Course Overview

Course Overview

[Autogenerated] Hi, everyone. My name is Jason Roberts and welcome to my course error handling in C#. I'm a freelance developer and author. If you want to create readable and understandable code that responds correctly to errors at runtime, then understanding and making use of C# exceptions is crucial in this course, we're going to learn how to handle errors in your C# code in a typesafe readable and structured way. Some of the major topics that we will cover include how to respond to runtime errors in your application, how to indicate an errors in your own code, how errors are represented by exception hierarchies, how to re throw and wrap exceptions and how to write unit tests to check exception throwing code. By the end of this course, you'll have the skills and knowledge of C# exception handling needed to manage runtime errors in your C# applications. Before beginning the course, you should be familiar with basic C# syntax. I hope you'll join me on this journey to learn about C# exceptions with the error handling in CSHP course at PL site.

Understanding the Importance of Error Handling

Introduction

[Autogenerated] Hi, everyone. I'm Jason Roberts from applause. Site errors in production systems can cause a lot of stress not only to end users but also to software developers, operations, people and managers, errors can occur for multiple reasons such as bugs in the code, network problems or even catastrophic hardware failures. Exception handling in C# allows runtime errors to be managed in a structured way and can help improve stability and manageability of production applications and systems. This course was created using CS Sharp 10 dot Nets six and the demos in this course were recorded using Visual Studio 2022. But you could also follow along in Visual Studio code. If that's your

preference. This course is 100% applicable to these versions. In this first module. We'll get an understanding of the importance of error handling and we'll kick off this module by answering the question, why handle errors in the first place? We'll then learn about one way errors were handled in the past using error codes and we'll learn why. This is not a good idea in C# applications rather than error codes. In C# we make use of exceptions to handle error situations. So we're going to learn why we use exceptions and we'll finish off this module by getting an overview of what exceptions are.

Course Overview

[Autogenerated] This course is organized into a number of logical modules. We're currently in this first module here understanding the importance of error handling in the following module getting started with exceptions. We'll learn how to create an exception and throw it. We'll learn about the stack trace and get an overview of how exceptions are handled in the next module. We'll learn how exceptions are organized into a class hierarchy and we'll learn a bit about exception constructors in the next module. We'll go into more detail about catching throwing and also reth throwing exceptions. And we'll also learn about more advanced catching using exception filters up until this point in the course, we will have been making use of exceptions defined for us. We can also define and use our own custom exceptions. So we have a module dedicated to this. And in the final module, we'll learn how we can write automated tests for code that throws exceptions. All of the demo code for this course can be downloaded from the course page at PL site.com and the demo code is organized into before and after folders. So without further ado let's learn why we want to handle errors in the first place.

Why Handle Errors?

[Autogenerated] Nobody likes to see dialogue boxes such as this telling us that our application has crashed and we may have lost all of our work. Making use of error handling in our program gives us a number of benefits. First off, depending on the type of error that occurred, we may be able to prevent the entire program from crashing. First and foremost, this can help prevent unhappy end users. Or if the program is some kind of back end process, prevent data loss or further cascade failures, depending on the type of error that we're handling, it may give us the opportunity to fix or retry the operation. So for example, if we have a transitory network error, we could detect this and then retry the

network operation. Once again, this can help reduce the number of messages we have to give to the end user or to the system administrators. Error handling can also help us present more meaningful messages to the end user. And if we can't fix the problem, we can perform a graceful exit of the program rather than it just crashing by gracefully exiting the program and not simply crashing. It also gives us the opportunity to try and save any work in progress. Handling errors also gives us the opportunity to log those errors. But apps to a log file or some logging service or api these log messages can prove highly valuable when trying to diagnose problems. As a side benefit, good error handling code also helps future maintainers of the code, understand what possible error conditions may occur and how they can be handled. So, error handling code also helps to produce self documenting code. Let's take a look now at an outdated form of error handling using error codes. And this will help us understand why using exceptions in our C# applications is the preferred approach.

Error Handling Using Error Codes

[Autogenerated] So suppose we have a method called process data. And this method just processes some data file as a return value. This process data method returns an inch value indicating whether the data file was processed successfully or not. If we want to call this method, we need to capture this error code. So here we're just capturing this in a variable called error code. And then we need to compare this error code with a set of known integer values. So in this case, the value zero could represent the fact that there was no error and the data file was processed successfully. We may have other error codes. In this case, the error code of one indicates that the data file contained invalid data. We may have another error code of two indicating that the data file was empty and so on. Implementing error handling using error codes poses a number of problems. First off, we may need to know all of the return values in this example, ins that represent errors. And we may also need to know all of the return values that represent success for each of the return values that represent errors. We need to remember to add an else if or switch statements and if we don't and one of these error codes occur that we haven't trapped program flow will continue as normal even though there are errors. And this may cause data loss or further damage, error handling with error codes may be harder to read than exception handling code. And these error codes are like magic numbers with no semantic meaning which may harm readability or understanding of the code base. We could of course replace these hard coded numeric values with constants. But this doesn't alleviate all of the problems. For

example, every time this method is called, we need to add if statements or switch statements to check the return codes, errors don't bubble up the call stack to higher level code. This means that we can't choose to catch certain errors at a higher level in the code base, which also means that we can't opt to catch some errors in just one single place in the code base. Every time the method is called, we need to add the if or switch statements. Another problem is how do we deal with system errors if we have no way of catching these system errors, how can we write error handling code to address them? So for example, how do we handle out of memory or access violation errors also because constructors don't return a value. How do we return an error if one occurs during object construction? It's for all of these reasons that in C# applications, we don't usually make use of error handling, using error codes, we instead use exceptions. So let's have a look next at why exceptions are a better choice for handling errors.

Why Exceptions?

[Autogenerated] When we're using exceptions to implement error handling. First off, we don't need to know all of the error or success codes that the method returns because we're not using return values as error codes. We also don't need to add if or switch statements every time the method is called, generally speaking, exception handling code is more readable with less clutter than error handling code that uses error codes. And we also don't need to have magic numbers or constants. Unlike error codes, exceptions can bubble up the core stack, which means we can catch exceptions higher up in the code base or in one place. And exceptions also allow us to respond to system errors. Unlike error codes, we can also indicate error conditions from constructors by using exceptions. So now we understand a bit more. Why exceptions are the preferred choice of error handling in C# let's get a high level overview of what exceptions are.

What Is an Exception?

[Autogenerated] Exceptions are simply put object instances. These object instances inherit ultimately from the system dot exception based class. We can generate our own exceptions by using the throw statement in C# and different exception classes represent different kinds of errors. Exception classes can contain additional error information in properties and different types of exceptions can be handled

with different error handling code. Exception definitions can come from various sources such as the standard exceptions provided to us by Microsoft in net itself, they can come from additional frameworks and libraries that we can download, for example, from NuGet and we can even create our own custom exceptions, which we'll be doing later in this course.

Summary

[Autogenerated] So that brings us to the end of this module. In this module, we started off by looking at why we actually want to handle errors in the first place. So we learn that we may be able to prevent the entire program crashing. The fact that error handling gives us the opportunity to fix or retry operations. And the fact that error handling means we can provide meaningful messages to the end user and also gracefully exit the program. We learned about an outdated mode of error handling using error codes. And we learn that we need to add if or switch statements to check for error codes and these error codes may be magic numbers with no semantic meaning. We learned why exceptions are the preferred error handling technique in C# such as improving the readability of code and also allowing exceptions to bubble up the call stack. Finally, we learned what an exception is and we learn that they're objects that inherit from the system dot exception based class. And the fact that they can be provided to us by .NET itself, they can come from additional libraries and we can create our own custom exceptions. Join me in the next module. When we'll be getting started with exceptions, we will learn how to create and throw exceptions. We'll learn about the stack trace and we'll get an introduction to catching thrown exceptions.

Getting Started with Exceptions

Introduction

[Autogenerated] Hi, welcome back in this module. We're going to start digging into exceptions in more detail. We're going to kick off this module by learning a bit more about exception handling such as the

concept of exception bubbling and also in this module, we'll get an introduction to the mechanism that allows us to handle errors. The try catch and finally blocks. We'll hop into Visual Studio and we'll get an overview of the demo code we'll be using in this course and then we'll go cause an exception to be thrown. We're going to learn about the stack trace and how the stack trace can help us locate where errors originated. We'll learn how we can create an exception instance and also how to throw it. We'll also get started with exception catching and finally, we'll wrap up this module with some exception handling, good practices. So let's kick off this module by learning about the concept of exception bubbling.

Understanding Exception Handling

[Autogenerated] Suppose in our code, we had method A and method A made a call into method B. In turn method B calls into method C if an error occurs in method C and an exception is generated and thrown. This exception will travel up to method B because it hasn't been handled in method C. If method B does not handle this exception, the exception will continue to move up or bubble up the call stack. If method A was the entry point to this program and method A did not handle this exception, the exception would essentially bubble up to the operating system and this program would crash. And for example, in Windows, this would cause a message to be written to Windows event log. We'll see an example of that later in this module. Let's take a look at a different scenario. Once again, we've got method A calling into method B and method B calling into method C. But this time method B calls into method C inside a try catch statement. Once again, an error occurs in method C and causes an exception to bubble up to the caller in this case, method B. But because method C called inside a try block. Method B can catch this exception or in other words, handle this exception. And if method B can in some way deal with the problem. For example, retrying the call to method C, it can prevent the exception from bubbling up the call stack to method A and ultimately potentially crashing the program. So in C# it's this tri statement that allows us to implement error handling. Let's have a look at this next.

Introducing the Try Statement

[Autogenerated] So in C# we make use of the try keyword to start implementing error handling inside this tri block, we have one or more operations that could potentially throw exceptions. So for example,

this could be to perform a calculation call into a database or access a file on the file system. After the try block, we can implement one or more catch blocks. In this example, we've got the catch keyword and then inside parentheses, we've got the type of exception that we want to catch. In this case, an argument, null exception. We can also specify a variable in the catch block in this case called ex and this captures the exception object and allows us to access properties of the throne exception. If we want to, we can add multiple catch blocks after the try block to catch different exception types and deal with them in different ways. For example, in this case, we're catching an invalid operation exception and we can continue to add additional catch blocks in this case, catching the exception type, which as we'll learn in the next module is the base class for all exceptions. One thing to bear in mind when you're implementing multiple catch blocks is that you must implement them in the most specific to least specific exception type order. And that's why the final catch block in this example catches the base exception type because this is the least specific exception type. So once again, we'll look at the exception hierarchy in more detail in the next module. There's also some alternative syntax for the catch block notice down here that we haven't specified the exception type. This means that this catch block will catch any other exceptions, regardless of the type of the exception. But notice in this example of the syntax, we're not capturing the exception object in a variable. So we can't access any of the properties of the throne exception. In addition to the catch block, when we're using a try block, we can also implement a single finally block. This finally block is always executed when control leaves the tri block. So regardless of whether or not you've got catch blocks or whether or not exceptions occurred in the tri block and were caught or not, the code inside the final block will always execute. It's inside this finally block that you should execute any cleanup code that should always be executed such as disposing of objects. If you need to, we can combine all of these blocks, for example, the try block followed by one or more catch blocks. And then at the end a finally block to execute cleanup code. So this was just a quick overview of the try catch and finally blocks. We'll be seeing these in action later in the course. Let's head over to Visual Studio now and we'll take a look at the demo code we'll be using for the rest of this course.

Demo: Code Overview

So here we are in Visual Studio, and we've got this ConsoleCalculator solution open. And if we have a look at this ConsoleCalculator project, we've got a couple of classes here. Let's start off by looking at

this Calculator class. We're making use of file-scoped namespaces here, so this class will fall inside the ConsoleCalculator namespace. This Calculator class has a single public method called Calculate, which returns an int and takes two ints and a string operation. Inside this method, we decide on the type of operation to perform and then call the relevant private method. Currently, we've only got this Divide operation defined, which calls into this divide method. If the provided string operation is not recognized, then we get this WriteLine to the console saying that we've got an unknown operation, and then we return a value of 0. This Divide method here, which is implemented using an expression body, simply performs the division and returns the result to the calculate method. This is a console application, so let's look at the main Program.cs file. If you're wondering where the main method is for this console application, we're making use of top level statements here, which essentially replace the code that goes inside the main method. This allows us to have less of that boilerplate code related to console applications. Also notice up here that we've got a using static System.Console, just so we can call the console WriteLine method without having to prefix every WriteLine with console dot. This console application asks the user to enter a first number, and then we read that into this variable number1. We then do the same thing for a second number, and then we ask the user to enter an operation. This is captured as a string. We don't have any additional input validation in this application here, just to keep things simple for demo purposes. Once we've got the first number, second number, and operation from the user, we create an instance of the Calculator and call its Calculate method, passing in the two numbers and the operation. We then call this DisplayResult method and pass in the result from the Calculate method and then ask the user to press Enter to exit the console application. If I just scroll down to this DisplayResult method, this is simply calling the console's WriteLine method and outputting the result using this interpolated string. Let's take a look at this in action. Just going to hit F5 to run the program. We'll enter a first number of 10, a second number of 2, and we'll choose divide as the operation and hit Enter, and as expected, we get the result 5. And we can hit Enter to exit. So next, we're going to run the application once again, but this time we're going to cause an exception.

Causing an Exception

So once again, we'll hit F5 to run this Console application with the Visual Studio debugger. Once again, we'll enter a number of 10 for the first number, but for the second number, we'll choose 0, and

then once again choose divide. This is going to cause a DivideByZeroException. Let's hit Enter, and we can see here that Visual Studio is telling us that we've got this DivideByZeroException being thrown. Let's stop debugging. And instead of running this in Visual Studio, we'll go and run this from the command line. What I'm going to do is right-click here, come down to Open Folder in File Explorer, and click that, and head into the bin, Debug folder, hold down Shift, and right-click, and choose Open PowerShell window here. We'll go and execute this ConsoleCalculator, this time outside of Visual Studio, so I'm just going to hit Enter and once again try and divide a number by 0. Once again, we can see we've got this DivideByZeroException thrown, but because we're running outside of the Visual Studio debugger and we don't have any error handling, this DivideByZeroException will have been bubbled up to the operating system. If we go and have a look at the Windows Event Viewer. To find this, just hold down the Windows key and hit X on the keyboard, and then come over here and choose Event Viewer. Come into the Windows Logs and the Application log, and we can see we've got this application error here. It's telling us here in the event information that the Faulting application name was ConsoleCalculator.exe. So this is an example of an unhandled exception, bubbling all the way up and crashing to the operating system, in this case, Windows. Let's have a look next at the stack trace.

Understanding the Stack Trace

What we'll do is, once again, run the application in Visual Studio by hitting F5 and once again try and divide by 0. Once again, Visual Studio breaks here and tells us we've got this DivideByZeroException. But if I just click View Details, what we've got here is the exception object details. Don't worry about all of these properties. We'll look at them in more detail in the next module. We'll just take a look at one of these properties for now, the StackTrace property. And if we just view this, and what I'll do to make this easier, I'll just copy these details and paste them into Notepad here so we can see them a little bit better. The stack trace is like a map that leads us to the point where the exception was thrown. If we work backwards here, we're starting off in the Main method of the Program class. We can see that the Main method, then called the Calculate method of the Calculator, and then the Calculate method of the Calculator called into the Divide method of the Calculator. So the stack trace is telling us that the exception occurred in the Divide method, and bubbled up to the Calculate method, and then bubbled up to the Main method. So this is an example of an exception being thrown by someone else's code,

in this case, .NET itself. We can also go and throw our own exception instances. Let's take a look at this, next.

Creating and Throwing an Exception

Let's go and make some changes to this Calculate method. At the minute, this Calculate method is returning a value of 0 when an unknown or unsupported operation is provided. Recall in the previous module that we talked about the concept of using error return codes or magic numbers to represent error situations. In this case, we're just returning the value 0 for an unknown operation, but the client has no way to know whether that return value is actually an error value or a legitimate result from an operation. So what we need to do is replace these two lines here with some code that throws an exception. So I'll start off by commenting out these lines. I'm just going to hit Ctrl+K and then Ctrl+C, and then we'll go and create an exception instance. In this case, we're going to create an ArgumentOutOfRangeException. There's a number of overloads to the constructor for this, one of which allows us to specify a parameter name and also a message. The parameter name constructor argument allows us to specify which argument had invalid data. In this case, it's the string operation parameter. At the minute, we only support the divide operator. The second parameter of this overload allows us to specify a message for this exception. So for example, we could set this exception message to be, The mathematical operator is not supported. At the moment, we're specifying the parameter name using this string, but if we go and refactor this Calculate method and rename the operation parameter, it's not going to automatically update this string here. So we're going to get invalid parameter names. A better way of doing this is to, instead of using a string literal, use the nameof operator, which will automatically convert the parameter name to a string. Now if we were to go and change the name of this operation variable, if we didn't change it here as well, we'd get a build error. Now we've created this ArgumentOutOfRangeException object. We can go and cause an exception to be thrown. To do this, we use the throw keyword, and then specify the exception object instance. In this case, this is in the variable, ex. You don't have to create this intermediary variable. Instead, we could simply create it and throw it in the same line of code. Let's take a look at this in action. We'll hit F5 to run the application in debug mode. We'll enter 10 and 2, but this time we'll enter an unsupported operation, in this case, addition. Just hit Enter, and notice here the Visual Studio debugger breaks. It's telling us that we've got this ArgumentOutOfRangeException. And if we have a

look at the details, we can see that we've got the parameter name being set to the operation parameter, and we've also got the message here, The mathematical operator is not supported. So that's how we can throw our own exceptions. Let's take a look next at the try catch block in action.

Getting Started with Exception Catching

What we're going to do is come back to the Program file, and we're going to come down here to where we call the Calculate method. What we want to do is surround all of this code in a try block. Now if any code inside this try block causes an exception to be thrown, we can catch it in one or more catch blocks. So let's add a catch block here. And for the type of exception that we want to catch, we'll specify the base class exception, and we'll also capture this exception object instance in this variable, ex. Now we can choose how to handle exceptions inside this catch block. All we're going to do in this example is to write a message out to the console window telling the user, Sorry, something went wrong, and we're also going to output some exception details. Let's hit F5 to run this, and once again, we'll enter an unsupported operation and hit Enter. Notice this time, we get our message from the catch block, Sorry, something went wrong, and the exception details such as our custom message here, The mathematical operator is not supported. So we got our catch block executing. This also means that because we handled the exception, if we were to run this at the command line, it would no longer crash to the operating system because the exception is not bubbling up and out of the program to Windows. This also means we won't get an error in the Windows Event Viewer. Let's just hit Enter to return to Visual Studio. In this case, because we're dealing with user input, including the operation that the user enters, we should add some validation code to the main method to validate the input. Because we can say that invalid input is part of the normal program flow, we could also write logic to try and prevent the Calculate method being called for invalid or unsupported input. We still could leave this exception throwing code here in case there was a bug in any validation code and the Calculate method still got called with an unsupported operation. Before we wrap up this module, let's take a look at some exception handling good practices.

Exception Handling Good Practices

[Autogenerated] So the first of the good practices is to not add a catch block that does nothing or just re throws the exception. Essentially what this means is that catch blocks should add some kind of value. This may be just to log the error to some log system or output the error to the end user. It's usually considered bad practice to swallow or trap exceptions. So this means we have a catch block that doesn't do anything and doesn't re throw the exception, it just swallows the exception like it never existed. The next good practice is to not use exceptions for normal program flow logic. So for example, if we're dealing with input validation, we actually expect some input to be invalid, invalid input is not an exceptional situation. So you wouldn't control program flow with thrown exceptions. In this case, because it's part of the normal expected logical flow instead, you might have methods such as is valid to check whether input items are indeed valid. Next up, it's good practice to try and design code to avoid exceptions in the first place. So for example, say we had a pass method that takes a string and converts it to an in if we pass some invalid string input, that can't be converted to an in this pass method could throw an exception. Alternatively, we could implement a try pass method which returns a Boolean indicating whether the pass was successful instead of an exception. And if the passing was successful, we get the result in an output parameter. Another example of this is checking the connection state property of a connection before trying to close it. If we called the close method on a connection that was already closed, we'd get an exception instead. This way, we only try and close the connection if it's not already closed. And so avoid the exception. You may also want to consider returning a null or null object pattern for extremely common errors instead of throwing an exception. The next good practice is to make sure we use correct grammar in exception messages. This means using correct punctuation, correct spelling and also ending sentences with a full stop. You may also want to consider localizing your error messages. Another good practice is to make use of final blocks for clean up. For example, this could mean calling dispose and by using finally blocks for clean up, we can ensure that callers of methods should be able to assume no unexpected side effects. If an exception is thrown and caught

Summary

[Autogenerated] So that brings us to the end of this module. In this module, we started to get a better understanding of exception handling. In C# we learn that when exceptions are thrown, if they're not caught, they bubble up the stack trace and to handle errors, we can make use of the try catch final

construct. After we got an overview of the demo code that we'll be using in this course, we executed the program and caused a divide by zero exception. We saw this being handled in the Visual Studio debugger and then we went and ran it at the command line and we saw that because the exception was unhanded and crashed to the operating system. It caused a Windows event log entry to be created. We learned about the stack trace and also the fact that exceptions have properties. We went in through our first exception and we got a first glimpse of the try catch blocks in action. Finally, we looked at some exception handling, good practices such as not using exceptions for normal logical program flow. Join me in the next module when we'll be digging a bit more into the exception class hierarchy and also some of the properties that exist on the exception based class.

Understanding the Exception Class Hierarchy

Introduction

[Autogenerated] Hi, welcome back in this module. We're going to be digging a little bit deeper and understanding more about the exception class hierarchy. So we're going to kick off this module by discussing what an exception represents. We'll learn that exceptions are organized into a class hierarchy and that the system dot exception class is the base class. For all exception types, we'll learn about some commonly used constructors to create exceptions and we'll also discuss some guidelines for the use or more appropriately, the non use of the application exception class. And finally, we'll learn about some commonly encountered exception types. So let's kick off this module by learning what an exception represents.

What Does an Exception Represent?

[Autogenerated] The Microsoft documentation defines an exception as any error condition or unexpected behavior that is encountered by an executing program. We can think of exceptions as being defined by the system by third parties or by our own code. For example, system exceptions are

thrown by the .NET runtime and are defined as part of .NET. For example, the out of memory exception or the stack overflow exception. Third party exceptions are provided by third party libraries or frameworks such as the J serialization exception provided by the Jason .NET library. We can also define exceptions in our own application code. For example, suppose we've created a rules engine, we could create a rules engine exception to represent errors. When executing the engine. Essentially the actual type of the exception class represents the kind of error that occurred. Any additional property values that are set on the exception object help to further define or refine the error that occurred.

The Exception Class Hierarchy

[Autogenerated] Exceptions are organized into a class hierarchy at the root of this hierarchy. We have the system dot exception base class. We're not going to be looking at all of the exception types on this diagram. It's just to give you an idea of how exceptions are organized. The system exception class inherits from the exception base class. And this has a number of derived Children including the out of memory exception, the stack overflow exception and the argument exception, the argument exception class itself is a base class for a number of other exceptions, including the argument, null exception and the argument out of range exception. The arithmetic exception class inherits from the exception based class. And this has further derived classes such as the divide by zero exception and the overflow exception, the application exception class inherits from the exception based class. And we'll be talking about this in just a moment and when we're creating custom exceptions, we also inherit from the exception based class, we'll learn how to do this later in the course. If you want to dig into more detail about the exception class hierarchy, check out the Microsoft documentation at this link here. Let's examine this exception based class in a little more detail. Now.

The System.Exception Base Class

[Autogenerated] So the system dot exception class is the base class for all types of exceptions. The class itself has a number of properties such as the message property, the stack trace property, the data property, the inner exception property, the source property, H result property, the help link property and also the target site property. Let's take a look at each of these. In turn. Now, the message

property is of type string and it describes the reason for the exception or error condition. If we're setting the message property, we should write it for the developer who's going to handle the exception. The message should completely describe the error condition and it shouldn't be cryptic and where possible or applicable. It should describe how to correct the error condition. One thing to note is that in some cases, the error message may be shown to the end user and it also may sometimes be logged to a log file or logging service. This means that we should use correct grammar and we also shouldn't include sensitive information such as passwords, personally identifiable information or other security data. Next up, we have the stack trace property. This is also of type string and provides information about the call stack. So one way to think about this is the trace of the method cause that led up to the exception being generated. So essentially the stack trace helps to show the execution path that led to the exception. The data property is of type I dictionary which gives us a collection of key value pairs. The key is of type string and the value is of type object, meaning we can store anything in it, we can store an arbitrary number of items in the dictionary. And these items can supply additional or supplementary user defined exception data. As with the message property, we shouldn't include any sensitive information in these keys or values. And one thing to bear in mind here is that we should be careful of key conflicts. For example, if you try and catch an exception that already has data in the eye dictionary, if you try and add a new piece of data with the same key, this will cause an error. Next up, we have the inner exception property and this is of type system dot exception. And this inner exception property allows us to capture the preceding exception in a new exception. This is often known as exception wrapping and we'll learn about this later in the course. The source property is of type string and it represents the application or the object name that caused the error by default. This string property will be set to the name of the originating assembly. The H result property is of type in 32 and represents the H result numerical value. This property is often used with com interrupt code. Next, we have the help link property which is of types string. This allows us to specify a link to an associated help file for this error. This link may be in the form of a uniform resource name or it may be a URL. Finally, we have the target site property. This is of type system dot reflection dot method base. And this gives us access to information about the method that through the current exception, such as the method name, the return type, whether the originating method was a public or private method along with a whole host of other reflection based information. So now we're starting to get an idea of the system dot exception based class. Let's take a look at some of its constructors.

Commonly Used System.Exception Constructors

[Autogenerated] There's a number of different constructors available on the exception class. And you'll often see variations of these in derived classes. Also, the simplest exception constructor doesn't have any parameters and we get a default message property assigned and a null in an exception property. The default message property that we get is different for each of the derived exceptions. The next constructor overload allows us to specify a string message. And this maps to the message property of the exception class. The third common constructor overload allows us to specify a message and also an inner exception. So it's this constructor that we use if we're wrapping an existing exception.

System.ApplicationException Guidelines

[Autogenerated] Recall from earlier in the module, we mentioned this application exception type. This quote from the framework design guidelines sums up this type system dot application exception is a class that should not be part of the .NET Framework. The original idea was that classes derived from the system exception would indicate exceptions thrown from the CLR or system itself. Whereas non CLR exceptions would be derived from application exception. However, a lot of exception classes didn't follow this pattern. So to summarize the application exception class, it should not be thrown by your code. It should also not be caught by your code unless you intend to re throw the original exception and custom exceptions should also not be derived from the application exception based class.

Commonly Encountered Exceptions

[Autogenerated] There's a number of exception types that it's useful to be aware of. First off, we have the exception and system exception types. The exception class represents execution errors that happen during the program's lifetime. And the system exception class is the base class for exceptions in the system exceptions, namespace. You should not throw either of these exceptions yourself and you should not catch these exceptions except in top level exception handling blocks. You should also not catch these exception types in framework code that you're developing unless you intend to re throw them next up. We have the invalid operation exception. This is thrown when the current state of the object is invalid. For a specific method being called, you can throw an invalid operation exception

yourself when your object is in an inappropriate state. For a specific method being called next up, we have the argument exception, the argument null exception and the argument out of range exception. The argument exception is thrown when a method argument is invalid. This argument exception class is also the base class for the argument null exception, which is thrown when a null is passed to a method argument. And it cannot accept nulls. And the argument out of range exception is also derived from exception. This exception is thrown when a method argument is outside of an allowable range. For example, if you had a divide method on a calculator class and someone tried to pass zero, you could throw an argument out of range exception stating that zero is not a valid value to divide by. You should prefer the most specific derived exception that meets your needs. So for example, you wouldn't throw an argument exception if you're trying to represent the fact that a null cannot be passed. When you're using the subclasses of the argument exception class, you should also ensure that the param name property is set. This property allows you to set the parameter name that caused the exception. Next, we have the null reference exception and the index out of range exception. The null reference exception is thrown when an attempt is made to dere reference a null object reference. For example, if you try to call the subs string method on a null string object, the index out of range exception is thrown when attempting to access an array or collection item, that's outside of its bounds. For example, trying to access the 11th item in a collection. When it only currently contains 10 items, these two exceptions are reserved for runtime use and usually when they're thrown, indicate a bug in the program code because they're reserved for runtime use. You should not throw these yourself to avoid these exceptions being thrown, you can check argument values when they're passed to a method. The stack overflow exception is thrown when too many nested method calls cause the execution stack to overflow. So for example, if you've written a recursive method, but you've forgotten to add a clause to actually exit the recursion. This exception is reserved and thrown by the runtime. So you shouldn't explicitly throw it yourself. You should also not catch a stack overflow exception because it's usually impossible to correct it. An out of memory exception is thrown when there is not enough memory to continue executing the program. Once again, this is reserved and thrown by the runtime. So you shouldn't explicitly throw this yourself. The Microsoft documentation states if you choose to handle the exception, you should include a catch block that calls the environment not to fail fast method to terminate your app and add an entry to the system event log. So this is just a selection of commonly encountered exception types. We could go on and on here. But I suggest you check out the Microsoft

documentation if you're interested in going further down the rabbit hole and finding out more about the different exception types available.

Summary

[Autogenerated] So that brings us to the end of this module. In this module, we started off by learning that exceptions represent any error condition or unexpected behavior. During the execution of our programs, we learn that exceptions are organized into a hierarchy with the system dot exception. As the base class, we learned a number of system dot exception constructors. One that takes no parameters, one that takes a string message parameter. And another that allows us to specify an inner exception. If we're wrapping an existing exception, we learned about some of the guidelines for the application exception class. And we also covered some of the commonly encountered exception classes such as the argument exception and the null reference exception. In the next module, we'll be digging a lot deeper into catching throwing and re throwing exceptions. We'll learn how to throw exceptions from expressions. We'll see catching different exceptions with different catch blocks and we'll learn about further refining catch blocks by using exception filtering.

Catching, Throwing, and Rethrowing Exceptions

Introduction

[Autogenerated] Hi, welcome back in this module. We're going to be digging deeper into the topics of catching throwing and also how to re throw exceptions once you've caught them. So we're going to kick off this module by learning that in addition to throwing exceptions from statements, we can also throw exceptions from inside expressions. We'll see an example of how to use multiple catch blocks to execute different code depending on the exception type. We'll also go and add a final block to our code to see it in action. We will learn that once we've actually caught an exception if we want to, we can actually re throw it which will cause it to continue bubbling up the call stack, we'll also learn that

once we've caught an exception, another option is to actually wrap that court exception in a brand new exception instance. In addition to having multiple catch blocks to execute different code, depending on the exception type, we can also filter this in a lot more detail by making use of exception filters. We'll see this in action and finally, we'll see an example of global unhanded exception handling before we head back to Visual Studio. I just wanted to give you a recap on the try catch finally blocks in the tri block, we have the actual code that we want to execute. This can be one or more lines of code and then we have one or more catch blocks. And if we want to, we can specify the type of exception that we want to catch and respond to. We then optionally have a single final block where we can execute cleanup code. So let's head over to Visual Studio now and we'll see an example of throwing an exception from an expression.

Throwing Exceptions from Expressions

[Autogenerated] So here we are back in Visual Studio with our console calculator solution open. What we're going to do is come back to the program file here and down here where we call the calculate method instead of passing in the operation that was entered by the user. I'm just going to specify null and to remove the compiler warning for this null. I'm just going to add the null forgiving operator. Let's go and run this now and we'll enter a first number of 10, 2nd number of two and we'll specify the operation as divide and hit, enter. Notice we get this system dot argument out of range exception thrown telling us that the mathematical operator is not supported. So in this case, we might want a more specific exception to be thrown such as an argument, null exception. I'm just going to hit, enter to exit and we'll come back to the calculator class here. Let's go and make some changes to this calculate method at the top here, I'm going to add a null check. So going to say if operation is null, then throw a new instance of an argument, null exception. One of the constructor overloads for the argument, null exception allows us to specify the parameter name. In this case, it's the operation parameter. Let's just hit F five to run this once again. And once again, we'll choose the divide operator. Notice this time we get the argument null exception thrown. It's telling us that the value cannot be null for the parameter called operation C# seven introduced throw expressions that allow you to throw an exception from within an expression. What we could do here is combine a throw expression with the null coalescing operator. So I'm going to remove this code here and we will start off by declaring a string variable called non null operation. We'll set this to the operation passed into this method and

then we'll use the null coalescing operator. If you're not familiar with this operator, it essentially returns the left hand side if it's not null. But if the left hand side is null. In this case, operation, it will evaluate the right hand side of these two question marks. In this case, this right hand side gets evaluated. It means that operation is null. So we can go and throw the argument null exception. Let me just clean this up. So you can see it on one line. And now if we get past this line of code, it means that the non null operation variable will be a non null value. We can then go and modify this if statement to reference the non null operation variable and then call the divide method otherwise throw the argument out of range exception that we already had. And i'll just go and clean up these comments here. Once again, let's run this. And as before, we can see we get the argument null exception this time. However, it's being thrown from inside this expression. So that's how you can throw an exception as part of an expression. In this example, an expression that's using a null coalescing operator. You can also throw exceptions from switch expressions that were introduced in C# eight. So what is a switch expression? Microsoft tells us the switch expression provides for switch like semantics in an expression context. It provides a concise syntax. When the switch arms produce a value. For example, in this instance of the calculate method notice here we're creating a switch expression. We've currently got two arms of this switch expression, one for the divide operation and one for the addition operation. But because we can throw exceptions from expressions, we could also add this default switch arm to throw a new argument out of range exception. Again, this is just a different example of throwing an exception from an expression. We could also simplify this code by making use of the argument null exceptions throw if null method, but we'll go and leave this code as it is for now, let's take a look next at how we can use multiple catch blocks to execute different code depending on the exception type

Catching Different Exception Types with Multiple Catch Blocks

If you want to execute different code depending on the exception type that was thrown, you can add multiple catch blocks. Let's go and do that here. We're back in the Program class. And we'll go and add another catch block, this time to catch ArgumentNullExceptions. We could go and perform some action here such as logging the exception, and what I'm going to do here is just write out the fact that the operation was not provided. We'll also output the exception details. If we wanted to write code to respond to ArgumentOutOfRangeExceptions, we could add another catch block. If we wanted to, we

could perform some actions such as logging the exception, and here, what I'm doing is writing out Operation is not supported, along with the exception details. Notice up here that we're still hard coding this null value. Let's hit F5 to run this. And because we've got that hard-coded null value, we're getting the System.ArgumentNullException thrown. Notice here, we're getting the message, Operation was not provided. And if we just exit this, we can see that this message came from this catch block here, the catch block that was responding to ArgumentNullExceptions. The other catch blocks didn't execute. Let's go and just fix up this hard-coded null. And once again, we'll pass in the operation entered by the user. Once again, we'll hit F5 to run this, but this time we'll choose an unsupported operator such as the addition operator, and hit Enter. This time we get the ArgumentOutOfRangeException thrown with the Operation is not supported message, which comes from this catch block that's catching ArgumentOutOfRangeExceptions. Once again, the other catch blocks haven't executed. If we run the application once again, and this time what we're going to do is we're going to enter a supported operation of divide, but we're going to go and cause a DivideByZeroException. Let's hit Enter. This time, we get the DivideByZeroException thrown, but we get this more generic message, Sorry, something went wrong. This message is coming from this more generic catch block here where we're just specifying the exception type as the base class, Exception. When you're using multiple catch blocks, you should order them with the most specific exception types first and then more generic exception types lower down. The last catch block should be the most general or, in other words, the least specific exception type where you just catch System.Exception if you need to. Catching System.Exception will catch any remaining exception types if they haven't been caught in catch blocks further up. Let's take a look, next, at the finally block.

Understanding the Finally Block

So once you've added one or more catch blocks, you can add a finally block. To do this, we just use the finally keyword, and then add any code we want to execute inside this finally block. In this case, I'm just going to output the text finally to the console window. Let's go and run this, and we'll just go and divide 10 by 2. This is a valid operation, so we don't get any exceptions thrown or caught, but notice the finally block is executing here. Let's run this once again, and this time we'll cause an exception to be thrown or try and divide by 0. This time, we get the DivideByZeroException thrown, as expected. But notice even though an exception was thrown in the try block, we still get the finally block

executing. Because the code inside the finally block always executes whether an exception is thrown or not, you can add cleanup code here that should always execute, for example, calling dispose on objects that implement the IDisposable interface to make sure you clean up any unmanaged resources. As a side note, you can just have a try and then a finally block with no catch blocks and use the finally block to execute any cleanup code that should always be executed, regardless of whether an exception was thrown or not in the try block. Let's take a look, next at the correct way to rethrow a caught exception.

Rethrowing Exceptions and Preserving the Stack Trace

Sometimes you might want to catch an exception perform some action, but continue to bubble that exception up the call stack. We can do this by re-throwing the caught exception. Let's head back to the calculator class, and we'll go ahead and make some changes here. Once we've determined that the user wants to perform a division operation, we're going to add a try/catch block here. Inside the try block, we call the Divide method, and for the catch block, we're going to catch a DivideByZeroException. Inside this catch block, we could perform some additional actions such as logging, and after we've performed this additional action, we want to continue to bubble the DivideByZeroException up the call stack. So what I'm going to do here is use the throw keyword and specify the exception instance that we caught, in this case, the DivideByZeroException. Let's go and run this modified code. And we'll divide 10 x 0. Notice we get this logging text output here. This came from our DivideByZero catch block, and then we get our general, sorry, something went wrong catch block executes. So at first glance, while this might look okay, if you take a closer look at the stack trace here, you'll notice that it's not actually telling us that the DivideByZeroException occurred inside the divide method. It's telling us here that the DivideByZeroException occurred inside this calculate method, which is not actually correct. I'm just going to hit Enter to exit, and if we look at this code here, notice we're calling this divide method, and if we have a look at this divide method, it's inside this method that the division occurs, and the DivideByZeroException was thrown. The reason we're not getting a correct stack trace is where we're re-throwing the exception here, we don't need to specify the exception instance. In other words, we can just use the throw keyword without specifying the exception that was caught. Let's go and run this version, and once again, DivideByZero, but notice this time, the stack trace is actually correct. It's telling us that the DivideByZeroException occurred inside

the Divide method here. So if you decide to re-throw a caught exception, make sure you use this syntax here so you get a correct stack trace.

Catching and Wrapping Exceptions

[Autogenerated] Sometimes you may want to catch an exception of one type and then wrap it in a different exception type wrapping exceptions should only be used when the receiver of the exception. For example, a catch block in some higher level code would not find the original exception type meaningful. For example, in our demo code, the receiver of the divide by zero exception is the main method. We might decide that a divide by zero exception is not actually meaningful and that perhaps it's too low level an exception type. Instead, we might decide that the main method finds arithmetic exceptions more meaningful. And it doesn't find meaning in all the different types of arithmetic errors like divide by zero or overflow exceptions. In this example, we could catch the divide by zero exception and wrap it in another exception such as an arithmetic exception instance. So what we're going to do here in this calculate method is make some changes. I'm just going to comment out this re throwing code and instead I'm going to throw a new arithmetic exception. Let's start off here by providing an error message. But then we're going to use an overload of the arithmetic exception constructor that allows us to specify an inner exception. In this case, it's going to be the divide by zero exception that we caught in the catch block. Notice here we're using the overload of the arithmetic exception that allows us to specify the inner exception. This is how we wrap the original divide by zero exception. In the new arithmetic exception, we learned about the inner exception property earlier in the course, let's hit F five to run this and we'll go and cause a divide by zero exception. The first thing to notice here is that the more general catch block is being executed, the one that's catching the exception base type. Also notice that the exception details look a bit different. Now we've got this arrow here which is telling us about the inner exception. And here tells us where the inner exception stack trace ends. Essentially we're being told that this arithmetic exception was thrown from this calculate method and that this arithmetic exception wraps a divide by zero exception, which was thrown from the divide method here. Let's just hit, enter to exit as a side note because the divide by zero exception class actually inherits from arithmetic exception even if we didn't wrap the divide by zero. And it was thrown an arithmetic exception. Catch block would still catch a divide by zero exception because divide by zero inherits from arithmetic exception. If you wanted to have one catch block to handle divide by zero exceptions,

and another to handle the more general arithmetic exception. Then you would need to put the most specific catch block first, namely the divide by zero, catch block. And then after it, the arithmetic exception in catch block in the next module, we're going to see how we could actually create our own custom exceptions to represent calculation errors. Let's take a look next at how we can further refine when a catch block executes by making use of something called exception filters.

Filtering Catch Blocks with Exception Filters

C# 6 introduced the concept of exception filters. Exception filters allow you to fine tune when a catch block executes by looking at the exception details in addition to the type of the exception itself. To demonstrate this, we'll head back to the Program class, and we'll come down, and we'll add another catch block right at the start here. This catch block is going to catch ArgumentNullExceptions, but we're going to add a bit of extra syntax here, namely the when keyword. The when keyword allows us to specify additional conditions that have to be true for this catch block to execute. So for example, if we only wanted this catch block to execute when the ArgumentNullException was for the operation parameter, we could examine the exceptions properties, in this example, the ParamName property, and check whether or not it was equal to the string operation. Now we can add code to this catch block that will execute when the operation parameter is null. Let's just go and output the text, Operation was not provided, and down here, we'll change this message to An argument was null. Notice that we've put the most specific version of the ArgumentNullException, the one that has this exception filter, above the more general version that does not have an exception filter applied. To easily simulate this, let's come back to the calculator, and right at the top of this Calculate method, we'll just go and throw this ArgumentNullException for demo purposes. Notice here, we're setting the ParamName property to operation, so this should match our catch block that has the exception filter applied. Let's go and run this, and we'll divide 10 by 2. And because of that hard-coded throw ArgumentNullException, we can see here that we've got this text, Operation was not provided, which is coming from the catch block with the exception filter. Let's go and change this to instead of being operation, it's now going to be number1. Once again, we'll run this. But notice now, because the exception filter is not matching the parameter name of operation, we get this more general An argument was null message. This message is coming from this catch block here, which doesn't have the exception filter applied. Let's take a look, next, at global unhandled exception handling.

Global Unhandled Exception Handling

Sometimes, you may want to execute code when an unhandled exception bubbles all the way up and before it reaches the operating system. For example, in ASP.NET, you might make use of exception handling filters or error handling middleware, and in Windows Forms applications, you might handle the application thread exception event. Because we're working in a console application here, we can use the AppDomain's UnhandledException event. The first thing we're going to do is come back to the calculator here, and we're going to go and remove this hard-coded exception, and now we'll head back to the main Program file. And at the top of this console application's entry point, we're going to declare a variable of type AppDomain called currentAppDomain, and we're going to set this to the CurrentDomain property of the AppDomain. This is going to give us the AppDomain that we're currently working in. We can now go and make use of the currentAppDomain's UnhandledException event. This event will fire when there's an unhandled exception that's been thrown. What we're going to do is wire up this event to this HandleException event handler method, which we need to create. So I'm just going to hold down Ctrl+period and hit Enter to generate this method. To keep things tidy, I'm going to move this method down to the bottom here. We'll go and make this method static, and all we're going to do here is output this message, Sorry there was a problem and we need to close. We're also going to output details about the exception object itself, which we can get from the ExceptionObject property of the UnhandledExceptionEventArgs. If I just come back up to the top of the program here, notice here we're using the Parse method to convert the numbers entered by the user to an int. If we input something that's not convertible to an int, we'll get an exception. We don't currently have a try catch block around this Parse method, so this will create an unhandled exception in the application. What we're going to do is come up to the Debug menu here, and we're going to Start Without Debugging, and let's enter a for the first number. This is going to cause an exception. Notice now we're getting our UnhandledException code execute. We've got a message here, Sorry there was a problem and we need to close, as well as details about the actual unhandled exception in the message, this System.FormatException. Just hit Enter to exit. If you decide to make use of this UnhandledException event, you should read the documentation that I've linked to here because there's a number of things you should be aware of, including what happens if there's multiple app domains in use.

Summary

[Autogenerated] So that brings us to the end of this module. In this module, we started off by learning how we can throw exceptions from expressions including from inside switch expressions. We learn that we can add multiple catch blocks if we want to execute different code depending on the exception type. And when we do this, we should order them from the most specific to the least specific. We saw the final block in action. And we learned that the code inside a final block will always execute regardless of whether or not an exception was thrown from inside the try block. We saw two different ways to re throw an exception. And we saw that if we do this incorrectly, then we lose valuable stack trace information. We also learn that we can catch an exception and wrap it in a new exception type to be thrown up the call stack. And to do this, we're essentially setting the inner exception property. We learn that we can further refine when a catch block will execute by making use of exception filters to do this. We use the when keyword and then specify some logical expression. And finally, we learned how we can respond to unhanded exceptions by making use of the unhanded exception event. Up until this point in the course, we've been making use of exceptions provided to us by .NET itself. We can also create and use our own custom exception types. Join me in the next module when we'll be learning how to do this.

Creating and Using Custom Exceptions

Introduction

[Autogenerated] Hi, welcome back up until this point. In the course, we've been working with exception types that have been provided to us by .NET itself. We can also create our own custom exception types in this module. We're going to start off by getting an understanding of custom exceptions. We'll start off with a high level overview and we'll also discuss when you might want to use custom exceptions. We'll get a high level overview of how we implement custom exception types before heading into Visual Studio and defining our first custom exception class. We'll also learn that we can create inheritance hierarchies of custom exceptions, for example, to create a derived custom

exception that adds an additional property. Once we've defined custom exceptions, we'll see how we can catch them and we'll round out this module by looking at an alternative to custom exceptions if all you want to do is add some additional data. So let's kick off this module by getting an overview of custom exceptions.

Understanding Custom Exceptions

[Autogenerated] The first thing to be aware of when it comes to custom exceptions in net is that you should actually use the predefined net exception types where they're applicable rather than creating your own custom types. For example, you should use the invalid operation exception. If a property set or method call is not appropriate for the current state of the object or the argument exception for invalid parameters. In these situations, you normally wouldn't go and create custom exceptions. If you catch an exception and then choose to throw an instance of a custom exception. Instead, you may want to consider wrapping the court exception inside the custom exception. We learned about exception wrapping in the previous module. And you also shouldn't use custom exceptions for the normal expected logical flow of your application exceptions as their name suggests should only be used for exceptional circumstances and not normal expected logical program flow. When it comes to deciding whether or not to create and use custom exceptions, you should only create custom exception types. If they need to be caught and handled differently from existing predefined net exceptions or exceptions from libraries or frameworks. There's no point in creating exception types, if you're never going to catch them. For example, if you wanted to perform special monitoring of a specific critical exception type, you could go and create a custom exception to allow you to catch that specific critical exception and perform the monitoring. Another reason you might want to create custom exception types is if you're building a library or framework for use by other developers, so the consumers of your library can react specifically to errors from your library. We'll see an example of this in just a moment with the Jason .NET exception types. Another reason you may want to consider using custom exceptions is if you're interfacing with an external API or DL or service of some kind, and you want to wrap any errors from that external service in a custom exception type. For example, if an API DLL or service that's external to your code, return status codes rather than throwing exceptions, you may want to translate those exception codes into exceptions so you can catch them using normal .NET catch statements. When it comes to implementing custom exceptions, you should follow the

naming convention whereby you end the name of the class with the word exception. And you should implement the standard three constructors which we will see in action in Visual Studio. In just a moment. Once you've created your class and implemented the constructors, you can add additional properties where needed as we discussed earlier. In the course, you should never inherit from application exception instead, you should inherit from system dot exception or your own custom exception types. If you're creating an exception hierarchy, you should always try and keep the number of custom exception types to a minimum and only create them where necessary as an example of an exception hierarchy. If we take the Jason .NET library as an example, the library defines this Jason exception class which inherits from system dot exception. This allows consumers of the Jason .NET library to catch and respond to errors in the Jason .NET library separate from other exception types in the Jason .NET library. This Jason exception is the base class for all exceptions that occur while using the library. But there's a number of derived exception classes such as the Jason reader exception. Jason serialization exception, Jason writer exception and Jason schema exception. Let's head over to Visual Studio now and we'll go and create our first custom exception type.

Defining a Custom Exception

So here we are back in Visual Studio with our ConsoleCalculator solution open. We're going to start off by defining a more general custom exception type to represent the fact that something went wrong during calculation. To do this, the first thing we're going to do is add a new class to our ConsoleCalculator project, and we're going to call this class CalculationException. We'll just add that, and notice that we're following the naming convention where we end the class with the word Exception. I'm going to change this class to public because in the next module, we'll be writing unit tests against exceptions, and we're going to go and add a constant string to this class. This constant is going to be called DefaultMessage, and this is going to specify a message to be used by default if the user of this exception doesn't provide a custom message string. So we're just going to say here that An error occurred during calculation. Ensure that the operator is supported and that the values are within the valid range of values for the requested operation. So this is a very general message. We can now go and create constructors for this exception. The first constructor that we're going to create is going to be the default parameter list constructor. And when we use this constructor, we're going to set the default message from the constant. To do this, we'll define the default constructor here, and

then we'll call into the base class's constructor and pass into the base class's constructor this DefaultMessage string. At the minute, this class is not inheriting from any other class. So to actually make this an exception that we can throw, we're going to inherit from System.Exception. And if we just have a look down here, we can see here we're calling into the System.Exception's constructor, which allows us to specify a string message. So this first constructor will automatically set a default message, but we also want to allow the user of this exception to specify a custom error message. So, we'll define a constructor that takes a string message parameter. And as before, call into the overload of the base class constructor that allows us to specify the message. In this case, it's not the default message. It's the message provided by the user. The third of the three basic constructors that we're going to create here will allow the user to specify an inner exception to be wrapped. So we'll create this constructor overload that takes an Exception parameter, and once again, call into the base class constructor. This time, we're going to use the overload of the base class constructor that takes a message, in this case, the DefaultMessage and also an innerException. This will be the exception passed into this constructor by the user. We can continue to add constructor overloads that we think will benefit the user of this custom exception. For example, we could add this constructor overload that allows the user to specify both a custom message and an innerException. Let's just build this to make sure we haven't introduced any errors, and the build succeeds. So that's how easy it is to create an exception class. Just as we saw with Json.NET, we could go one step further and create a number of derived exceptions from this CalculationException type. Let's take a look at that, next.

Defining a Derived Custom Exception

Before we go and add a new Exception class, let's just go and clean up this class. I'm just going to hit Ctrl+period here, and I'm going to convert this class to a file scoped namespace. This will just help reduce nesting in this file. You can make use of file scoped namespaces from C# 10 onwards. While we're here, we can also go and clean up these unused musings. Once again, I'm going to hit Ctrl+period, and then hit Enter to remove unnecessary usings just to clean up this file a little bit. Let's go and add a more specific type of exception to represent the fact that a requested operation such as subtraction is not supported by the calculator. We could, of course, use .NET's ArgumentOutOfRangeException for this instead of creating a custom exception, but let's suppose for some reason we wanted to catch and respond to this situation specifically. The first thing we're going

to do is add a new class to our project here, and this time we're going to call the class CalculationOperationNotSupportedException. We'll just add that. We'll convert this file to use file scope namespaces. And instead of inheriting from System.Exception, this time we're going to inherit from our CalculationException that we just created, and we'll also clean up these unused usings. As before, we're going to start by defining a DefaultMessage, Specified operation was out of the range of valid values, and then we can add a number of constructors. We'll go and add those in just a moment. But first, we can add one or more properties to add additional data to this custom exception type. So for example, we could have this string property called Operation that we can set to indicate the operation that caused this exception. We'll go and add this constructor, which will use this DefaultMessage here. We'll add a version of the constructor, which uses the DefaultMessage and a supplied innerException. We'll add this constructor overload that allows the user to specify a custom message and an innerException. We can also add constructor overloads to allow the user to specify the Operation property value. Notice here the string operation constructor parameter. This is going to set the Operation property of this class, and we can continue to add constructors as we see fit. In this case, one that allows us to specify both the operation that caused the exception and also a custom error message. If we wanted to, we could override the message property, which is ultimately coming from the System.Exception class. And for example, if we haven't set the operation property in this class, we could just return the message that's been set. But if the operation has been set, we could instead return the message, followed by a NewLine, followed by the Unsupported operation here. Once again, we'll build this to make sure we haven't introduced any errors, and the build succeeds. If, however, we come down to the Error List here, notice we've got these warnings telling us that the non-nullable property operation must contain a non-null value when exiting the constructor. And if we scroll up here, we can see these constructors are causing this warning. That's because these constructors don't explicitly set this Operation property. Because this project is set to use nullable reference types, we need to mark this operation as being nullable. To do this, we just need to make this a nullable string. And if we come down to the Error List now, notice those warnings have disappeared. So now that we've got our two custom exceptions, let's see how we can make use of them.

Using Custom Exceptions

To make use of our custom exceptions, what we're going to do is come back to the Calculator class that we saw earlier in the course, and we're going to come into this Calculate method. What we're going to do is where we try and divide these two numbers, we're currently catching this DivideByZeroException. We're going to change this to be Exception. And instead of throwing this ArithmeticException, we're going to throw an instance of our CalculationException. Recall that this custom exception is the more generic generalized version, and we also created an overload of the constructor that allows us to wrap an existing exception. In this case, it's the exception that we caught here. If we just scroll down here, if the supplied operation is not supported, instead of throwing this ArgumentOutOfRangeException, we're instead going to throw an instance of our CalculationOperationNotSupportedException. One of the constructor overloads that we added was this one, which allows us to specify the operation. This is the operation that was passed into this Calculate method. We can now go and make some changes to this Program file. If we just come down here, we'll make some changes to these catch blocks. We'll change this ArgumentOutOfRangeException catch block to instead catch our custom CalculationOperationNotSupportedException. And we'll just change the code here to write out the Operation property. We'll add an extra catch block here, this time catching the more general CalculationException. Notice that we put the more specific derived CalculationOperationNotSupportedException higher up and the more general version lower down, and now we can go and run the modified application. What we'll do is enter an unsupported operation, in this case, addition because we currently only have division supported. I'm just going to hit Enter. Notice that we get our custom CalculationOperationNotSupportedException thrown and caught. And it's telling us here that the unsupported operation is this addition. Let's just hit Enter to exit. We'll just go and run the application again, and this time, we'll specify a supported operation, division, but we'll try and divide by 0, which is going to cause an exception. We can see here that we caught our custom CalculationException. We're getting the default error message that we specified in the constant here. And we can also see here that we've got the wrapped DivideByZeroException. If you happen to find that you're creating a lot of custom exceptions simply to add some additional data to the exception type, there's a little known technique that you should be aware of that may sometimes offer you an alternative and mean you don't have to create custom exceptions. Let's take a look at this, next.

An Alternative to Custom Exceptions

[Autogenerated] The system dot exception class defines a property called data. This property is of type I dictionary which allows you to store key value pairs. When you add items to this I dictionary, you should specify the key as a string and the value can be any object. This means you can add an arbitrary number of additional data items to an exception. But under the covers, this property is implemented as a list dictionary internal which is optimized for a smaller number of items. The Microsoft documentation states that this list dictionary internal is smaller and faster than a hash table. If the number of elements is 10 or less, essentially, you can use this data property to add additional or supplementary user defined exception information. This can be used as an alternative to creating custom exceptions. If all you want to do when you create those custom exceptions is to add additional data properties. So suppose we had this calculate method here and in the block, we're throwing a new argument out of range exception if the operation is not supported. Currently, the code here is creating the new argument out of range exception and throwing it in the same line of code, but we could modify this and instead create a variable to hold the argument out of range exception. Once we've created this variable, we can access the data property, we can call the a method of the dictionary and specify a key and value. In this example, we're specifying this string key supplied underscore operation and for the value, it's the operation supplied by the caller. For example, division or addition. once we've added the additional supplementary information to the exception instance, we can simply go and throw it, we could now go and catch an argument out of range exception and try and get the supplied operation. For example, we could check whether or not the data property contains a key of supplied underscore operation. And if the dictionary does contain that key, we can get access to the value. We can now make use of that additional supplementary exception information. In this case, the supplied operation here, we're just writing out the supplied operation to the console window. There's a few things to be aware of. If you decide to use this data property. One important thing to note is that this property is not secure. Any catch block can read keys and values. Any catch block can change keys or values. Any catch block can delete items in the dictionary and any catch block can add new items. So you shouldn't use this property to store sensitive or confidential information. Another thing to note is that your application should still operate correctly. Even if the expected data items are missing or corrupted, you should also be careful of key conflicts in the dictionary, especially if you're catching and re throwing exceptions. But before you re throw them, you add extra items to the dictionary. If you try

and add a new entry to the dictionary and the key already exists, you'll get an argument exception thrown.

Summary

[Autogenerated] So that brings us to the end of this module. In this module, we started off by getting a high level understanding of custom exceptions. We learn that we should use existing predefined net exception types or types from libraries where applicable and that we should only create custom exception types. If they need to be caught, we learn that when we create custom exception classes, we should name them with the word exception. At the end, we then dove into Visual Studio and we created the calculation exception and we also saw how we can create our own exception hierarchies. We went and created the calculation operation not supported exception. And we also saw that we can add custom properties to our custom exceptions. We saw how to throw and catch custom exceptions and finally, an alternative to custom exceptions, the data property of the system dot exception base class. One thing that's sometimes overlooked when writing automated tests is to test that exceptions are thrown at the correct time in the next module. We'll be learning how we can test that exceptions are in fact thrown using the popular unit testing frameworks.

Writing Automated Tests for Exception Throwing Code

Introduction

[Autogenerated] Hi, welcome back. So once you've implemented error handling in your .NET application, perhaps you've thrown some exceptions, reth thrown some exceptions, added multiple catch blocks and you may have even made use of exception filters. But how do you know that the correct exceptions are being thrown at the expected times? We can actually write unit tests to check that the correct exceptions are being thrown. We can check that the type of the exception is correct and we can even go one step further and check that the properties of the throne exception are as

expected. So in this module, we're going to be learning how we can test exceptions using the N unit testing framework. We'll see how we can do a similar thing with the X unit .NET testing framework. And we'll also see how we can test exceptions. If we're using MS test, we're going to finish up this module with the key course takeaways and we'll also talk about some resources and further learning material. So without further ado let's head over to Visual Studio and we'll see how we can test exceptions with the N unit testing framework.

Testing Exceptions with NUnit

So here we are back in Visual Studio with our ConsoleCalculator solution open. In these demos, I'm not going to be digging deep into the testing frameworks and explaining exactly how all parts of them work. I'm just going to be focusing on the exception testing features, as I've already got dedicated courses on NUnit, xUnit.net, and also MSTest over at pluralsight.com. What we're going to do is add a new project to this solution. So I'm just going to come down to Add, New Project, and we're going to search for nunit, and we'll select this NUnit Test Project, and hit Next. We'll call this test project ConsoleCalculator.Tests.NUnit, and click Next, and we're also going to leave this as .NET 6. We'll just hit Create to create that project. We'll go and add a project reference to the ConsoleCalculator project. And up here, we'll convert this to use file scoped namespaces just to reduce that nesting. I'm also going to delete this SetUp method and rename this test class to CalculatorShould. We'll also rename the physical file here. Here, we have the first test method that will be executed when we run our tests. I'm going to rename this method to ThrowWhenUnsupportedOperation, and we'll start off by creating an instance of the thing we want to test, in this case, our Calculator class. The acronym sut here stands for system under test, and it just helps us to identify the thing that we're testing. So in this test, what we want to check is that if we call the Calculate method with an unsupported operation such as addition, that we get the CalculationOperationNotSupportedException thrown. So we'll start off by making use of NUnit's Assert class, and we'll call the That method. The first thing we want to do here is choose what action should cause the exception to be thrown. So I'm just going to use a lambda here, and we're going to call the Calculate method on our system under test, the Calculate class. We'll choose a couple of numbers, and for the operation here, we'll specify addition, which is not currently supported. We can now assert that when this Calculate method is called that it throws the TypeOf exception, CalculationOperationNotSupportedException. This is how we can check that a method

throws an exception. We could also follow a similar approach, but instead of calling a method, call a property setup, for example. Let's go and build this to make sure we haven't introduced any errors, and the build succeeds. What we're going to do is use this Test Explorer window here to execute the test. If you don't see this window, come up to the Test menu, and just select Test Explorer. If we expand this down, we can see our test project, test classes, and test methods, so we'll go and right-click here and choose Run to run this test. And we can see here the tick mark goes green, and this test is passing. At the moment, all this does is check that the correct exception is thrown. It doesn't check any of the properties of the thrown exception to make sure they're correct. Let's go and add another version. I'm just going to comment out this version and paste in this version. Once again, we're calling the Calculate method here with an unsupported operation. And once again, we're using the Throws.TypeOf method and specifying the type of exception we expect, but notice we've chained on a couple of additional items now. We've got this .With to allow us to inspect the thrown exception. And we're saying the thrown exception should have a property called Operation, which has a value, EqualTo addition. Recall that this Operation property was created when we created this custom exception. Let's just build this and run the test once again, and once again, the test passes. If we were to go and open up the Calculator class and come down here where we're throwing the CalculationOperationNotSupportedException, we can introduce a bug. For example, here were accidentally using a string literal and not the operation passed into the method. If we come down to Test Explorer and run this test again, it should fail. We can see now that the test fails, and it's telling us here we expected the CalculationOperationNotSupportedException to be thrown with a property equal to addition, but the property was actually xyz. Let's just go and fix this up. I'm going to undo that change, and once again, this test will pass. Let's come back to our Test class. An alternative approach is to actually get the exception object itself and then perform individual asserts against it. Let's just comment out this code, and we'll paste in this version. Notice this time instead of using Assert.That, we're using Assert.Throws, once again specifying the type of the exception we expect. We're also calling the Calculate method, once again, with the addition operator. So if this Assert.Throws method notices that a CalculationOperationNotSupportedException was thrown, this ex variable will be populated with the exception instance. We can now go and make individual asserts against this exception object such as checking that the Operation property Is.EqualTo addition. Let's just run this updated test code, and once again, the test passes . I'm just going to comment out this version. One

thing to bear in mind is the code that we've used so far is expecting the exact exception type to be thrown. So, for example, if we had this code, notice here, we're specifying the type of the exception as CalculationException. This was the base exception class we created earlier. So even though CalculationOperationNotSupportedException inherits from CalculationException, if we run this test code, it's going to fail. And we can see in the message here, we're expecting a CalculationException, but we're actually getting a CalculationOperationNotSupportedException. When you use the Throws.TypeOf method here, the exact exception type has to match. Even if it's a derived exception, this code won't see that as a matching exception type. If you do actually want to match against base exception types, even if more specific versions are being thrown, instead of the TypeOf method, you can use the InstanceOf method. If we run this now, the test will once again pass. Let's have a look, next, at how we can check exception throwing code using xUnit.net

Testing Exceptions with xUnit.net

Once again, we're going to add a new project to this solution. This time, we're going to search for xunit and select this xUnit Test Project. We're going to call this ConsoleCalculator.Tests.xUnit. Once again, choose .NET 6.0, and hit Create. As we did before, we'll add a project reference to the thing we want to test. In this case, it's going to be the ConsoleCalculator project, and we'll also hit Ctrl+period and hit Enter to convert this to file scoped namespaces. To save time here, we won't worry about renaming the Test class. What we will do, however, is rename this test method. We'll just say ThrowWhenUnsupportedOperation. As we did before, we'll start off by creating an instance of the thing we want to test, the Calculator, and this time, make use of xUnit's Assert class. With xUnit, we say Assert.Throws, and then as the generic type, specify the type of exception we're expecting. We can now specify the action that should cause this exception. As before, we're calling the Calculate method with an unsupported operation. Let's build this to make sure we haven't introduced any errors, and the build succeeds. And if we head down to Test Explorer, we can see our xUnit test project here. And if we expand this down, we can see our test, and if we run that, the test passes. If we were to specify the base CalculationException class and run the test, as with NUnit, the test is going to fail because the Throws method expects the exact exception. If you want to check for an exception or a derived exception, instead of the Throws method, you can use the ThrowsAny method. Once again, we're specifying the base CalculationException. But this time, if we run the test, the test passes. As we did

with NUnit, we can also capture the thrown exception into a variable, and then make further asserts against that exception object, in this case, checking that the Operation property is equal to addition. Once again, we'll run this test, and the test passes. Let's take a look, next, at how we can check exceptions with MSTest.

Testing Exceptions with MSTest V2

So once again, we'll go and add a new project to this solution. This time, we're going to choose the MSTest Test Project template. We'll call this project ConsoleCalculator.Tests.MSTest, and click Next, and hit Create. We'll go and add a project to dependency, and reference the ConsoleCalculator project, and also fix up this namespace to be file scoped, and change the name of the test to ThrowWhenUnsupportedOperation. We'll go and create the Calculator instance. And with MSTest, we make use of the Assert.ThrowsException method. Once again, specify the expected exception type as the generic type parameter, and once again, call the Calculate method WhenUnsupportedOperation. If we come back down to Test Explorer, and just expand this MSTest project down, and run this test, the test passes. We can also get the ThrownException object itself to perform further asserts against such as checking that the Operation property has been set to addition. Let's run this version, and it passes. When you use the ThrowsException method from MSTest, it expects the exact exception type to match. Unlike xUnit.net, MSTest doesn't have a ThrowsAny method that will also allow for derived exception types.

Summary, Key Takeaways, and Resources

[Autogenerated] So that brings us to the end of this module. In this module, we started off by learning how we can test exceptions using N unit. We saw that we can use the assert dot that method in conjunction with the throws dot type of method. And if we want to specify that derived exceptions are allowed, we can instead use the instance of method. Next, we saw X unit .NET in action, we saw we can use the assert do throws method to match an exact exception or the assert dot throws any method to also allow derived exception types. Finally, we saw MS test and we learned that MS test gives us the assert dot throws exception method that matches on an exact exception type and not derived exceptions that also brings us to the end of this course. Let's take a look at some key

takeaways that we've covered along the way. The first thing is that in net, we use exceptions to manage errors and not error return codes. We learn that when an exception is thrown, it bubbles up the call stack until they get caught by a catch block or crash to the operating system. And we learn that all exception types inherit from system dot exception. We also learn that we can set the inner exception property which allows us to wrap exceptions inside other exceptions. And we learn that if we want to respond to exceptions, we can use the try and catch blocks. We also learn that we can use a final block to execute any cleanup code. When it comes to adding catch blocks, we shouldn't add catch blocks that do nothing or simply re throw the exception. There's no point catching an exception should add value of some kind. We also learn that we shouldn't use exceptions for normal program flow logic exceptions should be for exceptional circumstances. We learn that if we've caught an exception, we can read through it, but we should be careful to preserve the stack trace when doing so. And we also looked at creating custom exceptions. We learn that we should only create custom exceptions when the existing exception types are not suitable. When we're throwing exceptions, we should be careful to avoid adding sensitive information because exceptions can sometimes be logged or interpreted higher up the stack trace when it comes to additional resources. And further learning, you may wish to check out the Microsoft C# programming guide to exceptions and exception handling. You can find it at this link here and you may wish to also check out the framework design guidelines for exceptions which you can find at bit iFrame as I mentioned earlier. I've got a number of related plural site courses including the testing .NET code with X unit .NET getting started course, you can find this at bitly slash X unit course. If you want to learn more about MS test, you can check out my PL site course here. And if you want to learn more about unit testing with N unit three, you can check out bitly slash N unit course, which will take you to my pl site N unit course. So that brings us to the end of this course. I'm Jason Roberts and I hope you enjoyed watching.