Course Overview

Course Overview

Hi, everyone. My name is Neil Morrissey, and welcome to my course, Debugging C#. I'm a solutions architect and developer. It's estimated that up to 50% of a developer's time is spent finding and fixing bugs. But even though debugging is a huge part of development, not everyone has the knowledge and skills to effectively debug problems. Whether you're trying to diagnose a complex issue during development or you're under pressure to fix a problem in a production application, being a better debugger makes you a better developer. This course will teach you a set of skills for debugging using real-world examples in an enterprise application. Some of the major topics that we'll cover include debugging features in Visual Studio and Visual Studio Code, an effective approach to debugging based on the scientific method, common traps to watch out for when debugging, and how to include code in your applications to help with diagnosing bugs. By the end of this course, you'll have the skills to find and fix bugs in your own code and in applications that you support. Before beginning the course, you should be familiar with C# fundamentals. From here, you'll continue by diving further into C# with courses on object-oriented programming and error handling. I hope you'll join me on this journey to learn debugging with the Debugging C# course, here at Pluralsight.

Sandbox

Debugging Fundamentals

Understanding Debugging

Hi, and welcome to the course. This is a course on Debugging C#, which involves a set of techniques and tools to help you find and fix bugs in code that you've written or in code that you didn't write, but you need to support. Debugging is a hugely important skill to have as a developer, and I think you'll appreciate that as we go through the course. But first, let's talk about versions of technologies used in this course. This course was created using the displayed version of C#, Visual Studio, and Visual Studio Code. But most of the features you'll see in Visual Studio are applicable to earlier versions also. Where newer features are used that are particular to a certain version of these technologies, I'll point them out as we go. Okay, so let's start with some basics. What is debugging and what is a bug? The term bug actually goes back to the 1800s, before computers, obviously. It was a term used in engineering to describe mechanical malfunctions. Thomas Edison is quoted as having used the term bug. But then in 1946 when Grace Hopper, one of the inventors of COBOL, was working on one of the first large-scale computers called the Mark II, an error in the computer was traced to a moth trapped in a relay. Ever since then, we call errors or glitches in a program a bug. A bug is a defect or an error in code. And I'll refer to bugs, defects, and errors interchangeably throughout the course. In the next clip, we'll talk more about specific types of defects that you may encounter during debugging. Debugging is the process of identifying the root cause of an error and correcting it. Debugging is different from testing, which is the process of detecting errors during development. So, testing is used to find errors, debugging is used to find the cause of the error in the code. You'll do a lot of debugging during development of software. On some projects, debugging might account for 50% of your total development time. But you'll also need to debug a program after it's been released into production and starts being used by actual users and they discover bugs in the application. They tend to do that. There are methods of doing some debugging right in the production environment, but often you'll be trying to recreate the problem in your dev environment. When there's a production issue, that's when the pressure is on because the bug might be preventing the user from doing their job or there might be a calculation producing the wrong result and you need to figure out the issue fast and publish a fix as soon as possible. That's when debugging skills are really important to have. But the reality is, not everyone knows how to debug. Studies have shown that there's a wide variation in the time that it takes different programmers to find the same defects in code. Developers who are experienced at debugging not only find bugs faster, but they also find more bugs. When you have to find and fix a bug in a production application, you're often pressed for time, and it can be stressful. You don't have time to rewrite the code and to make it more elegant, which is often the temptation. So, that's something else that debugging is not. Debugging is not refactoring. Refactoring is restructuring source code to improve its operation without altering its functionality. While you might be able to do that during debugging, if you have time, refactoring is still a different process from debugging. And as soon as you start altering the code significantly, it's possible you could introduce more bugs. Regression testing using unit tests can help identify those bugs before they get published, but unit testing is the subject of another course in this path. Diagnostics in debugging are also different, although there is some overlap. Diagnostics has to do with instrumenting your application to gather data about its internal operation after it's been released. So adding code for logging is a good example of diagnostics or tracing a web call across multiple servers, which would involve writing logs from each location, but in a way that the different calls can be correlated to show the bigger picture. You add code to your application for diagnostics, and you can also structure your application in various ways so you can turn diagnostics on when there's an issue to do some logging and turn it off when there's no problems being reported. And that ensures that the instrumented code isn't affecting the performance of the application, or you might only log certain types of errors. Either way, data you get from instrumentation can help you when you need to recreate the error in your development environment in order to debug it. That's where the overlap is. Instrumentation data can be used as input to debugging. I have an entire course on .NET diagnostics for applications in the Pluralsight library, but we're going to talk more about coding to help with debugging later in this course. So now you know at a high level what debugging is and what it isn't. Next, we'll talk about bugs themselves. I'll give some examples of the types of defects you'll typically be looking for by organizing them into categories. Then we start looking at the features in Visual Studio and Visual Studio Code that help with debugging. We'll use a console project to explore some of the features. Then you'll get dropped into an enterprise application where you'll see how to use the tools for debugging like breakpoints, inspecting live data, and stepping through code. The enterprise-type application that we'll be using is an ASP.NET web app, but you don't need to know ASP.NET in order to follow along. Fixing some of the defects will require knowledge of ASP.NET, but you're going to see how learning debugging skills can help you in supporting applications that you didn't write and may not be very familiar with. So next, let's talk about different types of defects or bugs.

Defect Categories

Let's talk about some of the types of defects or bugs that you'll typically be looking for when debugging. There really isn't a standard set of categories that everyone uses. This list is just basically the result of trying to find some common themes that bugs can be categorized into. Of course, it really doesn't matter how you categorize a bug. That's not the point. You're trying to find and fix bugs, not categorize them. This list is just helpful in discussing types of bugs. Let's start with functional errors. This is when software doesn't act the way it was intended to, so it's not behaving according to the functional requirements that it was designed to implement. This is a broad category of errors, and these bugs are often found during the testing phase of your project. An example could be a Save button that doesn't save. That's pretty obviously a bug, and the software isn't implementing the requirements, but it might not be that obvious. Maybe a form field isn't getting validated when a certain value is entered. So the bug doesn't show itself all the time, just when certain data is entered. You might not discover the problem until the app is in production and it's getting used in ways you didn't test for. Or maybe a user clicks the Save button multiple times in a row, and that results in multiple records getting saved to the database. That's a bug that only shows itself when the software is used in a way that you maybe didn't intend. You'll see later in the module how exception messages thrown by code can often give us a lot of information about these types of errors and even tell us exactly in the code where they were thrown, but we're not always that fortunate. The next category is security defects. These are bugs that open up your software to potentially serious attacks. These types of errors are often discovered during security and penetration testing where you try to essentially hack the software. Examples are things like data breaches in your database caused by SQL injections or cross-site scripting. Other bugs in this category could be buffer overflows and inferior authentication that compromises user information. To prevent these types of bugs, you want to do things like writing your code as if you don't trust any user input and make sure that internal exception messages aren't surfaced to the user so they cant determine the inner workings of your applications. Syntax errors are usually pretty easy to detect, especially when you're using an integrated development environment like Visual Studio or Visual Studio Code. These errors occur right in the source code and usually prevent it from compiling. It could be a misspelled command or a missing bracket or, of course, a missing semicolon at the end of a line. But there can also be syntax errors that are actually logic errors like using an equals sign when you should be using two equals signs. One means assignment, and one means equality. The compiler will usually pick up on it and alert you, but not always. It depends on the context where it's being used. Logic errors are coding errors that cause your software to produce the wrong output or could even cause it to crash. This could be a condition not being met that should cause a loop to complete, so your code ends up in an endless loop. It could be an object or variable not getting initialized or not resetting the value of a variable before restarting a counter, or it could be a problem with the control flow of the code, maybe functions getting called in the wrong order because of a problem with the algorithm that you've written. The list goes on. And from the way the software behaves, these could be looked at as functional errors too because the code isn't behaving according to the requirements. But when you start debugging, you find that the cause is because of a logic error. And logic errors can be broken down further into other types of common errors like calculation errors, which could be an incorrect formula that produces the wrong result. That result might get returned to the user, or it might get passed to another program or service. In an area like banking, calculation errors can be really costly, but calculation errors can happen in other ways too. In 1999, NASA lost its Mars Climate Orbiter because one of the teams working on the code used imperial units instead of metric units, which caused the thrusters to work incorrectly and the orbiter to crash on Mars. So again, calculation errors can be costly. Another type of logic error could be an out-of-bound bug where the system user interacts with the user interface in a way that wasn't expected like entering a parameter outside the expected range or entering a data type that wasn't expected. You might expect the user to enter integers in a field that expects whole hours, for example, but the user enters the decimal for partial hours because they expect the program to allow that. You always need to validate user input, especially when accepting text. Compatibility defects happen when an application doesn't perform consistently on different types of operating systems, hardware, on different browsers, so the application could end up showing the wrong font size, color or alignment. On different mobile platforms, this can be a big problem too with fonts looking different, content alignment appearing in unexpected ways or the scrollbar looking different. Compatibility defects can happen when integrating with other software too. And sometimes these types of defects are caused by varying bandwidth and operating speeds. You might need to write code to work around these types of issues like having to restart sending a file if there's a network dropout. System-level integration bugs occur when there's a mistake in the way that two different subsystems interact. These types of bugs often occur because the subsystems may have been written by different developers, which could result in issues with parsing messages or other types of issues. It can be difficult and time-consuming to replicate these types of bugs. Logging errors in your production systems can help with diagnosing difficult problems like this, especially when you log the data that caused the issue. We'll look at logging later in this course. There are many other types or categories of bugs like performance defects, which are related to the speed, stability, response time or resource consumption of your software. It could be the software is running slowly or response times are longer than acceptable for a user. It's possible to find many of these types of issues during performance testing, and it's usually by comparing performance to non-functional requirements where you've documented the expected performance of the system. Of course, not all projects document non-functional requirements in a lot of detail, so sometimes you'll end up responding to user complaints about performance. Usability defects are another type of bug that's often subjective. These are defects that hinder the user from using the software to its full capability. The user interface might be overly complicated or inconvenient to use. Sometimes this isn't subjective though like when it comes to accessibility. There are standards to evaluate your software against like the Web Content Accessibility Guidelines and even tools that can test your application for accessibility. But sometimes, usability is as simple as providing visual feedback, like when a password is rejected, to indicate to the user the rules that are being checked or maybe an icon whose meaning isn't clear to every user. We won't really be looking at performance defects or usability defects in this course. As I said at the beginning of this clip, there isn't a generally accepted list or standard when it comes to these categories, and there's definitely some overlap. So, categorizing the error isn't really the important part. These categories are just a way to start talking about the types of issues that you'll be seeing. Next, lets get into some debugging features of the integrated development environments we typically use with C#.

Visual Studio Debugger Basics

Let's take a look at some of the debugging features of Visual Studio. I have Visual Studio 2022 Community Edition installed. This is the version that's free for download. There's a console application available in the course downloads, so let's open a project or solution. And if you're following along, navigate to wherever you downloaded the code. I'll open the Solution file, and that will open the project. You can see there's only four classes in Solution Explorer, so it's a pretty small project. We won't look at the code yet. Let's run the project. We can start debugging right from the Start button here, or the Debug menu gives us the option to start with debugging or run the program without attaching the debugger. And there are shortcuts for both, F5 to start with debugging, so we'll be using that a lot. Lets run this with the debugger. So there's a menu with options to view expenses, enter a new expense, edit an existing expense, delete an expense, and total the expenses. Let's view expenses. When the application runs, three expenses are created to simulate having some data to work with, but there's no database underlying this application. These are just created in code, which you'll see shortly. So each of these expenses has a name, a location, a date, and an amount. Let's choose the option to total the expenses. It says the expense total is $22.50. If you do some quick math, you'll notice that this is wrong, which would normally be a bad thing. But since this is a course on debugging, this is fantastic. Let's close the console window, and that will stop the debugger. Now as I mentioned, you can press F5 to start debugging, and that will run the application. When you're not familiar with an application and you want to find the entry point where the code starts, you can start debugging with F10 or F11, and that will start the debugger and bring you to the first line of code in the application. So Program.cs contains the startup code for a console application. And if you're familiar with console applications in previous versions of .NET, you'll notice that there's no static void main method where the entry point is. Starting in C# 9 and .NET 5, that class is implied using what are called top-level statements. This means that the code you write in Program.cs is intrinsically wrapped in a static void main method by the framework, although you can override that and write the method yourself so it appears the same as previous versions of .NET. So the first line of code here creates a class called ExpenseManager. If we open the Debug menu, there are options down here to Step Into, Step Over, and Step Out of code. You should learn these key commands because you'll be using this a lot, and you don't want to have to do that from the menu. Step Over means that when you have a method call in your code like the constructor to create the ExpenseManager class, F10 allows you to run that code without actually debugging into it. It steps over the method call. If we had used F11 or Step Into, it would've brought the cursor into the constructor of that class, and we could debug inside there. From here, we can step through each line of code by pressing F10 or F11 since there aren't any method calls here, at least not to methods that we've written. Let's stop the app and look at another way you can start debugging. If you right-click on a line of code, you can choose Run To Cursor, and notice the shortcut is Ctrl+F10. So its kind of a variation on debugging into the first line of code. That starts the app and breaks right at the line of code you chose. I'll click the Continue button so the application can keep running. And let's bring up the app again, and let's list the expenses. Next, I want to run the code to total the expense amounts. But before I do, let's go into the code and set a breakpoint by clicking in the far-left margin beside the line of code where we want the debugger to break. The code is currently waiting for input at line 22 where it says Console.ReadLine. And this whole block of code is in a do while loop, so it keeps printing the menu, waiting for input and then calling a method in the ExpenseManager class, depending on what the user enters. And the do while loop continues until the user chooses the option that triggers the while condition. There's a variable holding this exitOption value just to make the code easier to read. Let's go back to the running application and select 5 to total the expenses. That breaks execution at our breakpoint, so now we're debugging. If I move my cursor over a variable where the code is already executed, a pop-up shows us the name of the variable and its current value. So we entered 5. And because input from the console is always a string, the "5" shows in quotation marks. This pop-up is called a data tip, by the way. Hitting F10 will step us through each line of code. After the string variable has been parsed as an int, you can see the value in this int variable. The switch statement evaluates the integer and jumps us to case 5. If I hit F10 again, it moves to the next line. But what if I want to debug into that method call? We've already passed that code. Easy. In Visual Studio, you can click and hold on the yellow arrow in the left margin where the cursor is and just drag the cursor up to a previous line of code. Now we can step down through the code again. You need to be careful doing this though because if you've modified the value of a variable that was set above, it doesn't get reset, so you might end up with some incorrect calculations because code is getting run twice when it wouldn't have under normal circumstances. I'll hit F10 and advance again. But now I'll hit F11, and that should bring us right into the TotalExpenses method in the ExpenseManager class. And it does. You can see that a tab opened for the ExpenseManager.cs class, and the cursor is stopped at the beginning of the method. Let's advance past this first line using F10. So there's a variable called expenses that has a Count method. So this must be some sort of array or collection that's holding the expenses we saw printed on the screen. It says the number of expenses from the Count method is 3, and that's how many expenses we saw on the screen. If I hover over the expenses variable, we can actually inspect the collection. So there are three objects in here, and they're each of type Samples.Debugging.ConsoleApp.Expenses.Expense. So the name of the Expense class with the full namespace in front of it, and we can expand each of these and see the values of each individual expense object in the collection. So there's the Amount, the Date, the Location, and the Name and the ID also. And you can see each object represents a different instance of the class with its own data. Now there's a variable called total, so this must be where the running total of expense amounts will be stored. The code has a for loop that's going to get the Amount property from each expense in the collection by using the numeric indexer to access the object in the array. So the first time through, the total is getting a value, and then the value gets added to the second time, and then that's it. But there was three expenses in the collection. Let's look at this again. I'll drag the cursor up, and let's run through this one more time. Make sure you drag it up past the line that sets the total to 0, or you could end up adding to the amount that's already been calculated. Rather than have to hover over the variable each time, I'm going to right-click on this total variable and select Add Watch. That opens up the Watch window at the bottom, and it added the total variable. So we can keep an eye on the value of this variable as it changes while the code executes. Now lets do the same thing for this i variable. It actually already has a value because the code ran before we dragged the cursor up, but this will get reset when it runs again now. F10 moves to the code that evaluates whether the loop should run. Lets set a watch on the numberOfExpenses variable too. I is, in fact, less than the number of expenses, which is 3, so the loop runs. Okay, first time through the total is 2.5, then the counter, i, gets incremented. I is evaluated against the number of expenses, so 2 is less than 3. And then the next expense object is accessed to add its amount to the total. We can see total incremented in the Watch window. Then the counter is incremented again, so now it's 3, which means i is now the same as the numberOfExpenses variable value, which is 3. And the condition that's being evaluated is whether i is less than 3. So the for loop doesn't run again. Okay, it looks like we have a logic error here. When we access the array, we're starting at array index 1, but arrays in .NET are zero-based. So we actually skipped over the first expense in the list. That's why it totaled up the second and third expenses and not the first. So let's implement the fix in the next clip.

Hot Reload in Visual Studio

So we need to make the loop counter start at 0, so that all the objects in the array will get totaled. Okay, now, in earlier versions of Visual Studio, you had to stop the debugger and restart the application to see these changes take effect. But, there is a feature now called Hot Reload that will let us see these code changes in the running application. Hot Reload was actually introduced in Visual Studio 2019 as a preview feature, but it's fully integrated in VS 2022 and .NET 6. I'll click Hot Reload in the menu here, and you can see the app is still running. I'll hit F10, which brings the cursor back to the calling code in Program.cs, and let's just run the app from here by clicking the Continue button at the top. And let's actually remove this breakpoint so the code doesn't break again. You do that just by clicking on the breakpoint in the left margin to remove it. Now let's go to the running app and select to total the expenses. Now the total is $30. I'll view the expenses again and let's double check that. So the calculation is now correct. We've successfully debugged the error, implemented the fix, and tested it. Before we leave Visual Studio, I want to show you another feature that's really useful. When you're debugging through methods getting called from different classes, it can get confusing, it's hard to keep track of where you are in the code and where the method was called from. Let's look at this ExpenseManager class. You saw in Program.cs that the constructor for this class was called right at the beginning when the app first started. In this constructor, it creates a new instance of a class called DataInitializer and calls the CreateSampleExpenses method on that class. Let's right click on this method and Go To Definition, and the shortcut is F12. That opens the DataInitializer class and brings us right into the method. So this code creates a new list of type Expense, and populates it with three new expenses, and returns this list to the calling method in the ExpenseManager class. Let's set a breakpoint at the start of this method. We're still debugging the running app, so let's stop that, and let's run the app again so the expense data gets recreated. As the app is starting, the breakpoint gets hit and the code is stopped. So, we're debugging deep inside this super complex application, and we might not know exactly how we got here, or we can't remember all the method calls that led up to this point. If we go to the Debug menu, under Windows, let's choose the Call Stack window. That brings up this window that has the hierarchy of method calls, with the current one where our code is paused right up at the top. Below that is the ExpenseManager constructor method where this one was called from. You can actually click right on that method in the Call Stack window and it will bring you there. Notice though that it didn't actually move the cursor, our code is still paused inside the CreateSampleExpenses method above, that's why the calling code here is green. If you hit F10 to advance the debugging cursor, it will bring you to the next line of code where the debugger is stopped. But we can trace back farther too, right back to where the call started in Program.cs. So this is really helpful when you're trying to form a mental map of how your code is structured, and it helps you with looking for the source of a problem earlier in the Call Stack. Let's click on the top method again, where the code is stopped, and that brings us back to the currently executing line of code. We can hit F10 to go to the next line. If you're anywhere inside a method and you just want to let the method execute and then resume debugging in the method that called this one, you can do that by stepping out of the method. The shortcut for that is Shift+F11. So, just remember, F11 is to step into the method, and

Shift+F11 is to step out of a method. So those are some debugging basics in Visual Studio. We'll go into a lot more features throughout the course, but these are really the core features that you're going to be using all the time. Next, let's see some of the features of Visual Studio Code.

Visual Studio Code Debugger Basics

Now let's look at features in Visual Studio Code for debugging. We're going to use the same code we used in Visual Studio. And again, this is available in the course downloads. I'll open up Visual Studio Code, which I already have installed on my VM here. And this is also a free download. The first thing I'll do is hold down Ctrl on my keyboard and hit the + key. That'll make all the fonts on the screen bigger. The main menu is along the left here. The Explorer view is open by default. Now Visual Studio Code is a code editor that works with many different languages, but by default, you don't get all the features of those languages like you do in Visual Studio. You need to install that support. So let's go to the Extensions tab, and there are no extensions installed yet. If I search for C#, there are some extensions listed here like the base language support for C#. But the one I want to install is the C# Dev Kit because that will install other dependent extensions like the C# language support. So let's click Install, and it'll take about 20 seconds, so I'll speed this up. Now if I clear the search and pull down this view, you can see there were four extensions installed, including the C# language support. Now lets open the C# console project from the previous clip. On the Explorer tab, you can open the folder from here or from the file menu. Let's navigate to that folder. And again, this is wherever you downloaded the course assets to. Go into the project folder and then navigate up one level, so we're in the folder with the solution file. Unfortunately, you can only see folders in this view, not the actual files. I won't save this configuration as a file for now. The screen refreshes, and we've got all the files in the folder listed here. So this is a bit different from Visual Studio. But because we installed the C# Dev Kit, there's actually a Solution Explorer view, just like in Visual Studio. It can just take a few seconds to appear. So you can work with your files from either view. You can see this has a Dependencies folder just like in Visual Studio. Now let's debug the program. That's done from the Run menu. It's possible to create different run configurations using a JSON file, but let's just go ahead and run this code. I need to select the C# debugger from the drop-down list the first time and select the project we want to debug, even though there's only the one. That builds the code and runs the console application in the debug console here. There's a place at the bottom where you can enter input to the program. So I'll run the first command to view expenses. That runs the code in the program. So, while this works, it really isn't what we expected. We want a console window to open. I'll stop the running program by clicking the stop button. You can specify how you want the console to be launched for each individual project by creating a JSON launch file, or it's possible to configure it as an environment setting. I'll just open this file up because we'll be debugging here shortly. Now to modify configuration for Visual Studio Code, go to File, Preferences, Settings. If I search for console and click Debugger, there's a setting here to indicate which console the target program should be launched into. By default, it says internalConsole, but we can change that to externalTerminal. Now if I go to the Run menu and click the Run button, a console window opens to run the program. Let's make the font size a little bigger here also. We can do that from Properties. Now I'll enter 1 to view the expenses. Okay, so with the program running, we can set breakpoints just like in Visual Studio. I'll go to the Program.cs file and click in the left side of the code editor window beside the line number. That sets a breakpoint. Now if I go back to the running application and enter another menu item, the code breaks at the breakpoint. And just like in Visual Studio, you can step through the code from the menu or using these shortcuts. So lets use F10 to step through. Now something that's different from Visual Studio is that you can't click and drag the cursor back up to rerun this code. What you can do though is right-click on the line you want to run from and select Jump to Cursor. Now you need to be careful you don't rerun any code that changes values when you do this or you could have some unexpected behavior. And click the Continue button on the menu, or I could hit F5. And I accidentally set a breakpoint when I tried to click on the cursor to drag it up. So I'll hit F5 again to continue, and let's clear this by clicking on the breakpoint again. Now let's close the program and look at another way to start it. Let's start the program by debugging into the first line, just like you saw in Visual Studio. For that, we need to use the Solution Explorer feature, and remember that you need to install the C# Dev Kit to get Solution Explorer. Right-click on the project, and choose Debug, Step into New Instance. That'll stop execution at the first line of code. Now let's step through using F10. And at the point where the code waits for input, I'll bring up the app, and let's total the expenses. Now we can F10 down to the switch statement and F11 into the method that calculates the total. Just like in Visual Studio, you can hover over variables to see their values, and that includes collections like this expenses collection of expense objects. We can inspect the contents of each expense item in the collection. You can add a watch for these variables also. Just right-click on the variable and select Add to Watch. The variable is added to the WATCH window on the left. Lets do this for a few more variables. Now if I hit F10 to step through the code, we can watch the values for those variables update. Now what about Hot Reload? This is something that wasn't originally available in Visual Studio Code, but you can now enable it as an experimental feature. Let's stop the program and go back to Settings from the File menu, Preferences. I'll search for csharp.experimental, and just check the box to enable Hot Reload while debugging. Now let's go to the ExpenseManager class, and I'll set a breakpoint in here. And let's run the application again by hitting F5. I'll choose to total expenses again, and there was already a breakpoint set, so I'll click Continue. And now we're inside the TotalExpenses method. If I change the start value of the indexer variable in this for loop and click Ctrl+S to save the changes, there's this button on the toolbar with the Hot Reload icon. Click that or use the shortcut Ctrl+Shift+Enter. Now if I step through the code using F10, I can see it runs three times, and the total in the Watch window is what we expect it to be. The last thing I'll show you here is the callstack. I'll just continue the code and run the TotalExpenses method again. Over on the left, under the Watch window, there's a callstack window. Just like in Visual Studio, you can navigate up and down the callstack to get a better understanding of how your application is calling methods. So as you can see, Visual Studio Code has many of the same features as Visual Studio when it comes to debugging. That wasn't always the case. When I originally created this course back in 2022, some of these features weren't available in Visual Studio Code. Next, let's configure a web application so we can see some more realistic debugging examples.

Configuring the Sample Project

Now let's configure the sample project in the course downloads. I have Visual Studio 2022 Community Edition installed. This is the free edition. When it opens, I'll choose Open a project or solution and navigate to the folder with the course downloads, so wherever you unzip the code for this module. And once the solution loads, you'll see the code in Solution Explorer. There's a folder here called Migrations, and this is used by entity Framework Core. It's basically SQL scripts in code that will create tables based on the classes in the Models folder. So we have to manually create this database before running the code. And we do that by going to the Tools menu, and under NuGet Package Manager, select Package Manager Console. When the Package Manager Console window opens, just type in Update-Database. If you get an error here that Entity Framework tools aren't installed, you can download and install them by running this command in the Package Manager Console and then run the Update-Database command again. You can see in the Package Manager Console window that some SQL was executed to create objects in the database. If you want to inspect those objects, you can open the View menu, then SQL Server Object Explorer because SQL Server Express is the database that gets installed when you install Visual Studio. If you're using Visual Studio Code and you don't have Visual Studio installed, you'll need to install SQL Server Developer Edition separately. Expand the localdb database, then the Databases folder. There's the new database. And expanding it, you can see the tables that were created. If we compare the classes in the Models folder, there's a table for Expenses, ExpenseTypes, and ExpenseTypeCategories. So the classes in the code will be used to represent the data structures in those tables. Now we can go ahead and run this application. This is an application to track expenses, and it's still in the early stages at this point. There's just some functionality to add, edit, and delete expenses. Lets navigate to the Expenses menu at the top, and this is where any expenses in the database will get listed. There aren't any yet because we haven't created any, but there is some lookup data that was added to the database. So let's click Create New. If I go down to the Expense Type drop-down, there's a list here, and these values are coming from the database. If I select a category from the drop-down list above, which is also coming from the database, then the list below is filtered. So we know there's a working database and the code is able to access it. I'm not going to explain how the code works beyond that because we're going to discover that as we debug the problems we find in the app, so let's start doing that next.

Debugging the Cause of an Exception

So we've been assigned to this project, which is an enterprise web application for managing expenses. And we've been told that there was a problem saving new expenses. So let's run the application and test this out. When we get to the home page, it says enter expenses from the menu item above, so I'll click on Expenses. There's a link here to create a new expense. I'll enter some values here. Let's create a meal expense for lunch today. And the expense was incurred at Mel's Diner for all you 70's TV show fans. I had the gourmet chili, so that was $12.75, and I'll select from the Expense Categories. Remember the expense types aren't filtered yet. They all show by default until we select a category. So I know meals is under expenses, and now the list is much smaller. So this will be a meal. Okay, we've got some data. We're ready to save it to the database. And kaboom, the app blew up, but this is actually a really good thing because we have some error information. It says there was a SQL exception. The INSERT statement conflicted with the FOREIGN KEY constraint, "FK_Expenses_ExpenseTypes_ExpenseTypeID". So we already know from the development team that the database was generated by Visual Studio using Entity Framework Code First, which means chances are this isn't a problem with the database design, but it's still possible that it could be. But we want to check out the code first and rule that out, especially since the database was auto generated. In the stack area here, it shows all the method calls that preceded the error, but this all looks like internal framework stuff. If I scroll down, the exception that was caught in the code says DbUpdateException: An error occurred while saving the entity changes. If we look down the call stack further, there's some internal code again. But then here's where we get to the code in the project that was written by our development team. It shows us the actual line in the code where the exception was surfaced. So, something leading up to this point in the code likely caused the exception to get thrown by the framework when it tried to update the database. We're going to want to look at the code and examine the data that was being sent because a foreign key constraint error implies to me that there might be a problem with the value of the foreign key that was passed in. It says the last line of user code was line 36 in the ExpenseRepository.cs class. So let's go back into Visual Studio and open up Solution Explorer, and let's search at the top for that class. There it is, so let's double-click to open it. So this is a class where the Entity Framework gets called to retrieve, save, and update data in the database. Down at line 36 is where the error was thrown. Let's set a breakpoint at the start of this method. We haven't stopped the application yet, but if you did, that's fine. Just run it again and enter some data. But since the app is still running, I'll just hit the back button to get to the Create page. All the same data is here. When you're trying to reproduce an error, it's a good idea to use the same data again so you know you haven't found a different error because of some different data value. We want to fix one thing at a time here. So, I'll hit Create again, and that brings us to the breakpoint we just set. But how did we get here inside the code? We don't really know the structure of this application yet, right? So that's where the call stack window can help us. I'll go up to the Debug menu, and under Windows, open the Call Stack window. Okay, it shows the class and method that we're in. And right under that is the class and method that called this one. It's the OnPostAsync method in the CreateModel class. If I click that, the file opens, and you can actually see the code is paused at line 62 here, waiting for the expenseRepository method to return. So this method is the entry point for the code when the user clicks the Create button on the screen. That's because this is the code-behind for the Create page. With Razor Pages in ASP.NET, that means there's a .cshtml page that has the actual markup. Let's look for that in Solution Explorer, and let's open it up. Now it says at the top that the model is the CreateModel. That means that the code-behind page is the model whose properties are being bound to this view page. Each of the input items is bound to a property that's preceded by the word Expense, and the property that passes in the ExpenseTypeID is down here towards the bottom just before the Submit button. So if all these properties are preceded by the word Expense, back on the code-behind page, there must be a property called Expense that's being used by the view page, and there it is at the top. It even has the BindProperty attribute to indicate that it's being bound to the page. So ASP.NET is taking care of sending the values from the page back to the code-behind using this bound property. If I hover over it, we can expand this particular instance because we're still debugging, and the code has stopped, and we can see the values being sent from the page. Okay, the data's there, the description, the location, and the price, and the ExpenseTypeID is 4. So there is a value there. That's good. Let's actually add this to the watch window so we can refer back to it because I just want to make sure that all this information is what's being sent to the database. You don't want to just assume that it is. Let's go back to the call stack and return to the ExpenseRepository where the database call is being made. The expense object is being passed in here, so let's just set a watch on this so we can see those values. If I expand this, let's compare it to the expense coming from the view. It looks like everything is the same, except the ExpenseTypeID value is 0 here inside the ExpenseRepository. Somehow, that value isn't getting passed in here. I think we've found the problem, but we'll have to figure out why this is happening, so let's do that next.

Fixing the Bug

Let's go back to the calling code. In the OnPostAsync method, there's some tricky code here. Instead of just taking the expense object that was filled in from the page, the one we saw at the top of the file, the code is creating an emptyExpense and then filling it using this TryUpdateModelAsync method. Then the UserID is being given a value, and I know that's because the team hasn't implemented authentication yet, so this is just a temporary value. But I have a suspicion the problem has to do with this method. Let's figure out what this code actually does. Let's copy this method name, and go to the browser, and let's search for asp.net razor, and the method name. That brings us to the Microsoft documentation. Let's take a look. This can be a little intimidating. There's all these overloads, so how do you know which one we're using? Let's go back to the code. If I highlight the opening bracket and type it in again, I get IntelliSense that shows me that we're on the fourth of seven method overloads, and this one we're using has three input parameters. The first parameter is highlighted here, and if I hit the comma, it thinks we've moved on to the second, and then another comma is the last one. So this overload expects a model, a string, and an array of expressions. First, let's remove these commas, and then let's go back and see if there's a similar overload in the docs. This looks like the one here, it's got the model as the first input, then a string, then an array of expressions. If I click it, it explains each of these parameters better. The model instance to update. Okay, that's the expense that's being created. The name is the name to use when looking up values, and that was expense in the code, which is the name of the property that's being filled in by ASP.NET from the page, so that makes sense. And then the third parameter is the top-level properties which need to be included for the current model. Hmm, let's go to the code. So, these properties need to be filled. If I hover over the Expense class, which we're creating an instance of, and right-click, I'll choose Go To Definition. So there are the properties for the Expense class. This is the class that's being used by the Entity Framework to generate the database too, so there's an ID, a description, a date incurred, location, price, and an integer called ExpenseTypeID. The comment above says this is a foreign key. Then there's the Expense Type object. So depending on the type of query that you write to get the expense from the database, Entity Framework can use the foreign key to also populate the related Expense Type object so you can use it in code. I know that kind of requires some knowledge of Entity Framework, but, just to let you know. Let's go back to the Create page. It looks like we're missing the ExpenseTypeID in this list of properties. So when the new emptyExpense object is getting created, it's missing the ExpenseTypeID from the page. It makes sense that's why the value is never reaching the database. So let's set a breakpoint here, and I'll continue running the code. That causes the error again, so let's back up and try to create the new expense. Now we're paused inside the code-behind for the page. I'll advance the cursor using F10, so the emptyExpense is created, and when we get to this point, the Expense coming from the browser is being used to populate this emptyExpense object. You can see the values up here, and once that's done, we can inspect the emptyExpense object. So the ExpenseTypeID is empty. We've narrowed down the error to the TryUpdateModelAsync line of code. Okay, well we know this is going to blow up, so I'll stop the application, and let's add an expression here to populate the ExpenseTypeID value. Okay, we've included that missing parameter, so let's try running the code now. When it opens up, I'll go to Expenses and create a new expense, and I'll just skip ahead to where the values are populated. I'm using the same values again so I can try to recreate the exact same problem. When I click Create, it goes to the code-behind that handles the Submit button, and let's step through again using F10. When we get to the line of code that calls the database code, I'll hover over it to check the data, and the ExpenseTypeID is now populated. F10 steps us into the ExpenseRepository code because we still have that breakpoint set, but I'm just going to click the Continue button in Visual Studio. And it looks like the data has been saved because we've been redirected to the list of expenses. This new expense is coming from the database, so we know the data saved properly. We've successfully fixed the error. Of course, we'd want to do some more testing to make sure there aren't other errors with different data values, but our goal here was to debug this particular error.

Debugging a Functional Error

Now that we have the Create page working, let's test the Edit page. I'll open up this existing expense, and let's change all the values of these fields to make sure that the record is updating properly in the database. I'll change each of the values so we'll know if an individual field didn't update. I don't need to change the Expense category because that doesn't get saved to the database; it's just used to filter the expense type list. Okay, now let's save these changes. When the record is saved, the page redirects to the list of expenses, and we can see the values have changed for the record. The Date is updated, along with the Expense Type, Description, Location, but the Price is showing 0. That's not what we entered. So there's something going on here. Let's go into the code and up to Solution Explorer. I'll clear the last search, and the Pages folder is open. That's where the Create page was, and the Edit page is a little farther down, so let's open that up. And it looks pretty similar to what you saw on the Create page. There's this Expense property, which is bound to the .cshtml page. And farther down, there's a method on PostAsync. It's structured pretty much the same. So since we saw how the expense is populated on the Create page, and we've already debugged an issue there, we know that we might be looking at the same cause for a different issue. I'll set a breakpoint here, and let's go back to the running app and edit the page again. I'll update the price, and let's save this. The breakpoint is hit on the Edit page. So this method is handling the Save button click on the server side. Let's step through the code with F10, and before we send the new expense to the Repository class, let's hover over the object and check out the values of its properties. And the price is 0 here. If we look at the TryUpdateModelAsync method just above, we can see already that the price property is missing from this list. Of course, if we didn't have the previous experience of debugging the exception where the expense type id was missing, we could use the same approach to debug this problem, but we'd have to understand where the button click is being handled in the server-side code, because we didn't start with an exception this time. But you can see this is where debugging starts to build up your knowledge of what can go wrong in an application. If a mistake was made somewhere, there's a chance the same type of mistake can show up in other places. Let's add that property, and run the app again. When the app starts, let's navigate to the same expense and try and update the Price field again. The breakpoint is still active, so let's step through the code and make sure we're seeing the price updated in the object that's being sent to the database. And now it is, so I'll click Continue and let the application run. When the page redirects, the Price field has now been updated. So, we saw the same type of fix to what at first appeared to be a different problem. There was no exception in this case. The application just didn't behave the way we expected it to. And because of our experience debugging this application, we had a sense of where to look because debugging helped us to gain a better understanding of the structure of this application. By debugging and stepping through the code, we learned that the page's code-behind transforms some data and calls another class to save the data. At this point, you might research some of the things you saw in the application and ask why it's being done this way. This isn't a course about ASP.NET, but I'm intentionally not using simple examples here. We're using an enterprise-type application, because you'll get dropped into applications in your career that you may not understand, and debugging will help you learn how the application works. But let me explain the structure a bit. First, why isn't the code just passing the expense object that's being populated from the page right to the repository to save into the database? And second, why is the code not just calling the Entity Framework right from here in the code-behind? Why does this need to be so complicated? Well, enterprise applications often go beyond simple design because of things like security and testability. The reason this TryUpdateModelAsync is used is so a malicious user can't pass hidden values over HTTP. There might be a certain field that you don't want the code to update, but if you just take whatever is sent from the browser and save it to the database, you're trusting the user input when you shouldn't. And the reason the Entity Framework code is being kept in a different class is for testability. Unit tests can use a mock version of the repository that doesn't actually call the database. It just returns the same values from the code each time. That allows you to only test this class in a unit test without testing all the way to the database. You're going to learn about unit testing later in this path on C# programming. For now, you've successfully debugged an error in an ASP.NET web application. Next, let's talk more about some of the benefits of debugging before we move on to more advanced topics in the next module.

Becoming a Better Debugger

Now that you've seen some debugging in action, lets talk about the benefits of debugging in addition to just getting your code working. Of course, there's the obvious benefit of supporting applications, being able to fix problems that are preventing the application's users from doing their work. But I'm talking about the benefits of becoming better at debugging for you, the developer. The process of debugging is actually an opportunity. As hard as it may be in the moment, you need to try and see it as such. First of all, debugging is an opportunity to learn about how the program works. This will especially help you if it's not code you wrote yourself. Understanding the code is key to fixing it, to solving problems in the code in the future, and to potentially refactoring the code if you're given the opportunity. Next, debugging your own code helps you learn about the kinds of mistakes that you make. It exposes your weaknesses, which allows you to improve. Ask yourself questions like why you made the error and how could you have prevented it? But also, you can learn about how you solve problems. How could you have found the source of the bug more quickly? Are you following a systematic approach or just guessing randomly? We're going to talk about a systematic approach to debugging in the next module. You can also learn about how you fix defects. Do you just put bandages on the problem, maybe adding if statements for special cases that change the symptom, but not the problem? Or do you take the time to understand the code beyond just the immediate code around the problem? Bugs are an opportunity to understand the program itself, but also an opportunity to understand your approach to finding and fixing problems in code. And that analysis will not only make you a better debugger, but also a better programmer who can avoid some of the same errors in the future. The more experience you get with debugging code and solving problems, the better you'll get at it. When there's a bug in the code, your first instinct might be to panic. Don't. You can figure this out. It might not be fast, and it might not be easy, but you don't need to run and ask someone else how to fix the problem. While it's okay to ask for help when debugging, don't make that your first step. Take the time to put in a real effort to solve the problem yourself. It's worth the time. In this module, you've learned some of the basics of debugging. We looked at different categories of bugs, and you learned about features in Visual Studio and Visual Studio Code that help with debugging. Then you saw how to use those features in a web application to debug an exception and a functional error. In the next module, you're going to learn about more tools for debugging, and you'll also learn a scientific approach to debugging that reduces the need for guesswork.

Debugging Data and Inputs

Common Debugging Traps

Now you've seen some basic tools for debugging in Visual Studio and Visual Studio Code. We'll get more into the debugging capabilities of those tools throughout this module, and we'll also discuss how to approach the diagnosis and resolution of a defect or bug. But first, I want to talk about what not to do because a lot of people don't know how to debug. So without a proper methodology, this may be where they start. You might see some things here that you've done in the past, but don't feel bad. We've all been there. This is just to point out some approaches that may be tempting, especially when you're under pressure to fix a bug and especially if you don't have a proper methodology to replace these approaches. Remember, debugging isn't just about tools. It's about learning how you investigate and solve problems in code and getting better at it. This list is based on what Steve McConnell, the author of Code Complete, calls the Devil's guide to debugging. You don't want to find the defects by just guessing by changing code until something seems to work. That means you don't understand the code, and even worse, maybe you're not keeping track of the changes you're making, so you could be introducing new bugs along the way. And if you're going down this road, you're probably not backing up the original code either, which makes this approach even worse. You need to calm down and develop a plan, and we'll talk about how to do that in the next clip. Next. don't assume the problem is trivial and you don't need to understand the code in order to fix it. You might have to fix someone else's code, and you're under tight time constraints to do it. It might seem like a waste of time to look at the module of code and really understand what it's doing. But trust me, it's not a waste of time. That doesn't mean you have to understand every line of code in the whole application before you start debugging or before you make a change, but you do need to understand the code that you're planning to change and the code that's right around it. You'll get better at narrowing down the problem to a certain part of the codebase as you get better at debugging. Next, don't put a band-aid on the problem, meaning don't code for special cases. If a certain value of a variable causes the loop to crash, you need to find out why. Don't just add an if statement to account for that particular value. I've actually seen that in production code, and it's scary because it means the programmer didn't understand the problem and didn't care to. No matter how much of a rush you're in, you're just asking for trouble with an approach like that. Next, don't debug by superstition. This is when you assume that it's the computer's fault or the compiler or it's a problem in the .NET Framework or because of a full moon or anything else that's not the code. Let's face it. It's most likely the code. Look, it's hard enough to find a defect when you're actually looking for it. If you assume that your code doesn't have any errors in it, they're going to be even harder to find. Debugging by superstition can also extend to trying to blame others before you've really done your homework. Don't be that person that right away blames the network or the server or the operations team or the deployment or other programmers on your team. It's possible it could turn out that someone else did make an error or something is misconfigured, and it might even really look like that from the start. But you should first investigate your code with an open mind if there's any possible chance that that could be where the problem is. This will also improve your credibility in your organization. You won't be that person who right away points the finger at everyone else. And if it does turn out that your code is the problem, you won't have the embarrassment of having to admit it later after having blamed something or someone else. Yes, it's embarrassing when an issue in the application is caused by your code or when you break the build. But when you admit it and get to work solving the problem, you'll gain people's respect. We all intrinsically know that none of us are perfect, and people respect the person who focuses on fixing the problem, not trying to shift the blame. So, we've talked about some of the wrong ways to approach debugging. In the next clip, we'll talk about an effective approach to debugging based on the scientific method. Then, will work through an example in Visual Studio using the web application from the previous module. First, we'll determine the factors in the interface that seem to cause the error, and then we'll work on a theory of what might be causing the problem. Next, we'll keep refining our theory using debugging tools in Visual Studio to inspect the data. You'll see features like the Autos and

Locals windows. Then you'll learn how you can execute test cases by modifying data right in the debugger. You'll see how to do that in the Autos, Locals, and Watch windows and also in another really powerful window, the Immediate window. And then, when we locate the source of the defect, we'll fix it in the code, test the fix, and look for similar errors in the code. We're going to learn a lot about debugging data in Visual Studio. So then for completeness, we'll take a look at some of the capabilities in Visual Studio Code for debugging data. Next, let's talk about an effective approach to debugging.

An Effective Approach to Debugging

Now how should we approach debugging? Programming is computer science, so let's look at the scientific method of solving a problem. The scientific method usually starts with a question. But in debugging, the bug is really the question, what's causing this error? So first, you gather data through repeatable experiments and analyze the data. Next, you form a hypothesis that accounts for the relevant data, then you design an experiment to prove or disprove the hypothesis. Next, you execute the experiment to prove or disprove your hypothesis and then analyze the data and refine your hypothesis and repeat the process as needed. You might see these steps presented in different ways or additional steps added like publishing your results and having them peer reviewed. But the principles here are the same, so let's not get too academic about it in the discussion board. Now how do we translate this to debugging code, which is what we're here for? First, we need to stabilize the error, and that's about repeatability. The defect is easier to diagnose if you can make it happen regularly. That might mean iterating over the whole scientific method and coming up with new test cases, datasets or paths through the application until you can stabilize the error. Next is locating the source of the error, which means using the scientific method again, gathering data that produces the defect, analyzing the data, and forming a hypothesis and then determining how to prove or disprove that hypothesis. Executing your experiment, in this case, probably means examining the code, either by reading it or by stepping through using your debugging tools. Next, you found the source of the error, so you need to fix the defect. Really at this point, you've done about 90% of the work already because you've gained an understanding of the problem and found its cause. Fixing a bug is often trivial, assuming you understand the technology that you're coding in. After you've fixed the bug, you need to test the fix. Another tip here from experience, please test your code. Don't just throw it over the fence to the testing team, assuming that you have one. And after you've tested the fix, you should really look for similar errors in the code. If you wrote the code, you may have found a blind spot for yourself as a developer, and you can prevent a similar error from happening again in another part of the codebase. If its someone else's code, chances are pretty good that they might've made the same mistake somewhere else too. I am borrowing a lot of this again from Steve McConnell's famous book, Code Complete. So, this isn't just the opinion of some guy on Pluralsight. That book is a pretty standard text in programming, and I've found in 20-odd years of debugging that this is a solid approach that won't fail you. Lets talk a bit more about stabilizing the error in the next clip.

Stabilizing the Error

The first step in our approach is stabilizing the error. It's about repeatability, of course, making the error happen consistently. But it's also about narrowing the test case to the simplest one that still produces the error. You want to get it to the point where changing any aspect of the test case changes the behavior of the error. To simplify the test case, you can use the scientific method of hypothesis, testing, analyzing the results, and repeating if necessary. If there's multiple factors that when used together cause the error, you need to form a hypothesis about which factors aren't relevant, and those factors could be things like certain data fields that are entered or a certain navigation path through the app. You might go to the effort of setting up unit tests, or you might be manually testing the code using the user interface. Stabilizing the error isn't about how you test. It's about the thinking process of what to test. So when you change the factors and rerun the test, if you still get the error, then you know you can eliminate those factors and you've just simplified the test. There may be experimentation involved to find the data values or steps that cause an error, particularly if the error was reported by someone else and there wasn't a lot of detail. Keep notes on what you've tried and what you want to try because it'll get confusing really fast. Reproducing the error is often done through manual testing that you perform in your dev environment, whether on your local computer or on a test server. If possible, you want to avoid testing in the production environment. Of course, if you're debugging during development, which you'll be doing a lot of, this isn't an issue. But if the error is happening in production, hopefully there's some logs that show the code where the bug is occurring and the data that causes it. We'll talk about how to code your application to do that logging later in this course. When there aren't logs available, you may need to gather information from users that have encountered the error, and you'll need to try your own experiments to reproduce the problem. If possible, watch those users use the system. They may be using it in a way you hadn't thought of. Users can be very creative when they have a job to do. Just because a requirement wasn't documented, they might've figured out a way to accomplish it in the system anyway, and that can have side effects. A lot of times, problems are caused by data values that weren't expected when the code was written. If you get those values from logs in the production application or from information provided by users, that's great. But when there's a complex calculation, you might need more data. In the worst case, you may need to copy the database from production into an environment where you can test against the real data that's causing the problem. I've done this in the past, but it's an extreme solution. It shouldn't be your go-to, and not just because of the work involved. There are risks and challenges associated with this, depending on the sensitivity of the data in production. Make sure you discuss it with your security people and other stakeholders like the business owners of the data because sensitive data should never be copied into environments that don't have the same controls as production. At the very least, the data may need to be depersonalized by scrambling personally identifiable data. Again, I'm not suggesting copying the production database as your first choice when debugging. It's just one of several options that can help you reproduce the error, and it comes with some challenges to be aware of. Intermittent errors are often the toughest to track down. When they're really tough, it can often be a timing issue. Concurrency problems are a good example where multiple users are accessing the same piece of data at the same time. That's a whole special case, and there are design patterns that can help you prevent those types of errors. I've got an entire course called Enterprise Patterns: Concurrency in Business Applications. It's a few years old, but the code is C# and ASP.NET Core, and the patterns are from Martin Fowler's book, Patterns of Enterprise Application Architecture, so you can apply them to any programming language. Lets look at working through a real debugging session. Along the way, you'll see how to use more debugging tools in Visual Studio.

Debugging a Calculation Error in Visual Studio

Let's start with stabilizing an error, which is about repeatability. In this clip, we'll mostly be testing the app using the interface, but we'll also look at another way to gather information before we examine the code by querying the underlying data in the database. That will help us form a theory that we can then test in the subsequent clip. For this module, there's a new version of the web application. The database hasn't changed though, so you can just download and run the new code. Let's run this, and when the app opens, go to the Expenses page. As you can see, there's been some updates here. There are now fields that can be used to filter expenses by the various properties of the expense. There's also data here, and this is getting added to the database automatically when you first run the app. The dates will be different for your instance, but the same entries will be here, plus any that you've added previously. So we can filter on things like the Expense Type. Let's search for Meals & Entertainment, and we can further filter these results. Let's look for expenses where the description includes the word lunch. Now let's look for the lunch expenses between February 1st and the end of February. Now you can also see that this page calculates the total of the filtered expenses. That's going to be helpful in a second, but let's clear these two filters and get all the expenses in February. There aren't a lot, but the total here at the bottom says that they amount to $226.98. Up at the top of the page, there's a new link added called Reports. I'll actually right-click on this and open it in another tab. So this page lets us run reports on all the expenses for a particular year and month. Let's run this. It defaults to the current month and year, and it looks like there aren't any current month expenses in my case. We can select a different month and past year, and all the expense types are listed here. There's a column for the number of expenses for that type within this month and the total for each expense type. Lets try this for February. Okay, now we have some data. There are expenses entered related to Meals, Supplies, and Fuel under the Vehicle category. There's a subtotal for each expense type, and all of those subtotals add up to $156.77. Okay, wait. When we ran this for February on the other page, it was a different number. Let's go back. It looks like we've got a bug on one of these pages. We've identified it ourselves here. But this might come in from the testing team or from users during user acceptance testing or worse, after the app has been deployed into production. Either way, we need to debug this issue. We could start jumping into the code, but we don't really understand the issue yet. We need to gather more data and try to form a hypothesis. There's not a lot of input we can vary on the report page, just the month and year. So let's filter these expenses by a different month. I'll set this up to get the expenses for the month of March. That comes out to $292.17. Now let's go to the Reports page, and let's run a report for March. It's $292.17, so the report is fine for March, and there's a problem with February. Well, February is a funny month because it's shorter, and sometimes there's a leap year. So is it possible this problem only happens in February? Let's test that hypothesis and gather some more data. I'll go back three months to January, select the start and end of the month, and filter the list. So it totals $188.30 in expenses for January. If my hypothesis about February is correct, January should total the same on the Reports page, so let's try that. And it's a different total, $154.08, and it was $188.30 on the Expenses page. So, we've eliminated the theory that this has to do with February being a short month or having something to do with leap years. If we had more data in the database, we could verify that with other months, but let's move on. Now at this point, I want to be 100% sure which page is calculating correctly so we can further narrow down where the problem is. Since we have access to the database, we can calculate the totals directly from there. That'll take the code out of the equation. You can do that in SQL Server Management Studio or right from within Visual Studio. Let's go up to the View menu and select SQL Server Object Explorer. I'm already connected to my local SQL Server instance, which gets installed with Visual Studio. So, I'll expand the nodes, and there are the tables that were created by Entity Framework Code First. Let's run a query by right-clicking on the database and choosing New Query. I'll do a SELECT SUM, and we want to sum the price column, and this is from the Expenses table where the DateIncurred column is greater than or equal to a certain date and less than or equal to a certain date. I can never remember the right date format, so let's go back to Object Explorer. I'll right-click on the Expenses table and View Data. Then I can just copy one of these dates. And notice that the dates all have a time of 12 AM. That's done by the code, so we don't have to worry about factoring times into our queries. We can just query the item based on its date. I'll update the start date and do the same for the end date, changing the day to the last day of the month. And remember, the expenses all have a time of 12 AM in the database. So as long as the query is less than or equal to this date, any expense on this end date will be included. You could change the time to 11:59 PM and 59 seconds if you wanted to include later times on that date, but it's not necessary for this particular application. Now lets run the query. The sum is $226.98. I'll go back to the running app. And on the Expenses page here, the total is $226.98. So this page is getting the correct result. Let's go back to the Reports page and run the report for February again, and it's $156.77 on this page, so definitely not the correct amount of $226.98. All right, so we know which page has the problem, and we know that this happens for more than just one month. Where do we go from here? Well, it could be a problem with the addition calculation on the Reports page, I suppose. But what other data have we gathered in our testing? Well, in both instances where the total amount was different on the two pages, January and February, it was always less on the Reports page. That leads me to think that maybe we're dealing with less records on the reports page. So, rather than testing that theory using the interface, let's see how we can use our debugging tools to do it. We'll do that next.

Inspecting Data in the Debugger

We've gathered some data from the interface and determined that the Expenses page is getting the correct results and calculating the total correctly, and the Reports page has an issue. The current theory we have is that the Reports page may not be using all the expenses for February or January because for each of the problem months, the total on the Reports page is less than the total on the Expenses page. Of course, our theory could be wrong, and there may be another cause. But let's try to prove or disprove this theory using the debugging tools in Visual Studio. Lets start by opening up the codebehind for the Expenses page, which is the Index.cshtml.cs file. Remember, we know this page is calculating the total correctly, so we also believe it's using the right record set to do that. At the top of the page are the properties for the search fields on the screen that are used to filter the results, the SearchDescription, SearchExpenseType, the start and end dates, etc. And we can see the controls on the view page that those are bound to. Then there's a collection of expenses, a double that holds the AmountTotal and a SelectList, which is used by Razor Pages as the data for a drop-down list. Again, this is for the filter criteria. OnGetAsync is a method that's called when the page posts back, so whenever the filter button is pressed to filter the results. The _expenseRepository is called by calling the SearchExpenses method and passing in the properties bound to the view. Let's set a breakpoint here and then run this by clicking the Filter button on the view. Now we can inspect each of these variables individually by hovering over them, just like we did in the previous module. But Visual Studio has some debugging windows that can make this a lot easier. Let's go up to Debug, and then under Windows, open the Autos window. Autos shows variables around the breakpoint. That'll make more sense shortly. Let's open up another window, the Locals window. At this point, it looks like they do exactly the same thing, but you'll see shortly that that's not true. Let's pin this so it stays visible. I'll hit F10 and advance to the line where we're going to call the repository to retrieve the expenses. If I go over to the Autos window, now there's a bunch of variables showing with their current values. So, there's the Expenses variable, which is empty because we haven't called the repository method yet to populate it. And there's also properties that we defined higher up, which are bound to the view. So these are the values that are coming from the view. The string properties are null because there was no criteria entered, so the SearchDescription and SearchLocation, but the start and end dates are filled with February 1st and 28th. The Autos window shows the variables around the current breakpoint. So as we step through the code, the variables showing in this window will change, depending on where we are. The Locals window, on the other hand, shows variables defined in the local scope. So this is the entire IndexModel class. That's the name of the class that defines the codebehind on this page. And that class inherits from an ASP.NET Razor framework class called PageModel. So besides the variables that we've defined on the page like Expenses and AmountTotal, this also includes things like the HttpContext, the ViewData collection, Request and Response data, and other ASP.NET-related state information. You may not know much about ASP.NET, but you can actually learn a lot about what information is available by examining these properties. Okay, let's go back to the Autos window, and I'll hit F10 to advance the cursor. Now the Expenses collection has been populated from the Repository class. You can see that in the Autos window there were nine records returned. And if I hover over the variable and expand it, the records are also available here. In the last module, you saw how we can drill into each object to see the values of its properties. Something that can be really handy is that you can choose which of these properties to see in the higher-level overview by clicking the little pin icon next to the field name. So if I click the pin next to DateIncurred, when we look at the list of expenses, the value for DateIncurred shows, and you can show multiple properties here, so let's also pin the price, and let's expand the ExpenseType, which is a child class in this object. And let's choose the property we'd like to show from here, the Name. Now we can pin the ExpenseType object, which will promote the field to the main list. Now at a glance, we can see the Date, Price, and ExpenseType for all the expenses in the list. And that view of the data is the same in all the windows too, so we don't have to recreate this view in the Autos window. If I go to the Expenses collection, right-click and Add Watch, you can see those properties are showing in the overview here also. Of course, we can still drill into each expense, and we can unpin these properties and pin others if we like. Another way to add a watch is to just type in the variable name, so let's add a watch for the AmountTotal, and you can type in expressions here too. As I step through the code with F10, this AmountTotal updates in the Watch window. But you've seen that in the previous module. So let's go down a little further in the code, right-click, and Run To Cursor. That'll get us past the foreach loop. Okay, let's open the Autos window again and hit F10 to advance, and now the variables have changed. The ExpenseType list has been populated, and the SearchDateStart has the lowest date from the expenses for the user. And as we advance, the SearchDateEnd variable is added, and other variables higher up fall out of scope. Let's F10 to get out of this method, which continues execution of the code. If we go back to Visual Studio, all of the windows are disabled because the application isn't stopped at a breakpoint. Now let's go over to the Reports page, and let's select February for the month. Before we run this report, let's create a breakpoint in the code. You can see in the URL that the path is to the Report view. So let's go into Visual Studio and open up the codebehind for that view. There are properties that are bound to the view here also, and farther down is the OnGet method. So lets set a breakpoint here and go back to the interface and run this. We can get enough information about the code by reading the comments. BuildFilterChoiceLists will populate the Year and Month drop-down lists. Then the code initializes a collection of objects for displaying the summaries on the view. If its not the first time the page is loading, we go to the else statement, which creates the start date based on the year and month selected, then uses a static method on the DateTime class to get the number of days in the month for the year selected and then uses that to create an end date. Lets hover over the start date. And so we can still see this date after we move the cursor, We can actually pin this data tip so it stays visible. The days in the month are 28, and lets pin this end date also. After we move through the code with F10, the list of expenses for this month has been populated, and we can actually see that in the Watch window because remember, we set a watch on the variable named Expenses when we were debugging on the Index page. So, those expenses are gone, and they've been replaced with the values of the collection with the same name on this page. So the Watch window is just matching names, not storing the actual instance of the collection. We've lost the old collection to compare this to. There is a way to keep the state of variables though, so let's see how to do that and continue our debugging in the next clip.

Modifying Test Data in the Debugger

Let's click Continue to resume execution of the application. We want to be able to compare the collection on the Expenses page to the collection on the Reports page. So let's go back to the Expense page and run this query again, and I'll right-click further down and Run To Cursor. So the Expenses collection has been populated. Now I want to go to the Autos window and right-click on the Expenses collection and select Make Object ID. What this does is let you track a specific object after the variable has gone out of scope. This works for reference types in C#. We could add this to the

Watch window now, or it actually gets added automatically to the Locals window, so we'll just use it there. Remember, the count here is 9, and we still have the AmountTotal in the Watch window showing the sum of the expenses. Let's continue the code and go to the Reports page, and let's run this report. That brings us to the breakpoint. The Expenses in the Watch window is just the Expenses variable. This isn't the stored object from the Expenses page, so it shows as null. I'll run this code to the cursor further down. And now the local Expenses collection shows 6 expenses, and we can expand those. But the Locals window still has the object ID that we created. So it's showing the Expenses collection from the other page, which has 9 Expense objects in that collection. So we've proved our theory that the Reports page is doing its calculation based on less records or expense objects, 9 expenses versus 6 expenses on the Report page. Now let's compare the calculated total. On the Report view, that's stored in this ExpenseTotal field on the model, which means there's a property on the code-behind called ExpenseTotal. That's this field here on the view. So let's navigate back to the breakpoint, and let's add the name of that variable in the Watch window. Notice we still have the AmountTotal from the Expenses page, but it's grayed out because this is just the last value that was stored here, and there isn't another value on this page with the same name. If I hit F10 to move past the BuildSummaries method, the ExpenseTotal gets populated, so the calculation must be happening inside that method. Since we have the list of expenses from both pages, we could just manually compare them and figure out what's missing, and that might help us figure out the source of the problem. But let's pretend we have a much larger dataset, and they would be hard to compare manually. Could we make the dataset smaller and still cause the error to happen? Let's try that, if only because it will illustrate some more debugging tools. Let's go to the Expenses page, and let's choose a smaller window for the expenses. I would probably start by selecting all the expenses in the first half of the month. But since the first expense here is on February 23rd, let's start at the 25th and go to the end of the month, and I'll click Filter. That brings us to the breakpoint. I actually need to remove this object ID because it won't get repopulated, and we don't want to get confused by seeing the old data. So let's Run To Cursor again. And on the Autos window, the Expenses collection now shows 5 expenses. I'll right-click and Make Object ID, so now that's available on the Locals window. And on the Watch window, the AmountTotal has been updated, too. We can verify that by hovering over the variable in the code. Now let's continue execution and go to the Reports page, and let's run the same query. Let's run the code right up to where the expenses are retrieved from the repository. Now, there was no way for us to specify specific dates in the interface. We could only choose a month. It's actually possible to modify the values of variables in the code that's being debugged. You can do that right inside the Autos, Locals, and Watch windows. Just double-click on the value and update it right here. So, I could update this user ID, which is just an integer, and you can see that the new value will get sent to the repository, so I could retrieve the expenses for a different user. We don't want to do that though, so I'll change this value back. So let's try that for the startDate. I forgot what the start date was that we used. I'll just go back to the Expenses page. Okay, so it was February 25th. I'll update this date to the 25th, and click Enter, and we get an error. You can't actually update dates using this approach, but you can type in expressions. So we can actually run code inside this editing window. I'll type in new DateTime and pass the parameters for the year, month, and day. Now the value of this DateTime variable has been updated. You can see that if I hover over it, but I also want to show you another way to do this in Visual Studio. I'll just change this value back for now and make sure the value was updated by hovering over it. We can use another really powerful debugging window called the Immediate window. Under Debug, Windows, it's right under the Watch, Autos, and Locals windows. I use this window a lot, actually. You can inspect variables in the code just by typing their name and hitting Enter, and you can run code right in here, too. So we can give the startDate a new value by saying startDate = new DateTime, than the year, month, and day. So we'll reset the value to February 25th. Now if I hover over that variable in the code, you can see it has a new value. I'll hit F10 to advance the code, and you can see that there are only two expenses in this collection. Remember on the other page, the collection had five objects. Let's advance the code to populate the ExpenseTotal. Okay good, so the calculations are still different. That means we've still got enough data to cause the problem to show itself. Now we can manually look at the two collections and try to identify the differences, which might lead us to a new theory of why this is happening. Let's do that in the next clip.

Fixing the Defect and Looking for Similar Errors

The first thing I noticed is that both of these expenses from the Reports page both have a DateIncurred of February 25. The collection from the search on the Expenses page shows those same two expenses from February 25th, but also these other expenses from February 28th. And the 28th is the end date of our query on both pages. So, it looks like the Reports page is filtering out expenses with a DateIncurred that equals the end date of the query. Let's go to the Reports page, and let's drag the cursor back up to the call to the repository method that retrieves the expenses, because I think that's where the problem is. This time, I'll hit F11 to navigate into the method. And there's not much code here. You may not be very familiar with LINQ yet, but let's try to make sense of this. It's retrieving expenses for the userId that was passed in, where the DateIncurred is greater than or equal to the start date that's been passed in, and the DateIncurred is also less than the end date. Oh, okay. This operator should be less than or equal to. Right now, the query is only retrieving expenses where the DateIncurred is lower than the end date, which is why the expenses for February 28th are getting filtered out. So this will happen whenever a report is run and there are expenses on the last day of the month. So let's stop the code and let's update this. And before we test again, let's go to the Debug menu and Delete All Breakpoints. That way, we can just run the code in the interface and see if it's working as expected. Let's run the app. And when it opens,, I'll go to the Expenses page and let's filter for all the expenses in February again. And the total is $226.98. Now let's open up the Reports page on another tab again, and let's run the report for February. And now the total is $226.98. Of course, we'd want to test this for January, also. So let's do that. I'll update the start and end date on the Expenses page, and the total is $188.30. Now let's go to the Reports page and run the report for January. And the total is the same here now too. Remember, it wasn't before. To be thorough, I'd also want to run the tests that passed before too, just to make sure I didn't introduce any new problems with the fix. So let's run the report for March. The total is $292.17. And on the Expenses page, let's change the dates to get the expenses for March. And the total is the same, $292.17. We've successfully fixed the defect and tested the fix. The last step of the approach that I outlined earlier in the module is that we want to look for similar errors. If I made this error in my code once, there's a good chance I did it somewhere else, too. Let's look at the Expense repository class. I'll just quickly skim over the link queries where there are comparisons. And sure enough, I have less than here for the price filter, where there should be a less than or equal to operator. In the next clip, let's look at some of these same debugging tools in Visual Studio Code.

Debugging Data in Visual Studio Code

Now let's see how to accomplish some of these same tasks in Visual Studio Code. In this demo, you'll see how to inspect data in the Variables window, which is similar to the Locals window in Visual Studio. We'll add variables to the Watch window, store variables that have gone out of scope, similar to the Make Object ID functionality in Visual Studio that you saw, and then we'll modify input values right in the debugger. Let's start by opening Visual Studio Code. You can navigate to the folder where you stored the web application for this module. Since I've already opened that folder, I have a shortcut here. When it opens, I'll expand the folder and open the Pages and Expenses folders. Now I can edit the codebehind for the Index page and the Reports page. Lets run the application. But before I can do that, I need to generate the build and debug assets, which is a new folder in the project folder behind the scenes. Now this is the same code that was created in Visual Studio. So it's using the SQL Server Express LocalDB instance that Visual Studio installs, and the config file still points to that. I haven't done anything different for VS Code. But if you only have VS Code installed, you'll need to download and install SQL Server Express or configure the solution to use a different back-end database. Let's run this. And when the app opens, navigate to the Expenses page, and let's update the dates of the search so we can just get the expenses for the last few days of February. Before I run this, I just want to create a breakpoint on the Index page. Okay, now lets filter the search. I'll go back to VS Code, and we're stopped at the breakpoint and in debug mode. The Variables window on the left shows a Locals subfolder, and the top-level variable is this, just like the Locals window in Visual Studio, this being the class of the codebehind page. If I expand this, there are the same variables we saw in Visual Studio, the variables created in code like AmountTotal and the Expenses list object, as well as ASP.NET data that's inherited from the parent PageModel object. Here's the start and end dates for the search that were entered in the view. If I hit F10 to populate the Expenses variable from the repository, this collapses in the locals. But if I expand it, you can see the collection was populated. I can right-click this collection and Add to Watch. Now it gets added to the Watch window. You can also add variables and expressions to the Watch window manually. I'll type in the AmountTotal variable. Now rather than step through the sloop, I'll right-click farther down and Run to Cursor. Now AmountTotal has been populated in the Watch window. Now when we navigate away from this page, say to the Reports page, the value of Expenses in the Watch window here will get overwritten by the collection with the same name on the Reports page. And if we right-click on the Expenses variable, we don't have an option to Make Object ID for this collection like we did in Visual Studio. But using the Debug Console, we can create a new variable that's a copy of the Expenses variable. I'll call it SearchPageExpenses, and don't forget to add the semicolon at the end. Now that variable shows in the Locals window at the top. It has the collection of expenses. And if I right-click on here, I can add this to the Watch window. Okay, now let's continue running the code. Let's open the Reports page codebehind, and let's set a breakpoint here where the report is generated. Now let's open the Reports page in the running application, and lets run the report for February. I'll step through up to the point where the Expenses collection gets populated. Notice in the Watch window that this.Expenses is null because it's now referencing the Expenses collection on the Reports page, which is the variable that's in scope now. But the SearchPageExpenses collection in the Watch window still shows a count of 5 because this is a copy of the expenses from the other page. Now let's try and modify the date value of the startDate variable. I'll double-click on it in the Variables window and try to change the date, so same issue as in Visual Studio. So let's try entering an expression here to create a new DateTime with the day starting on the 25th, and that works just like in Visual Studio. If I hover over the startDate variable in the code, you can see that its been updated. I'll hit F10 to advance the code. And now there are two expenses in the collection, and the Expenses collection from the other page still shows the five expenses that were retrieved for the same date range. We can expand each Expense object and view its properties. But unlike Visual Studio, we can't pin any of these properties to the parent list view to make it easier to compare the values. There is a way we can do this by modifying the code though, and you'll be seeing that later in the course in the module on coding for debugging. In this module, you learned about some of the traps you can fall into if you don't have an effective approach for debugging. Then we looked at an approach that you can use for debugging based on the scientific method. And as a reference for this approach, you can check out the book Code Complete, Second Edition from Microsoft Press. Steve McConnell is the author. It was published in 2004, but the concepts are just as relevant today. Next in this module, we did some demos to illustrate debugging data using the tools in Visual Studio. You saw a calculation error and how to stabilize the error to make it repeatable. Then we inspected data in the debugger using debugging windows in Visual Studio. Next, you saw how to modify inputs right in Visual Studio while debugging. When we found the source of the defect, we fixed it and looked for similar errors in the code. And finally, you saw how Visual Studio Code allows you to do many of the same tasks as Visual Studio when it comes to inspecting and debugging data. In the next module, we're going to talk more about breakpoints and tracing data using code from the System.Diagnostics namespace in .NET.

Advanced Breakpoints and Tracing

Introduction

Throughout the course so far, we've used breakpoints to slow down the execution of the code and inspect program logic and data values. In this module, we're going to expand on breakpoints by looking at some features of Visual Studio that allow you to create breakpoints that only pause program execution when certain conditions are met. You'll see how to log data to the output window using a variation on breakpoints called trace points and see how breakpoints can be set to only break execution when other conditional breakpoints have been hit. We'll work through an example to show how these features can be useful. Then we'll look at a different approach to trace points that lets you log to more destinations than just the output window. You can add code to your application specifically for debugging using the System.Diagnostics.Debug class and write the output of those statements to places like text files, XML files, the console window, and even the Windows event log. Trace listeners are the mechanism to do that. So let's get started by looking at some of the changes to the web application that we'll be using in this module.

Updates to the Sample Web Application

There's been some changes to the app for this module. You can find the updated web application in the course downloads. The database hasn't changed though, so you can just update the code. Let's run this and take a look. I'll just expand this window, and the change is on the Reports page. Now if you're following along with a downloaded code, the dates for your data will be different. The sample application creates data, starting in the month that you first run it and working backwards. So the bug that's been introduced is in the previous month's data. I first ran this code in April to create the database, so I'll go back to March for the example. You can't actually see the difference, but behind the scenes, there are some business rules being implemented. Different expense types can have different business rules that affect how they calculate. For example, when an employee submits a meal expense, it can only be up to a certain maximum, depending on if it's breakfast, lunch or dinner.

Let's look at how this is implemented. I'll open up the codebehind for the Reports page and scroll down to the OnGet method. This gets called when the page first loads and whenever the Run button is clicked. First, the Month and Year drop-down lists are built. Then a collection is instantiated, and this collection is a list of ExpenseSummary object instances. That class is defined in the same file, so we can jump to it from the drop-down at the top of the window here. So ExpenseSummary just has a name, and it contains a list of ExpenseTypeSummary objects, which are defined below. So basically, these classes act as a view model to help display the summaries on the view or the screen. Each ExpenseSummary has a name and contains a list of expense type summaries, and those expense type summaries have an ID, a name, a count of the number of expenses that are of this type for the selected month on the Reports page, and a total for those expenses. So each of these lines is an ExpenseTypeSummary object. Let's scroll further up and see how this data gets populated. If the Run button has been clicked, that means there's a value in the Month and Year properties that are bound to the view, so we end up in the else block of code. There, the startDate and endDate of the month are built, and the _expenseRepository is called, passing in those dates along with the ID of the user. The results are stored in a variable that's available to all the methods on the page, and then the BuildSummaries method is called. Let's scroll down to see that. So first, a list of Expense Type Categories are retrieved from the repository, and these are the major headings like Expenses, Vehicle, and Home Office, the ones you saw on the screen. Then the code creates a PriceAdjustmentRulesEngine object, and we'll come back to this shortly. Next, we loop for each category and create a new ExpenseSummary, so that parent container. Assign the name, and then get all the expense types for that category. So, we're still just dealing with metadata here, not the actual expenses for the month. We loop through each of the expense types in the category and create an ExpenseTypeSummary. Remember, this is the line item that shows up on the report. We assign the ID, the Name, and then query the list of expenses for the month that was retrieved in the OnGet method. So these are the actual expenses for the month for this user. We loop through each expense in the list, add it to the count for the ExpenseTypeSummary, then pass the expense to the rulesEngine where any adjustments are made to the price. Then we add that price to the total for this expense type. Then the total for this expense type is added to the grand total for the month. This rulesEngine is defined farther up, so let's take a look at what this class does. This class has a list of objects that implement IAdjustmentRule. That's an interface that I created in code. In the constructor, it manually adds instances of classes that implement this interface, which you'll see shortly. There are ways to add these classes dynamically using reflection, but this is just a simple example. Then there's this method that gets called from the Reports page, and it takes the expense as input and loops through all the business rules in the list, checking to see if they're applicable. And if they are, the business rule is evaluated, and the price is returned, which might be adjusted or remain the same, depending on the evaluation of the business rule. Let's take a look at these rules. This DinnerAdjustmentRule implements the IAdjustmentRule interface, which means it has to have two methods, one called IsApplicable, and the other is Evaluate. In the IsApplicable method, it takes the expense and checks to see if the ExpenseType is Meals & Entertainment and that the Description is either dinner or supper. If both of those criteria are true, then the IsApplicable method returns true. The Evaluate method checks if the dinner expense price is above $15, which this company has set as the maximum amount allowed. If the price is above $15, then it's adjusted to the maximum, and 15 is returned. Otherwise, if it's under $15, then the actual amount is returned. The LunchAdjustmentRule is similar. It checks if the expense is a lunch expense, and if so, it verifies that it's not over $6. Obviously, this company is not very generous. So you can see that this business rule engine is a flexible way to add additional rules without having to add a bunch of if statements. It's a common design pattern, and I'll provide a reference at the end of the module where you can get more information on this pattern. But for now, the testers have informed us that there's a problem with the calculation. The Meals & Entertainment expense type isn't calculating properly. That's all we know at this point, so let's do some debugging in the next clip.

Conditional Breakpoints

Okay, so there's a problem with the business rules that adjust the total for the meals. You've seen how these totals are created, so let's look at how we can debug this using some advanced features of breakpoints. All of the calculations are taking place within loops. The innermost loop is for the expenses for the month. The next outer loop is for every expense type, regardless of whether there are actual expenses recorded for the report month. And the outermost loop is for the expense type category. Let's set a breakpoint here inside the middle loop, the one for each expense type. The app is still running, so I'll just run this report again. We hit the breakpoint. I'll just hit F10 to inspect the actual expenses for this expense type, and there aren't any. In the search of the expenses for the month, the search returned 0 expenses, and that's because this is the first expense type in the list. We can see the name of the expense type by hovering over it. Well, that makes sense. This breakpoint is going to get hit for every expense type, regardless of whether it's relevant. If I hit Continue, it moves to the next expense type in the foreach loop. But again, this one doesn't have any expenses. Of course, we can just keep hitting Continue and checking the expense type until we reach an expense type with some data, but there's a better way. Let's right-click on this breakpoint, and there are some options here. Besides deleting or disabling the breakpoint, you can set conditions. Conditional statements are basically if statements, and you can base these on expressions that you write here. So the breakpoint is only hit if the condition is satisfied. Hit Count causes the debugger to break here after this breakpoint has been hit a certain number of times. So maybe you've got a loop that's executing too many times. You can set the number here so you can inspect the data at a certain point. And filters break when the code is on a specific thread, process or machine, so this is good for debugging code that's running in parallel or multiple threads. We'll just look at conditional statements here though. You can select a break when the conditions you specify is true or when the variable that you specify has changed in value. Let's go with is true. And I want the debugger to break when the list of expenses that's retrieved above has a count that's greater than 0, so whenever we're in the loop of an expense type that has values for this particular month. I'll hit Enter, and now I'll close this and hit Continue at the top. If I hover over the collection that was retrieved, it shows that there are four records. So the loop continued running until the condition was met that the collection has expenses in it. And the first expense type with data is actually the Meals & Entertainment type, and we can inspect the data by hovering over the collection. Now we could step through the loop and try to debug where the problem is with each individual expense in the collection that was retrieved, but it would be much easier if we could look at all the expenses at once and try to figure out which one has the problem. Let's do that next.

Using TracePoints to Log Application State

Let's see how we can use breakpoints to log data so we can review it. I'll set a new breakpoint after the business rules engine has been called inside this loop. First, I have to advance the cursor with F10 though. Okay, now I'll right-click and choose Insert Tracepoint. You can see that the same dialog comes up, except instead of Conditions being checked, Actions is checked. You can actually use conditions and actions together, and I'll show you how shortly. Actions are tracepoints. What Actions allows you to do is to log a message in the Output window. Let's remove the breakpoint farther up because I don't actually want the code to break anymore. I just want to do some logging. Within this text box, you can enter strings and the values of variables by enclosing them in curly brackets. So let's log the ID of the expense that's being evaluated, then the Price field and the adjusted price, that's the return value from the call to the rules engine, calculate adjusted price method, then will print the name of the ExpenseType. I'll close this and click Continue. That brings us back to the running app. So let's go to Visual Studio again, and let's open the Output window. I'll pin this window, and let's scroll up a bit. There's output here from Entity Framework. It actually shows the SQL statements that are generated when Entity Framework calls the database, which is really helpful when you're trying to work out LINQ queries. But farther up here, there's the data that was output from the tracepoint. This is kind of noisy though, so let's remove some of this logging that's being done from .NET. I just want to see the code that's being output from the tracepoint. Some of this we can filter by right-clicking on the Output window and deselecting items here like the Thread Exit Messages. There's a lot of those. Now some other messages are being output using ASP.NET logging providers. We'll talk more about those in the next module and how you can leverage them to do your own logging, and that can really help debug errors that happen in production. But for now, know that you can set a log level for individual categories, including those that are generated by .NET, so this Microsoft.EntityFrameworkCore.Database.Command category. If I copy this and enter it in the appsettings.json under the Logging, LogLevel node, I can specify the level of message I want logged to the Output window. Let's actually turn off all messages by selecting None. I'll save this, and let's do a Hot Reload. And now down in the Output window, I'll right-click and select Clear All. Now we can run this report again, and a lot of the other messages are gone. There's the messages that were output from the tracepoint in the debugger and this other category, which we could filter in the appssettings.json file, but let's just leave it. Now this is still more data than we need. We only wanted to see the expenses related to Meals & Entertainment. So let's go back to the Reports page codebehind and to the tracepoint we created and right-click on it and select Conditions. You can see the action is still populated to log our message to the Output window. And now we want to add a filter so this action only runs when a certain condition is met. Let's copy the name of the expense type we want to filter on, and let's add this conditional expression that says this breakpoint only applies when the Name of the ExpenseType is equal to Meals & Entertainment. Now unlike before, the debugger won't actually break here, and that's because this check box is selected to Continue code execution. We could uncheck this, and it would behave like before, breaking every time the condition is met. But we just want to log the tracepoint data, so let's clear this Output window and run this again. Okay, now we've got data from our tracepoint, and it only applies to the Meals & Entertainment expenses. Great! Now let's look at these business rules again, the ones that apply to meals. So these rules check the ExpenseType and then do a string comparison with the Description field of the expense. We're not printing that though, so let's modify the tracepoint on the Reports page and add the Description field. Lets clear this and run the report again. Now if we look at the data, we can see that the two dinner expenses were above $15, and they were both adjusted accordingly. Let's look at the LunchAdjustmentRule. This one adjusts the price if the expense is over $6. That happened for this expense, but not for the first one. Okay, let's look closer at this particular expense. Since we've got the actual ID of the database record, we can do a conditional breakpoint based on that. If you're following along with the downloaded code in your environment, make a note of the record ID with your data because it'll be different than the ID in my database. We'll use that expense ID in the next clip. Let's do that next.

Dependent Breakpoints

Now we know which record has the problem, so let's inspect it a little closer. I'll go to the Reports page, and let's remove the tracepoint, and let's insert a conditional breakpoint inside the loop for the expenses retrieved for this month for the expense type. I'll set the condition so the expense ID equals 191, which is the primary key of the expense we identified from the tracepoint. Let's close this, and let's open up the LunchAdjustmentRule because this is the one that applies to the problem. If I set a breakpoint here, this breakpoint will get hit for every expense unless I set a conditional breakpoint for that expense ID, of course. But I want to show you another advanced feature of breakpoints. You can have the code break here based on another breakpoint being hit. So we have this conditional breakpoint on the Reports page that only gets hit when the expense ID is equal to the expense we want to investigate. So if I only want to break inside the business rule after that breakpoint has been hit, I can right-click on the margin and select Insert Dependent Breakpoint, then I can select from the list of breakpoints in the application. I only have one here, but you can see the fully qualified path to the class, method, and even the line number. Now this breakpoint should only get hit after the other conditional breakpoint has caused the code to break. Let's go back to the interface and run this report. Okay, so the conditional breakpoint is hit. I'll click Continue, and now the breakpoint in the BusinessRule class is hit. If I advance, you can see it's not returning true. Let's expand this Expense object and make sure we have the right one. Okay, its ID is 191. It's a meals expense, and its description says lunch, but lunch is lowercase here. And in the string comparison, it's looking for Lunch with a capital L. Let's copy this code and open up the Immediate window. Remember, this lets us run code and see the results. I'll paste this in and run it. It's still operating on the same expense, remember, which is the ex variable, and it returns false. But what if it was checking for a lowercase match? And that returns true. Okay, so that's the problem. This is an easy fix. We just need to convert the variable value to lowercase using the string method ToLower, and then the contains method needs to be updated to check for the lowercase version. So no matter if the data is capitalized in any way, it'll be switched to lowercase before comparison. Let's save this and do a Hot Reload. Now I'll click Continue, and lets run this report again. The first conditional breakpoint is hit. I'll hit Continue, and the dependent breakpoint is hit. And if I step through the code using F10, now the condition returns true. And back in the RulesEngine class, you can see the price is $8.25. And after the evaluate method is run, it's now $6, and that's how the business rule was designed. Let's go back to the Report page, and I'll right-click below the call to the business rules and select Run To Cursor. Now the cursor is stopped here. We can see the new price is $6. Lets continue, and the calculation has changed. If we want to verify that this is correct, we can go back to Visual Studio and set a tracepoint to show the data again. But first, lets delete all the breakpoints from the Debug menu. Now I'll right-click in the margin here and choose Insert Tracepoint. And I've copied some code here, just like the code you saw earlier to display variable values for the expenses. Before we run this, let's clear the Output window again just to keep things clean. Now I'll run this report. And if we look at the meal expenses, we can see that the ones over the limits were adjusted accordingly. So those are some advanced features of breakpoints. Logging from your application using tracepoints can be really helpful in getting a better overall view of the state of your application and data. It's limited to just the Output window though. So let's look at another way we can log messages from inside the application to various destinations like text files. We'll do that next.

Understanding System.Diagnostics.Debug and TraceListeners

Now lets talk about another way you can log information from your application during debugging. You can use classes in the System.Diagnostics namespace to add code to your application that only gets used while your app is compiled in debug mode. The System.Diagnostics.Debug class has methods to write output in a variety of ways. There are methods like Write and WriteLine, and they have overloads that can allow you to include categories in the output, which can help you filter through information. You can also write a debug statement only if a certain condition is met. This is better than putting the Debug.WriteLine statement inside a standard if statement because only the Debug.WriteLine statement will get removed when the application is compiled for the release configuration. It's just a way to keep your code clean. It's also possible to use placeholders in the strings that you output. And I'll show you in the demo how you can add padding to the fields to make the output easier to read. In terms of where these methods write to, you can define that by attaching TraceListeners to the static TraceListeners collection of the Trace class. The Debug class uses that collection. System.Diagnostics.Trace and System.Diagnostics.Debug are really intertwined. In fact, if you substitute Trace.WriteLine for Debug.WriteLine, not only will the output go to the same TraceListeners, but the trace statements will remain in the code that's compiled for release and deployed to production. Trace is really the original way for adding diagnostics to your application, but there are more advanced methods in .NET for adding diagnostics now. We'll talk a little about logging in the next module. There are built-in TraceListeners in the System.Diagnostics namespace that you can use, and they all inherit from a common base class, the TraceListener class. The default TraceListener writes to the output window. TextWriter TraceListener allows you to write output directly to a file. You can pass in a file stream or specify the file name. It's pretty simple though. It just writes messages without any additional information. TextWriterTraceListener is also inherited by derived classes, so you can write your log files in specific formats. ConsoleTraceListener writes messages to the standard stream, which is the console by default. DelimitedListTraceListener writes trace output in a delimited text form and uses the delimiter that you specify like a comma. This one doesn't actually work with the Debug.WriteLine messages though, only with certain messages of the Trace class. XmlWriterTraceListener lets you write structured data to an XML file. And there's another TraceListener that lets you write messages to the Windows Event Log. And if you don't find the functionality you want with the TraceListeners available in System.Diagnostics, you can write your own

TraceListener by inheriting from the base class. Now let's see how to use debug and TraceListeners in the demos.

Using System.Diagnostics.Debug

I've added some code to the sample application. This is also in the Downloads folder for the course. On the Reports page code-behind, I've added a using statement for System.Diagnostics, that's the namespace with the Debug class. I'll just collapse these regions, and let's take a look at the OnGet method. There's some code here in the code block where the report is run. First is Debug.WriteLine. This will output a message to all the trace listeners attached to the Debug class. By default, there's only one. That's the default trace listener that writes to the Output window. I've used the overload here that allows me to specify a category. The string for the category just gets prepended to the message. Then there's a line that uses Debug.Write, which just writes the message without a line break, so the line can continue with the Debug.WriteLine message. And then there's just a blank line that will help with formatting. Let's move down. Inside the loop where the expense price is adjusted using the business rules, there's a Debug.WriteLineIf. This allows you to only output a message if a certain condition is met. In this case, it's testing that the expensePrice was in fact adjusted. Now, why wouldn't I just have a standard If statement to test this and then put a Debug.WriteLine inside the code block for the If statement? Well, remember that when we compile this application with a release configuration, all the debug code will be removed. If the Debug.WriteLine was inside a standard If statement, that If statement would remain in the release code, it would just be empty. A minor thing, but this allows us to keep our code clean and efficient. Next, there's a Debug.WriteLine, and this one uses .NET composite formatting, which allows us to specify a string with indexed placeholders and an array of objects to replace those placeholders. Inside the curly brackets, we can also specify padding so we can create columns and make this easier to read in the output. WriteLine has several other overloads, as you can see. You've already seen the ability to enter a category. You can just pass an object like an exception, and WriteLine will use the ToString method of the object. And there are other combinations of those parameters. Let's run this app and see what the debug statements produce. When the app opens, let's go to the Reports page, and let's run the report for February. Remember, you'll have the same data in your version, but it may be a different month depending on when you're watching this course. Just go back two or three months from the current month to see this data. Now let's go back to Visual Studio and look at the Output window. This is where the default trace listeners in the Trace Listeners collection writes the debug messages. You can see there are a few entries for Gas expenses interspersed with output from the Entity Framework logging provider. Further up are more entries, and there are the entries from the OnGet method. Let's take a look at the console window. This opens by default when you run ASP.NET Razor Pages. It shows the Entity Framework code, but not any of our debug messages. Okay, let's stop the running application, and the first thing I want to do is clean up some of the noise in the output here. We don't need to see this Entity Framework logging, so let's open appsettings.json, and let's filter some of the messages out. I'll paste in this code that will filter out any messages starting with these category names. That should make it easier to see our debug messages. Next, let's add some trace listeners so we can see the output written to different destinations. We'll do that next.

Checkpoint 01 - Add a Debug Statement

Checkpoint 02 - Add a Category

Configuring TraceListeners

I've already added the code in the project for the trace listeners. We just need to open up the Program.cs file and uncomment the line that calls that code. This line calls a static method in a class that I've added here called CustomDebugConfig. You can put this anywhere. I just put it at the root of the project. There's a using statement at the top for System.Diagnostics. Then there's just the static method. First, it sets a string to the location where we want to write the text files. If the directory doesn't exist, this code will create it. Then the code creates some trace listeners. The first is the DefaultTraceListener. Because we're going to clear the collection of trace listeners, we could leave this one out, and there wouldn't be any messages written to the Output window. We could also configure this trace listener to write its output to a text file by setting the LogFileName property. Next, the code configures a ConsoleTraceListener. That will write messages to the console that's opened for ASP.NET. Next is a TextWriterTraceListener, and that will get created in the folder we specified above. After that, there's an XmlWriterTraceListener, which will get written to the same folder. And finally, we configure an EventLogTraceListener. And in the constructor, we're passing in the name of an event source. That's just a way that we can find our messages in the default application event log. I have to use the name of an existing event source because the ASP.NET process doesn't have permissions to create a new one by default. There's a green squiggly line here that's warning us that this EventLogTraceListener will only work on Windows platforms, which makes sense because .NET 6 is cross-platform, and Linux doesn't have an event log. Next, the code removes all the existing trace listeners from the collection on the Trace object. Remember that the Debug class uses the TraceListeners on the Trace class. The Debug and Trace classes are very interconnected. Then we add each of the TraceListeners to the collection and finally set AutoFlush to true. This will write the contents of the buffer automatically without having to call Trace.Flush each time. That's mostly important for the XML trace listener. Let's run the app now. When the app opens, I'll go to Reports and run the report for February again. And let's run the month after that because I know there's data there. Okay, let's take a look at the file system. I've just hit F5 to refresh Windows Explorer. This diagnosticsfiles folder is the one that the code created. Inside it, there are two files, the output of the TextWriterTraceListener and the output of the XmlWriterTraceListener. Let's open the txt file. So there's the message that was written in the OnGet method, including the Write and WriteLine messages that were concatenated. Then the expense entries are formatted into columns because of the padding that I used. Where the expense price is different from the adjusted price, there's a line that precedes the expense to indicate that. And back in the code on the Reports page, we can see the Debug.WriteLineIf statement that wrote that message. Okay, what about the XML file? Let's try to open it. It says the file is being used by another process, which is the ASP.NET process, so we'll wait until we stop the app. The Output window has the same messages, and remember, that's because we added the DefaultTraceListener to the collection. Let's take a look at the console window. The messages are written here too because we added the ConsoleTraceListener to the collection. How about the EventLogTraceListener output? I'll search for the Event Viewer on this Windows VM, and let's open it up. By default, the trace listener writes to the application log. This column is the event Source, which is basically a category to help us filter messages. We could create our own event source for the application and then reference it in code. But remember, I used the .NET Runtime as the event source because it already existed. Back in the Event Viewer, the messages were written as information messages. And each Write and WriteLine statement got its own entry in the event log. So, depending on which trace listener you plan to use, you might organize your debug messages more efficiently. Okay, let's go back to Visual Studio and stop the application, and we actually have to kill the process by closing the console window in order to release the lock on the XML file. Now I can open the XML file in the folder on the file system. By default, it opens in Visual Studio. Let's format this by highlighting all the code with Ctrl+A and then formatting it with Ctrl+K, followed by Ctrl+F. Now we have an XML trace event node for each Write and WriteLine statement in the code. The formatting doesn't make much sense for XML output, but just like with the event log, you probably won't be writing to multiple trace listeners when debugging. You'll pick one or two and format the messages in a way that makes sense for those trace listeners. Before we finish here, I just want to show you how debug messages are left out of the code when you compile with a release configuration. Let's close this file, and lets change the configuration to Release, and I'll delete these two files. These shouldn't get written again because even though the code will configure trace listeners, the Debug, Write, WriteLine, and WriteLineIf statements won't be part of the release. Let's run the report again. We'll do multiple months again this time so its the same test. And as expected, no files were created. And in Visual Studio, there were no debug messages written to the Output window or to the console. So System.Diagnostics.Debug gives you a way to include code to make debugging easier, and that code will get removed automatically from the Release version of the app. In this module, we looked at advanced features of breakpoints, including conditional breakpoints and tracepoints. And you saw how to use the Debug class to output tracing messages to different destinations. If you'd like to know more about the business rules design pattern I used in this module, you can check out this course by Steve Smith in the Pluralsight library. In the next module, we'll look at some more code features to help with debugging and also how to use logging providers to log diagnostic messages in your deployed code, which can give you information to help track down bugs.

Checkpoint 03 - Use a ConsoleTraceListener

Coding for Debugging

Controlling the Debugger in Code

In the previous module, we looked at how to use the Debug class in System.Diagnostics in order to log information to different destinations while debugging. System.Diagnostics has a class called Debugger. Debugger is a class that lets you control the actual debugger in Visual Studio or VS Code. Let's look at some of the properties and methods. Debugger.Launch lets you add code to have Visual Studio open a debug dialog so you can debug the code. Now why would you want to do this? This is useful in a situation where you've got an application that's running outside of Visual Studio, maybe a DLL that's been loaded by another process. Perhaps it's an Office app like Excel that's loading a module. Visual Studio has a feature called Attach to Process, but for whatever reason that may not always work. Instead of trying to attach to the code from the outside, you can have the code attached to your development environment using Debugger.Launch. I've also seen this mentioned in the documentation for products where you're developing extensions using C# like for ArcGIS Server. When you need to run the compiled code as part of another process, Debugger.Launch can give you a way to attach to your development environment, assuming that it's installed on the machine where the code is running. When you're already attached during a debugging session, you can have the code break by calling Debugger.Break. So you don't have to set a break point in the development environment, the code itself causes the debugger to break. If you just add this code as is, it will act similar to Debugger.Launch if a debugger isn't already attached. You might not want that though. So the Debugger class has a property that lets you check if a debugger is already attached, which as you can probably guess is Debugger.IsAttached. Then you can only force a break during your debugging sessions. Let's see these in action. I have the console application open from earlier in the course. Let's see how we can add code that forces the debugger to break. Let's start by adding a using statement for System.Diagnostics. Then down in one of the menu option choices the user makes, let's add this code that checks if a debugger is already attached so the code can query the environment to see if you actually debugging. Then I'll add this Debugger.Break method. This will cause the code to break here if there's a debugger attached. If there isn't, for example the application is deployed, then this condition won't be met. If we didn't have this if statement, then the user would get a pop-up asking to attach a debugger. But I'll show you a better way to do that, if that's actually the intention. But first, let's run this. I'll run the first option that doesn't have the code we just added. And now let's choose the option with the Debugger.Break code. The code breaks at the line we specified, and this works in VS Code also, by the way. I can hit F10 and continue through the code from the Debugger.Break line.

Now lets change this code and call Debugger.Launch. There's no point in this if we're already debugging though, so let's compile and publish the application to a folder on this computer and then run the exe outside of Visual Studio. We're going to publish this in the debug configuration so the PDB symbol files are published to the same folder. I'll go to the project file in Solution Explorer, right-click, and choose Publish. Choose Folder, Next, and Folder again. Now let's navigate to that folder on the desktop and create the published profile. A published profile is a way to reuse these settings and the destination. Now to compile and send these files to the folder, I'll click Publish. The files are here now, so lets double-click the exe. When the console opens, I'll select the option to list the expenses. That works, so let's select the option that has the Debugger.Launch code, and I got a pop-up that says Choose Just-In-Time Debugger. I can open a new instance of Visual Studio, or since I have an instance open where the code is already open, I can select that. My open instance of Visual Studio gets focus, and it says Source Not Available. But if I just hit F10, it brings me right to the line of code where the debugger was launched, and I can continue debugging from there. So those are some of the ways you can control the debugger right from within code. You probably won't use them often, maybe never. But in a situation where you might need them, now you know they're available. Let's continue by looking at another way you can pause code execution only when a certain condition is met by using Debug.Assert. Then we'll look at some attributes you can add to your code to also influence the debugger. First, the DebuggerHidden attribute lets you step over code, and then the DebuggerDisplay attribute to control how data appears when debugging. Then we'll talk about logging and how you can add logging to your application to gather data on problems when the code is released to other environments.

Using Debug.Assert to Clarify Intention

In the previous module, you saw how to add conditional breakpoints so your code would break when an expression was true like a variable having a certain value or a hit count exceeding a threshold. Breakpoints are great, but they're related to your debugging session. What if you want to include certain assumptions in your code that the state of the application meets certain expectations so that you get notified when that expectation isn't met? That's what Debug.Assert allows you to do. You typically add a Debug.Assert line at the entry point to a method to verify that the incoming parameters are within expected values, then you know that the code is being called properly. When used like this, it's particularly useful when working on a team and your code is being called by other developers' code. This way, your expectations about how the code should be used are baked right into the code. When your app is running in debug mode and the condition in Debug.Assert fails, the code breaks at the Assert line, so the developer can check local variables. You can also specify a message to output, and that message gets sent to all the TraceListeners that the Debug class is configured to use. Remember, that's the Output window by default unless you add other TraceListeners. And just like with other methods of the Debug class, Debug.Assert statements will get removed when the app is compiled for release, so you don't have to worry about unexpected behavior in your production apps. During development, assertions reflect the assumptions of the developer who wrote the code. And even when they're not triggered during execution, they provide built-in documentation for anyone reading the code, except this documentation is actually enforced. Before I do a demo, I just want to show you that Debug.Assert is used extensively in .NET itself, so this is an approach that Microsoft uses with their own code. You can see the source code for .NET on GitHub. You can browse through the various libraries yourself, or there's also a site called source.dot.net that allows you to browse and search the code. For example, let's search for StringBuilder. From the list that's returned, we can see its in the System.Private.CoreLib library. I'll select the class and the cs file. Now if I hit Ctrl+F to open a search and search for Debug.Assert, there are quite a few instances in this file, as shown by the lines on the right. Debug.Assert is used at the start of this method, and this method is called by several other methods in the file, by the way, so it's reused throughout this class. We can scroll down and see many other instances of Debug.Assert. It's often used at the end of a method also to verify that any calculations are valid before returning. So the point is if Debug.Assert is something that Microsoft thinks is useful to include in its own source code that's maintained by a team of developers, then you might want to consider using it too. Let's see how we can do that in the sample web application, next.

Demo: Debug.Assert

Let's see how we can use Debug.Assert in the web application. This is the same code from the previous module. I'm inside the ExpenseRepository. This is the class that calls the database to retrieve expense information. I've already added a using statement for System.Diagnostics. Let's scroll down to the method that's called for the Report page, GetMonthlyExpenses. Let's insert a line at the start of the method and call Debug.Assert. There are several overloads. We can just include a condition to evaluate, or this overload was added in .NET 6 to improve performance. It only evaluates any interpolated string formatting items if the message is required. String interpolation is when you prepend to the string with a dollar sign and include any variables and expressions in curly brackets right in the string itself. They get evaluated, and the string is produced. This overload, just make sure that doesn't happen when the assert is passing, only when it fails. And there are overloads that allow you to include a short message and optionally also a longer message. Remember, these messages get output to the trace listeners that are attached to the Trace class, and that collection is also used by the Debug class. Unless you specify trace listeners, only the output window is used by the default trace listener. Let's add a condition that userId != 0. This is a little confusing because you're not entering a condition here that should be true in order to cause the debugger to break and output this message. The condition here is your assumption of what is expected. In other words, it's what should happen in order to successfully continue. If this condition is not met, that's when the debugger will break. Let's go over to the code that's going to call this method. That's in the Report page code-behind. At the top, I have the userId hard coded. Remember, this is just during development because we haven't implemented security and logins yet. Down in the OnGet method, the _expenseRepository method gets called, passing in the hard-coded userId. Let's run the app. When it opens, I'll go to the Reports page and run the report for the current month. Even though there's no expenses returned, the repository code still got called. And as expected, the Debug.Assert condition passed. Let's close this to stop debugging and go up to the top of the Report page, and let's change this userId value to 0. That should cause the assert condition to fail because remember, it's expecting that the userId is a non-zero value. Let's run the same report. And now the debugger breaks at the assert line. If I hover over the userId, the value is 0. I'll hit F10 to continue to the next line. That will cause the message to be sent to the trace listeners. If I open up the Output window and scroll up a bit, there's the message. It says DEBUG ASSERTION FAILED and the short message we added to the method. So that's how you can add assert statements to your code to clarify and enforce the logic and intention of the code. Next, let's look at some attributes you can add in code that affect the debugger.

Checkpoint 04 - Add Debug.Assert

DebuggerHidden Attribute for Navigation

Let's start looking at some attributes that control the way the debugger behaves while you're debugging. Let's say you've been debugging a section of code that makes a lot of calls to other methods and you want to navigate into and out of those methods to trace the path through the code. You already know from earlier in the course that you can use F10 to move through your code without debugging into those method calls, and you can use F11 to debug into those method calls. When you're moving fast though, you might want a simpler approach to just keep hitting F11, so let's try that and see what happens. I have a breakpoint set on the codebehind for the expense page. When the page loads, the breakpoint is hit. Let's hit F11 and F11 again to continue into this method call to the ExpenseRepository. It says Nullable.cs not found. It looks like it's trying to step into .NET code, code that's inside the framework. Let's close this and continue. Same thing again. This is kind of a pain. Let's skip ahead to farther down in the code. There's a call into the repository again here, so the GetExpenseTypes method. I'll hit F11. Okay, good. We made it in. I want to keep hitting F11 to move through here though, and the debugger tried to step into another framework method for the Entity Framework source code. Okay, so when we're debugging, we just want to step through our own code usually. So let's look at how we can configure that in Visual Studio. I'll stop the running application, and let's go up to the Tools menu and open up Options. Under Debugging, General, there's a checkbox here that says Enable Just My Code. Let's check that. Now when you use F11 to navigate through your code, the debugger won't try to step into framework methods, but we can control things even more. What if you're stepping through your code using F11 to step into all the methods being called and there's a particular method that you don't want to step into? You know there's no problem with this method, and you want to just step over it. Of course, you could hit F10 instead. But if you're like me, you get into a rhythm when debugging, and you don't want to have to think about which key to press while you're focused on the code. There's an attribute you can add to particular methods in the System.Diagnostics namespace. The DebuggerHidden attribute will cause the debugger to step over this method, even though you're pressing F11 to step into it. There are actually a number of attributes here, and we'll look at another important one in the next clip. But let's add this, and lets run the code again to test this out. When the app opens, let's go to the expenses page, which causes the breakpoint to be hit. Now I'll continue hitting F11. And this time, I'm brought into the method in the Repository class, just like I wanted. No calls to the .NET Framework code. I can keep hitting F11, and the debugger doesn't try to open up Entity Framework code either. I'll keep stepping through until we get back to the calling method on the expense page. We don't want to debug through this loop, so I'll Run To Cursor farther down. Now we'll hit F11 again, and were brought into the repository. Let's continue on. And now this GetMinExpenseDate method is the one that I added the DebuggerHidden attribute to, so F11 shouldn't move the cursor into this method, and it doesn't. But the next method doesn't have the attribute, so F11 brings us right into that. DebuggerHidden is just one of those convenience things that you might never use. But in the rare case where you need it, it's good to know it's available. Next, let's look at an attribute that can help you format data in the debugger.

DebuggerDisplay Attribute for Formatting

In an earlier module, you saw how you can hover over a variable while debugging and expand the members when it's a collection. The debugger just calls the toString method on the object. So unless you've overridden that, it just returns to the type name. You could implement your own toString method for your custom classes, but that might have side effects if you're using that method in the interface, for example. You saw earlier how you can pin properties in the object, and that causes those properties to display at the top level. And this also translates to the Watch window, and the Autos, and Locals windows, too. But each developer needs to do this in their own instance of Visual Studio, and Visual Studio Code doesn't have this capability of pinning properties. So you might want to have your object display in a certain way and make that available for everyone on the team. The System.Diagnostics namespace has an attribute that you can add to classes and fields in classes. This will display the values of any expression you enter within the quotes. I'll just paste in some code here to show properties from the class. So anything inside the curly brackets gets evaluated like the ID property, and then I'm calling the ToString method on the DateIncurred property, passing in a format specifier. And of course, you can drill into child members like ExpenseType. Let's run this and see how it displays. Let's go back to the Expense page, and that loads the collection from the Repository class. I'll hover over, and let's expand the collection. Notice how long it's taking. So we have the object values displaying as expected, but the debugger had to call each property on each instance before it opened, which is a pretty costly operation. Imagine if there were hundreds of records. Of course, if you're only showing one or two properties and your dataset isn't that big, then this is fine. But there is a way you can make this more efficient, especially when there's a lot of work to do to surface this view. Let's close this and go back to the Expense class. Let's add a method, and this will only be for displaying a string in the debugger, so it's marked as private. Inside, there's just a getter, and this uses placeholders to retrieve the property values. And you can use format specifiers here too, just like with any other string. The advantage here is that this method, the code, is putting the string together, not the debugger. Up at the top in the DebuggerDisplay attribute, instead of having the individual properties, you can just call the name of the private method you added. Let's also add this nq suffix, which just asks the expression evaluator to remove any quotes when displaying the final value. Nq means no quotes. Let's run this again, and go to the Expense page. Now when I hover over the Expenses collection, it opens much faster, and it looks the same. The string here was created inside the class and just surfaced by the debugger as a single string. And as you'd expect, this view of the object is the same in the Watch window, as well as the Autos and Locals windows. Now let's see this in Visual Studio Code. Remember, VS Code doesn't have the ability to pin properties like Visual Studio does. I have VS Code pointed to the same code base, so let's run the app from VS Code here. When it opens, I'll go to the Expense page, and I have a breakpoint set in the same place. So let's open the debugger, and I'll hover over the Expenses variable. The formatted string is getting shown, although it's a little truncated. But over in the Watch window, we can expand this and see the summary string that shows the important properties just like we designed it. So that's how to use DebuggerDisplay to control the way objects are displayed in the debugger. Next, let's talk about adding logging to your apps to help with debugging problems in deployed applications.

Adding Logging Code to Help with Debugging

Logging is a big part of app development. You should always include some logging in your apps to help you diagnose problems once those apps are deployed and being used. There are entire courses in the Pluralsight library on logging, so I won't go into too much detail here. But I think it's important to discuss, especially now that you have an idea of the type of information your application can emit that can help you with debugging if you're able to capture it from the application after it's been deployed. Earlier, we talked about writing messages to TraceListeners using Debug.WriteLine. Those are removed from the release compilation of the app though. If you replaced all those Debug.WriteLine statements with Trace.WriteLine, they would end up in the released code and could write to the same TraceListeners. The Trace class is one of the first methods of logging introduced in .NET, but this isn't how logging is typically implemented these days. .NET has a framework called ILogger, which is intended to abstract logging implementation. It's used by default in ASP.NET web applications, and you can also add it to any kind of app. In ASP.NET, it's configured for you at startup, and you can include a logger in the constructor of any class through dependency injection. I'll show you how simple it is in the upcoming demo. Then you write log statements throughout your code, and the log messages get sent to logging providers, which are similar to the TraceListeners you learned about earlier. .NET has built-in logging providers that are easy to configure. In fact, most of the configuration is done in the appsettings.json file. If you've gone through the documentation for TraceListeners, you saw configuration examples in XML. That's not possible since .NET Core was introduced because we now use JSON files. That's why the configuration examples that you saw earlier used code. But ILogger is fully configurable through the JSON config file, at least for the built-in logging providers. There's a logging provider that writes to the console and a logging provider that writes messages to the Output window in Visual Studio. So far, logging providers just sound like TraceListeners, right? Well, they have some additional functionality, actually. The console logging provider has support for custom formatting like color formatting of messages and JSON output. There's also a logging provider for the EventLog on Windows machines, but then there's also a logging provider for the EventSource framework, which is a low level, cross-platform logging system that allows you to use tools to connect to your app and listen for log information. EventSource is used extensively by the framework code, so its a way to integrate with deeper monitoring. But ILogger is a framework. It's an interface where other logging providers can be plugged in as destinations. So Microsoft has logging providers that are Azure-specific. You can configure logging for web apps deployed to Azure App Service, which can write their logs to the file system in Azure where they can be accessed or even streamed in real time, or the logs can be written to Azure Blob Storage, which is a file storage service in Azure. And there's also a logging provider for Azure Application Insights. Application Insights is a service that lets you monitor applications from a central location in Azure. And you can collect information from inside your application in addition to the logging data and also things like visitor usage patterns, which are actually collected from outside the code. Application Insights is a big topic way beyond this course, but it's just an example of how ILogger is a flexible way to log that kind of future-proofs your application. There are lots of third-party frameworks for logging too. Some of them are complete replacements for ILogger, but a lot of them actually integrate with the ILogger framework so you can continue writing messages in the same way inside your code. But by configuring a third-party logging provider, you can log to other destinations. And being that ILogger is a framework, you can also implement the interfaces yourself and write your own logging provider. So next, let's see a demo of how you can configure logging in a web app and how you can filter the level of logging that you want to do.

Using iLogger in .NET Applications

Now let's do a demo with ILogger. We're going to add log statements to the web app to log to the console, as well as to the Windows Event Log. We'll use the application event log that's on every Windows machine. But in order to be able to filter the messages from all the others that are generated by Windows, we'll add an event source, specifically for the web app. Then we'll configure the logging provider in ASP.NET for the event log to use that event source. You'll see code to log at various levels like traces, information, and errors. By having messages right at specific levels, you can configure logging in appssettings.json to only log for the level that you want, for example, to only have warnings and errors and not information messages sent to the logs. Finally, you'll see the output in the event log and the console. I have the Windows Event Log open. This is what Windows writes to. And when the events come from .NET, the event Source shows us .NET Runtime. You saw in the TraceListeners demo that we wrote to the same event source. This time, I want to create a new event source, specifically for the web application. That'll make it obvious here which messages belong to our app. You can create event logs and event sources for existing event logs using PowerShell. Here, I'm using New-EventLog with the existing Application log and adding this event source called SampleApp. Now in Visual Studio, I have the web app open. In Program.cs, this is where the configuration of logging providers takes place. If you just leave the default code from the template, you get some logging providers configured for you, or you can do what I've done here, which is clear the logging providers and add them back explicitly. I've added the console logging provider and then the event log logging provider, and I've added some configuration. So when messages are written to the event log, they'll use the event SourceName we just added. It's important to know that the event source has to exist already on this machine. The process the web app is running under won't have permission to create the event source. Then on the Report page, because we're using ASP.NET, dependency injection is available out of the box, and we can just inject an ILogger instance in the constructor. That gives us access to the logging service that was configured in Program.cs. Notice I'm using the name of the class here in the interface. What that does is add a category name to all the messages that are written using this instance of ILogger. You'll see that later. We store this logger in a private class-level variable. That way, we can use it from anywhere in this class. Down here, I have a trace-level message. So, when I want to enable messages to be logged at the trace level, this will get written to the logs. There are various levels of messages you can write, and you can decide later in the configuration which levels actually get written. This LogTrace method is just a shortcut for the Log method with a log level already specified. You can see there are levels for Information, Warning, Error, Debug, Critical, and Trace. And there are methods provided that add the level for you automatically like LogInformation and LogTrace. Farther down, I've added a LogInformation message, and this will output the start and end date that are calculated. Notice I'm using placeholders here. Depending on which logging provider you use, these placeholders can get stored as individual fields in the destination, which can make searching and sorting much easier. In most logging providers though, this will just get logged as a single string. I've also added a LogError message, and this accepts the exception itself. So you can get all the information about the stack trace, exception type, and inner exception sent to the log. Farther down, I'm logging an information message for each expense that had its price changed as the result of a business rule. Now let's look at configuration in the appssettings.json file. You can have general settings that apply to all log providers, or you can override for each specific log provider. So I want to log all messages to the console at the trace level and higher. So that includes information, warning, and error messages. And I can break that down further for specific categories like the internal messages from the framework. For the event log, I've set the Default level as Error, so anything lower than an error won't get logged like warnings, information, debug, and trace messages, but we can override that for specific categories. So for ASP.NET Core messages, I want to include warnings. And for messages coming from the Report class, remember we configured the logger with the category name, which was the class name. So for those ones, I want to see the information messages in the event log. We can change this any time after the app is deployed. You might only log errors by default. And then when there's a problem, you can turn on more logging by changing the log level for a specific provider so you can get more information. Of course, you need the log statements in your code in order to get those messages. Let's run the app. And I'll go to the Reports page and run a report. Now let's open the console, which is where one of the logging providers is writing to. So here are some trace messages, User 123 has entered method OnGet, then the information message about the StartDate, then a trace message that the code is in the

BuildSummaries method, and then information messages about expense prices that were changed. So the trace and information-level messages were written here. Let's look at the event log. I'll refresh this. And the first thing you'll notice is there are messages written with the event Source as SampleApp. That's the custom event source for the web app. You can see the messages in the output at the bottom. These are the information-level messages because we didn't configure the event log to write trace messages. Because we used a custom event source, we can also filter this log. I'll type in the name of the event source and click OK. And now were just seeing log messages related to the web app. There's another column here called Event ID, and there are overloads on ILogger that let you write your own event IDs, so you could filter this even further. ILogger is a flexible framework for logging messages in deployed applications. You'll probably want to log to a central source like Application Insights or a third-party logging destination, but you can see that configuration is pretty straightforward. Next, let's do a wrap-up of the course.

Exercise 01 - Configure a Logging Provider

Course Summary

Well, you've made it to the end. We covered a lot, so let's have a quick recap. First, you learned some basics of using a debugger, specifically in Visual Studio and Visual Studio Code. Then you saw how to use debugging features by debugging an exception and a functional defect in a real-world application. You learned some of the benefits of becoming a better debugger like learning the types of mistakes that you make and how you solve problems. Then you learned about some of the typical traps in debugging and also an effective approach you can follow when debugging based on the scientific method. After that, you learned more debugging features of Visual Studio like debugging data using the Autos, Locals, Watch, and Immediate windows. Then you saw advanced breakpoints like conditional breakpoints and tracepoints. Next, you learned about features in System.Diagnostics that can help you with debugging like writing output to TraceListeners and inserting assert statements to verify inputs to methods. And finally, in this module, you learned more about logging and how it can be used to gather information from deployed applications. Thanks for watching the course, everybody. My name is Neil Morrissey, and I hope you'll join me for future courses, here on Pluralsight.