a few key classes that you'll work with all the time that you don't dispose in the normal way, tasks, httpClients, DbContext, all of those will have their lifetime managed in a different way, and you need to understand what works best for the way you're using those objects in your applications. So we went through and fix up all of the

issues in our modern application, and we did some performance tests with the profiler. Before fixing up all of the object lifetime issues, processing a large file took nearly 3 minutes, and during the processing, object allocation kept spiking between 1 million objects and half a million objects, which was giving the garbage collector a whole lot of work to do. I didn't get any exceptions when I ran the demo, but that doesn't mean they're not there, and those potential issues with exhausting sockets or database connection pools, they're still there in that version of the app. And when we fixed all those issues, we got some nice performance improvements too. We're still allocating an awful lot of objects, over half a million when we processed a large file, but we reduced processing time by 30 seconds, and again, we did that just by implementing those IDisposable best practices. But if you compare these drafts to what we had for our badly designed original application, the performance of the modern app is actually far worse. Now, those drafts don't give you the full picture because I was running under the profiler, so I ran some more tests to get a genuine comparison. And you can see that there is a cost to using these modern approaches with Entity Framework and dependency injection. When it's not running under the profiler, my original application can process those large files in under 15 seconds, but the modern application takes more like 25 seconds. So that's really just an aside as far as this course is concerned, but you should be aware that sacrificing some performance is usually worthwhile if you have an application that's easier to work with and easier to maintain. And with that, we've covered pretty much everything you to know about IDisposable, except for 1 new feature that came in with C# 8, and that's for dealing with disposable resources when you're using asynchronous code.

Using IAsyncDisposable with Asynchronous Streams

So by now, you're really familiar with the IDisposable interface. It just has that single method, void Dispose, and it's part of the System namespace, and it's where you're going to clear up your managed and unmanaged resources. But this is only good if your resources can be freed up with synchronous method calls. If your resources need to be freed asynchronously, then you can't do it with the standard IDisposable interface, and that's why C# 8 introduced the IAsyncDisposable interface. It's also part of the system namespace and has a single method, DisposeAsync, but that returns a ValueTask object, which is something you can wait on in an asynchronous method call. But the goal of IAsyncDisposable is pretty much the same as IDisposable except it's for dealing with your resources asynchronously. And the clearest way to understand this is with a really simple demo, and so we'll have a quick look at a really simple application, which shows you how to deal with disposable asynchronous resources. We'll see how to implement IAsyncDisposable, and we'll see how to use that when you're working with asynchronous streams, which is the main use case for IAsyncDisposable, and then we'll see how you actually dispose of these

asynchronous objects. In the downloads for this module, there's this AsyncStream solution, which is a really simple app that will let us focus on this particular problem. It's a console application, which prints out a whole bunch of random strings, and it uses this RandomStringGenerator class to do that. The RandomStringGenerator exposes an asynchronous stream, so this is an IAsyncEnumerable. And IAsyncEnumerable was also new in C# 8, and it's used in classes that are producing a stream of data. So in this case, the StringGenerator needs to asynchronously wait to read some data from a buffer, and then you can do a yield return for each object that it pulls out of the buffer. Now, I don't really need to use a buffer for this particular functionality, but by using a MemoryStream, I can illustrate the problem here. So the way this class works is when you ask it for a set of random strings, it flushes the memory buffer, and then it fills up that buffer with a bunch of random GUIDs and resets the stream position to 0. Then in the AsyncStream, it just pulls out the instances from that buffer. The buffer is a MemoryStream, and that's initialized to use 100 MB of memory so we have some space to work with when the app runs. Inside the Program class, I initialize a new instance of that RandomStreamGenerator and then the await foreach, that's me consuming that asynchronous stream of data. So in this case, each iteration will just pull a new string from my memory buffer, but this could be a remote service that's publishing a stream of events. And by awaiting each iteration, I'm freeing up the CPU to do other things when there are no events coming in. So this application works, and I can run it now. I'm going to put a breakpoint here after all the random strings have been written out. So I'll hit F5 to start the application. We see a whole bunch of random strings get printed out. And if I go back and look at the diagnostic tools and I'll take a snapshot right here, I can see I'm using my 100 MB there, and that's the memory that I allocated inside my stream. And that memory is still allocated even though my event stream has finished, and I'm not getting any more data out of my StringGenerator. So, really, I should be cleaning these things up as I go, and that's where IAsyncDisposable comes in. If I look at the RandomStreamGenerator and look at the definition from my MemoryStream, that just inherits from the BaseStream class. And if I look at that definition, this class implements IAsyncDisposable, as well as IDisposable. And IAsyncDisposable is where I need to hook into. And because I have an instance of that class inside my class and I'm using it in an asynchronous way, then I need to implement IAsyncDisposable as well, so I'll go ahead and do that inside my RandomStringGenerator. So I'll implement IAsyncDisposable, and if I get Visual Studio to help me out here, I don't get the offer of the nice dispose pattern that I get with IDisposable. I just get a generic implementation of the interface. And there's my public ValueTask DisposeAsync method. I'm going to move that to the end of the class. And just like with the standard disposable interface, I don't want to put my cleanup logic in here directly because that will cause problems with inheritance. So what I want to do instead is I want to have a protective method I'm going to call and my convention that's called DisposeAsyncCore, and then I'll call SuppressFinalize on the garbage collector in the same way that I normally would, and all I need to do

then is implement my DisposeAsyncCore method. Now that's giving me a protected virtual async method. And inside here, this is where I call DisposeAsync on my MemoryStream class. So just like with IDisposable, if I use any disposable classes as instance members, I need to call dispose on those when my object gets cleaned up, and it's the same with DisposeAsync. So by doing this, I'll clean up my memory buffer when my StringGenerator is no longer needed. And of course, my DisposeAsync also needs the async modifier because there's an await inside that method. So now my class implements IAsyncDisposable, and all I need to do inside my Program class is make sure I dispose of that object. And just like we have the using statement for IDisposable objects, we have the await using statement for IAsyncDisposable. It works in the same way. It's going to instantiate an object of my class inside the using statement, and then it runs inside a block, and effectively, it calls DisposeAsync when the object goes out of scope. So just a couple of small changes, but now when I run my application again, it works in the same way. It starts up, prints out all the random strings, and then I see my RandomStringGenerator has been disposed. That's a Console.WriteLine that I put inside the DisposeAsync method. And if I look at my diagnostic tools here and take a snapshot of the memory, I'm only using 70k instead of 100 MB. So at this point, my RandomStringGenerator has gone out of scope, DisposeAsync has been called on it, it's called DisposeAsync on the MemoryStream, and that's freed up all the memory that was in my buffer. So this is a pattern that you won't use too often unless you're dealing with a lot of asynchronous streams. In which case, you need to know how to clean them up efficiently, just the way that you do with ordinary disposable objects. So in that demo, we saw how to consume IAsyncDisposables using the await using construct which, wraps around the await foreach loop. If you're struggling to see where you might use this, the gRPC specification allows you to publish streaming services, and you would consume them in C# using an await foreach loop, and you might wrap that service client inside an IAsyncDisposable and work with it in your await using block. Implementing IAsyncDisposable uses a very similar pattern to IDisposable. You need to implement it if you're using an IAsyncDisposable as a local member, and you'll have a public DisposeAsync method that calls into an internal DisposeAsyncCore method, and again, you'll call GC.SuppressFinalize to make sure the garbage collector knows that this object has been dealt with. Inside your DisposeAsyncCore, you'll call DisposeAsync on your local fields, and again, you need to make sure you're doing that safely in case dispose gets called multiple times. Now, in some cases, you'll write a class that can be used asynchronously or synchronously, and then you might want to implement both IDisposable and IAsyncDisposable. And that's fine, and you can absolutely do that, but you need to make sure that you only dispose of your objects once. In your DisposeAsyncCore method and in your dispose method, you'll need to make sure that the objects that you're working with haven't already been disposed when you come to clean them up. And that's our final best practice. You need to implement IAsyncDisposable if your class uses an AsyncDisposable field as a local member variable, and this is likely to be something you work with more and

more as gRPC becomes more popular. And that's all for this module, so we'll finish up by looking at what we've learned.

Module Summary

In this module, we focused on tracking down problems that happen when you don't dispose and seeing how we can find those issues and fix them. We look at static analysis, which is a feature of the .NET Compiler that you have to explicitly opt into, and that can tell you most of the instances where you're using an iDisposable class, but you're not disposing it. We also saw an alternative approach to memory profiling where you can run your applications in release build and get a more accurate picture of your object allocations and your garbage collector calls. And both of those tools are helpful, but they won't find every single problem for you. And making sure you're correctly managing all of your object lifetimes comes down to experience and domain knowledge, so you understand what needs to be done with the classes that you commonly work with. And we saw that equally applies when you're using a more modern architecture. Dependency injection shifts the responsibility of object lifetime management to the service container, but that doesn't mean you won't have any disposable issues. You need to set the right lifetime scopes for each of your objects when you register them, and again, there are exceptions that you need to understand how to deal with. We looked at tasks, and HttpClient, and DbContexts, and all those classes implement iDisposable, but in most cases you don't call dispose on them, you leave that to happen as part of the object lifecycle management. And lastly, we had a quick look at asynchronous streams, which is where you need to implement and work with the iAsyncDisposable interface. That's a separate interface from iDisposable, and that allows you to write classes that implement one or both of the interfaces, depending on how the resources that you use inside that class need to be cleaned up, whether they can be cleaned up synchronously with an iDisposable call, or whether they need to be cleaned up asynchronously with IAsyncDisposable. And that's it for this module where we added a couple more best practices to our list. We've covered an awful lot of detail in this course, and we flagged up the best practices as we walked through our demo applications, but although it's important to understand all of the best practices around disposable objects, some of those practices are more important than others. So in the next module for the course, we'll focus purely on the best practices, and give you some guidance on which ones you should always be looking to implement, and which ones you should know about for the edge cases. So stick with me for Just the Best Practices, the next module in iDisposable Best Practices for C# Developers, here, on Pluralsight.

Just the Best Practices

Must-haves, Nice-to-Haves, and Edge Cases

Hey, how you doing? My name is Elton, and this is Just the Best Practices, the next module in IDisposable Best Practices for C# Developers, here on Pluralsight. We've covered a ton of material in this course, and we've introduced the best practices as our understanding of IDisposable has developed. And in this module, we'll summarize, focusing on the best practices, and help you with some guidance as to which ones you'll use all the time and which ones you need to remember for occasional scenarios. So firstly, let's have a look at the must-haves, and the obvious one is Best Practice #1, dispose of IDisposable objects as soon as you can, and that's because classes implement IDisposable for a good reason. They expect to be disposed of when they're no longer in use so they can clear up whatever resources they're using, whether that's managed or unmanaged resources, and it's really easy to dispose of those objects. The typical pattern is just with a using block. So when the block ends, Dispose gets called on the object without you having to do anything else. But in some cases, the lifetime of the object is slightly different, and in that case you can use a try block to do your work with the object and call Dispose in the finally block. And really, that's all the using block does anyway. It's just syntactically clearer to work with. And to help you make sure you do that, there's Best Practice #6, enable static analysis with rule CA2000, which is part of the default rule set when you enable analysis in .NET projects. And when you do that, you will see lots of these errors, telling you that you need to dispose of objects before they lose scope, and that's going to save you a lot of problems down the line. But remember not to rely on this exclusively because it's quite a conservative rule, and it won't flag up every instance where you're not disposing of your objects. And that brings us to the last of the must-have section, which is to know your domain. Static analysis and memory profiling will help you find errors, but there's really no substitute for knowing the classes that you use on a regular basis and understanding how to manage their lifetime. And in some cases, that's obvious, classes, like Streams and SQLConnections, and even more complex classes, like ServiceScope, they all implement IDisposable. And any time you new up one of those objects, you're going to want to dispose of them as soon as you can. But in a more modern architecture, you're going to find yourself working with classes that need to be treated differently. And we saw that Tasks, DbContexts, and HttpClients, they all implement IDisposable too, but typically, you don't call Dispose on them, and you don't work with them in a using block, and that's because the lifetime of those objects is going to be managed for you with a dependency injection container, or in the case of Tasks, with the scheduler that runs in the background thread. That doesn't mean you don't need to care how these objects get disposed. It just means you need to understand how the object gets managed, things like the lifetime that you're setting for the scope of the objects with your dependency injection container. And then there are some nice-to-have best practices, things that you won't do every day, but you should bear them in mind for when they do

crop up. And the first of those is Best Practice #2, if you use IDisposable objects as instance fields, then you need to implement IDisposable in your own class. And that means you need to be aware if you're using a class that is IDisposable, then you need to implement IDisposable in your own class to clear that object up, and make sure when you're using that class, you stick with Best Practice #1 and dispose of that object as soon as you're done with it. If you're using Visual Studio, then IntelliSense will create a boilerplate implementation of IDisposable that uses the Ddispose pattern, and that will guide you to using the correct Dispose logic in your class. Part of that is Best Practice #3, you don't know when Dispose will get called on your class, and you need to allow for it to be called multiple times without throwing any exceptions. And what that basically means is in the protected method where you actually do your disposing, you're going to have lots of defensive code to make sure you never throw exceptions in case your object gets Dispose called on it multiple times. And remember, you need to put that logic in your protected method to meet Best Practice #4, which is about implementing IDisposable to support a class hierarchy, and this is the correct implementation of the Dispose pattern. So in your BaseClass, you'll have a public Dispose method that calls Dispose on your protected method and then tells the garbage collector to suppress finalization for this instance of your class. In any DerivedClasses, you override that protected method, and you make sure in your hierarchy that each class only disposes of its own resources and calls Dispose on the BaseClass to make sure everything gets cleaned up. And lastly, there are the edge cases. That doesn't mean these are practices that you should ignore. It just means that you won't see them so often. And the first of those is Best Practice #5, when you're using unmanaged resources, then you need to clean them up inside a finalizer. Finalizers always get called by the garbage collector, so it's your last chance to clean up any managed resources that otherwise .NET will not be able to clean up for you. And the pattern for that is really simple. The finalizer method starts with the tilde and then has the class name, and you'll want to make sure that you're reusing the same logic everywhere. So you'll call your protected Dispose method, but let it know this is a finalizer scenario and not a Dispose scenario. And then in that Dispose method, you'll only clean up the managed resources if you're in a disposing flow, and you will always clean up the unmanaged resources whether you're disposing or finalizing. But remember that you need all that defensive code for both of those workflows to make sure Dispose can be called repeatedly without throwing any errors. And the final best practice is for when you're working with asynchronous streams. So if your class uses an AsyncDisposable field, then you should implement IAsyncDisposable. So you won't see this too often, but it's pretty important to remember that it's there so you can do it when you need to. Implementing IAsyncDisposable uses its own pattern, so you'll have a public DisposeAsync method, which returns a Value class, and that's what powers the await using statement in code that wants to use this class. But just like with IDisposable, you don't do the work inside the public method. You do it inside a protected method to support cleaning up resources across a class hierarchy. And that brings us to the end of the course. I hope you

found that useful, and I hope your C# projects are going to be a lot more reliable, thanks to the practices that you've learned here. So if you have time to leave a rating for this course on Pluralsight, I'd really appreciate that. And if you want to see what I'm up to next, then you can follow me on Twitter. And I've got 25 other courses on Pluralsight covering everything from C# and .NET, continuous integration and monitoring, Azure, Docker, and Kubernetes, so you're sure to find something else from me that will interest you in the library. Thanks again for watching. My name is Elton, and this was IDisposable Best Practices for C# Developers, here on Pluralsight.