Course Overview

Course Overview

Hello, everyone! My name is Gill Cleeren, and welcome to my course, C# Fundamentals. I'm the CTO of Xebia Belgium. C# is a very popular language to build applications with. Using the language, you can build all types of applications, including web applications, desktop applications, games, and even AI-powered applications. C# is of the .NET ecosystem, and it is an actively maintained language, which means it receives frequent updates. This course will be your guide to learn how to build applications using C#. First, after we make sure that your machine is ready to go, you'll learn about the C# syntax first. C# is an object-oriented language, so you'll learn about created classes and using objects. Next, you'll learn about common tasks, such as debugging and writing unit tests. And finally, you will learn more advanced concepts, such as catching exceptions in your code. Some of the major topics that we will cover include learning about C# and its relation to .NET and Visual Studio, understanding the C# syntax, creating console applications and capturing input, learning the basics about object-orientation, and concepts such as inheritance and interfaces, creating unit tests, and debugging your C# code. By the end of this course, you will have a good understanding of the C# language. This course does not expect any knowledge about C# or .NET. I hope you will join me on this journey to learn C# with the C# Fundamentals course here at Pluralsight.

Getting Started with C# and .NET

Module Introduction

Hello, and welcome to C# Fundamentals. My name is Gill Cleeren, and I must say I couldn't be more excited that you are joining me to learn C# together with me. We're going to have a lot of fun learning this awesome language. C# is a development language, and it's a great choice to build applications

with. I simply love it! It's a modern and actively maintained language that can be used to create all sorts of applications with, from very small applications to complex enterprise application architectures. C# is part of .NET. So, what is .NET then? Well, great question! .NET is a developer platform created by Microsoft which allows us to build or develop applications. Indeed, we need to develop our applications on the .NET platform. .NET is open source, too. So that means that we can see all the code that Microsoft created for .NET. Using .NET, the sky is the limit in what we can build with the platform. You can start out now with simple apps and just learn the basics, which is what we will be doing in this course. But .NET has got you covered and can grow with you to build all sorts of applications, including desktop applications with the graphical interface, cloud apps, web applications, games, mobile applications, and even AI, or applications powered with artificial intelligence. .NET is, quite frankly, huge. When we are building applications with .NET, we are tapping into the entire ecosystem that will help you with all your endeavors. .NET contains several languages that we can use to write our code. There's a very extensive set of libraries that you can use which offers a lot of functionalities that's already been created for you, both from Microsoft and from the community. And finally, .NET comes with several tools that you can use to create your applications. Tools are available for all environments, including Windows, Mac, and Linux, thus giving everyone the ability to write .NET applications. In this course, we're using .NET 8. One of the cool things about .NET is that it can run everywhere. .NET is what we call cross-platform. It is not tied to just Windows, for example. So just like the tools we use to create them, the applications we create with .NET can run on Windows, Linux, and Mac. As I just mentioned, there are different languages available to write .NET applications, one of them being C#. In this course, we will be using C# 12. Other languages include Visual Basic and F#. To follow along with this course on C#, no previous knowledge about the language is required. This course will teach you everything you need to know right from the very beginning. So this course is aimed at learning how to write .NET applications with C#. If you're coming from another language, like Java or C++, this course will be a great fit, too, to help you gain the required knowledge to write C# applications. I will assume only some very basic understanding of programming concepts. But again, just the very basics. If you don't have these yet or are in doubt, no worries; we've got you covered! We have an amazing course here on Pluralsight called What Is Programming?, of which you can see the link here on the slide. Take a look at this course if you want to understand the basics about computer programming. Through this course, we will be building a few applications together. How great is that?

And all of these are focused on one of our customers, Bethany. She runs a store where she sells the most amazing pies called Bethany's Pie Shop, and we are building the software for her business. We will be looking at some applications to manage her staff throughout this course. The concepts used in this course include employees, managers, wages, taxes, and so on. So, pretty general concepts that should be clear to understand. I am a big fan of doing rather than watching a course. For that reason, the entire course is built so that you can follow along with everything I'm doing. Please try to write the code I'm writing. You will learn a lot this way. To be able to follow along, you'll need the exercise files, which you can find here on the Pluralsight website. Head over to the Exercise files tab to start the download. Oh, and before I forget, please post any questions you may have under the Discussion tab. I'm happy to help with any issue you encounter with this course. This course was created for use with C#, which at the time of the recording of this course is the latest available version. C# is part of .NET, and so we're using again the latest version here, too, which is, as mentioned, .NET 8. We will soon talk about the different options you have to create your applications. I am using the latest version of Visual Studio, which is the 2022 edition, and I'm using the free edition named the Community Edition. I'll show you where to get it later in this module. All code and samples will run perfectly fine if you use the same versions that I am using. However, nearly all features of C# that we will be using in this course will work fine with all the versions, too. So that includes C# 10 and C# 11. As said, we're using .NET 8 in this course, the latest at the time of the creation. But also if you're using .NET 6 or .NET 7,, you're good to go. I do recommend to work with the latest versions, though, as we are doing in this course. .NET and C# evolve quickly nowadays, so it's best not to stay behind. However, for the fundamentals we are covering here, not much, if anything, is changing in between different versions, so don't worry about it too much. With all of this out of the way, we are ready to start learning C#. So, let's get our hands dirty, shall we? We now already know a bit about .NET, so it's time to learn a bit more about what C# is. I will do that in the first part called Hello C#. Now I did mention, I hope you will follow along with me during this course. I will therefore make sure that you have everything you need. So, we'll set up your machine next. And with the tools installed, we will create together our first C# program, and we will understand the basics of the C# language already. Sadly, while writing code, we'll make errors and we will introduce bugs into our code. Those can be nasty to find, but we're in luck. .NET comes with a great debugger that will help us greatly to pinpoint errors in our applications, and I'll show you how to use this next. We can also create applications using VS Code, short for Visual

Studio Code, and the CLI, so using the command line. I will show you this alternative option here as well. Understanding how to find things you don't know is hugely important, so in the last part of this module, we'll explore the wonderful .NET docs which are indispensable when learning and using C# and .NET. I'm excited to get started! I hope you are, too. Let's dive in!

Hello C#

As promised, let's get acquainted to a bit more with the language we'll be learning here, C#. As you know by now, C# is a programming language, a very popular and well-maintained one, I may add. C# is indeed the most popular and most commonly used language to build .NET applications. C# is an object-oriented language, meaning it has everything on board to allow for object-oriented development. If you're not familiar with object orientation, or OO for short, no worries, of course. It's something we will spend a lot of time on later in this course. It is is also a typesafe language, which in short means that when we compile our code, the compiler can check that we are using types correctly. And if not, it will flag an error. Again, something you will learn in this course. If you already know one of these learning C# will be even easier a development language like CS Sharp needs to evolve. CS Sharp was introduced quite some time ago back in 2002 together with the release of .NET itself. Over the years, Microsoft has invested a lot in the language. And as you can see here, we have received many updates over the years. We're now at C# 12, which is the version covered in this course, some of these updates brought major new features while others were more limited in scope. But one of the most interesting things about this evolution is that nearly everything is backwards compatible if you would compile the C# one application today. So with the current compiler, things would probably still compile fine. How cool is that C# is a set a development language for .NET application. And together with .NET it will enable us to build all types of applications. C# can run pretty much everywhere. In this course. To avoid any distractions, we will use console applications, console applications use the terminal. So a text based interface to interact with the user. But using C# we can build much more than that, you can create desktop applications. Think of applications like a word processor or a data entry application that runs directly on the user's machine CRP can also of course be used to create web applications or websites, think of webs shops, online invoicing systems or a registration portal, mobile applications that run on iOS or Android can also be built with CRP. And also applications that

run without a graphical interface can be created using C# Windows services. Long running background processes are a good example of this and they do a set can be created using C#

Setting up Your Environment

I hope you're excited now to start learning how to code in C#. We'll do that in a minute. We first need to make sure you're all set up to actually do. So let me show you how to set up your environment to start coding in C# first. To create .NET applications with C#, you actually have several options, and I'll show you the two most commonly used approaches and those we'll use in this course, too. Most developers write their C# applications using Visual Studio, of which the latest version is called Visual Studio 2022. On Windows, this is the most commonly used way to create C# and .NET applications. The second option to write our C# applications is using the .NET CLI and Visual Studio Code. CLI is short for command-line interface. Using the command line, we can write commands to create new .NET applications, compile demands on. Visual Studio Code is the code editor you would typically in combination with the CLI. And the nice thing about this approach is that it's available for really all platforms. So also if you're on Mac or on Linux, you can use this approach. Both approaches can be used, and I will show you both so that no matter what platform you're on to learn C#, you can do so. Now for completeness, let me add that there are other options to create C# applications for third parties, such as Jetbrains Rider. And I'll start with the approach that I will use most in this course, and that's the one based on Visual Studio 2022. Visual Studio is Microsoft's flagship integrated development environment, or IDE for short. The term IDE points to applications that can be used to create software and typically contains a code editor and debugger, but also a lot more bells and whistles to make your software developer life easier. Visual Studio 2022 is a large application, but you don't have to know all the details to get started with it. I will show you just enough to get you on your way with it. There are several editions of Visual Studio available, different levels, let's say. First and foremost, to get started with C# development, you don't have to pay a dime. There's a free fully featured edition of Visual Studio available called the Community Edition and it is yes, totally free, and yes, you can even create commercial applications with it. It is by no means a trimmed down version of the other editions. On the contrary; it's a complete package that contains almost the same set of features as the other paid editions. We will use this version in this course. Do note that there are some license restrictions which do apply. Now check the Microsoft website for more info on these. The

second edition is the Professional Edition, which is paid for and comes with some extra features. It's aimed at developer teams, but not really the large ones. Finally, there's the Enterprise Edition which is also paid, and that one, as the name gives away, is aimed at larger teams. Most important lesson here, we can use the Community Edition for free. Yay! Thank you, Microsoft!

Demo: Setting up Your Environment Using Visual Studio

Let me in the first demo of this module take you through the setup of your environment so that you have the correct tools installed to get started. In this course, we'll use Windows, and in there, I will use Visual Studio 2022. You can get a free copy of a Visual Studio by going to Visual Studio at microsoft.com. And there you will be able to download the Community Edition. The other editions, of course, will also do fine. If you click here on Community Edition, it will download the installer for you. Click on the installer to get started installing Visual Studio. And this is the screen that you will get. You will see that Visual Studio is based on this thing called Workloads. Based on the different type of development you want to do, you can actually install the related files. So by default, it will install a core setup, let's say, to get started, but then if you're interested in web development, you can click here to install this workload. If you want to do cloud development, then you can do this workload. You see here also that we can do mobile and desktop development as well. You don't need all of these workloads. At this point, we are just going to be using console applications. I'll explain what the console is in just a minute, but you can call on to the screen again later on, if you're interested, if you're willing to start, let's say, building another type of applications. Once you have installed Visual Studio, click on the icon to launch it. And the next demo, we'll see how to use Visual Studio for our first application.

Building Your First C# Program Using Visual Studio

With the tools ready, we're good to go and write our first C# program using Visual Studio 2022. Let's get acquainted with some of the basics of the IDE. We will start writing C# code in just a minute. But our code needs to, well, belong somewhere. Visual Studio, of course, contains a code editor, but our code will belong to what is known as a project. Projects are containers for code files, and they will contain one or more files which contain our code. Once we are done writing our code for our

application, projects will be compiled by the compiler. They are the unit that will get compiled. Visual Studio comes with a large set of starter projects, too, called project templates, for all kinds of applications really. They can help to get you on your way. In this course, we will focus on learning C#, and for that purpose, we will use the console applications template, which is exactly what I'll show you in the upcoming demo. As already mentioned, console applications use a terminal text-based interface, and so we don't need to bother creating a graphical interface. Of course, the most important task at hand is writing code. Visual Studio comes with a code editor. While code is in essence not much more than plaintext, the Visual Studio Code editor does come with many added features. As you can see, different colors are being used by the editor to highlight points of the code. Code highlighting will make your code much more readable. You'll see along the way in this course that the editor will try to help you in many ways, even cleverly suggesting what you may be writing next.

Demo: Creating Your First C# Application

Time to get to work. We are going to create our first application using our freshly installed Visual Studio 2022. We'll use the template and we'll take a look at the generated C# kit. Next, I will show you what steps we need to do to run our first program. Now, in this demo, I'm going to make you a C# developer already. Now with Visual Studio launched, we arrive here on this start screen from where we can actually launch a few operations. What we'll do is we'll create a new project. As mentioned in the slides, the project is really the container for all the code and all the files really that make up your application. When we compile our application later on, so build the application to make it executable, the project will also be what gets compiled, what gets built. So click here on Create a new project, and we'll build our first project together. What we see here in this Create a new project window is a list of templates. Microsoft already provides us with some starting templates to get us on our way when building applications. As mentioned, we'll keep things simple and we won't be focusing on the UI, really; we will be working with a Console Application. And the console uses a terminal, so a string-based, a text-based terminal that allows us to see text messages and also input text that we can work with in our application. So we'll select the console application here. Now, do make sure that in the list of languages, you are using C# and that also here in the Console App template, you're actually selecting the C# version of the Console App. Notice also here that this type of application can easily be executed on Windows, but also on Linux and on Mac. That is that cross-platform functionality that I

was referring to earlier. In this window, we are asked to give our project a name. I will give it a name. Let's call it HelloFromCSharp. You can also choose where you want your application to reside. When we click on Next, we are asked for the version of the framework. As mentioned, we are using .NET 8 here, but you can also use different versions of .NET if that would be a requirement for your application. When I now click on Create, what Visual Studio is going to do, it's going to execute that template and generate a few files for us already. There we go, we have generated our first application based on that template, that project template that came with Visual Studio. Now without doing anything just yet on the code, what I'm going to do is I'm going to run the application. And to do so, we can actually click here on that green arrow here that also shows the name of the project again, HelloFromCSharp. And when I click here, Visual Studio will compile the application and also run it in one go. That goes pretty fast typically. And you see here that a console window, a terminal window that shows us a little bit of text, is showing, and it is showing Hello, World! Congratulations! You have created your first C# application, You're now officially a C# developer. Although you haven't written any code yet, but we'll fix that very soon. Now without going into too much detail, what we see here in Visual Studio is a code editor. Of course, writing the code is the most important task that we do, and therefore, also the code editor is very important in what we see here. And the code isn't just plain black text on a white screen. It's that it's using quite a few colors to give us an indication of the meaning of the code. Now, before we close this demo, there's one question I'd like to solve at this point. How did we get that Hello, World! to show on that console window? Well, here you see line of code that says Console.WriteLine, and I'm passing Hello, World! So using this console here, we are able to write text to the console, to that terminal, that is, and it is also possible to accept text from the console, and we'll see how to do that in the next demo.

Demo: Writing C# Code

So far, Visual Studio did most of the work. Now, time to change that. In this demo, we will already write some C# code ourselves. C# code and code in general needs to be correct, so we need to oblige the prescribed syntax. If we do make a mistake, we'll need to fix it. We'll learn about some of the first syntax rules and how you can see that we've missed some. Now, although I did say that you are already a C# developer, we didn't actually write any C# yet. So let's do that now. Let's actually write our first C# code together. We had this generated line here that Visual Studio actually generated when

it created our application based on the template. We can actually start by changing that. And let's change this in "Hello everybody." When you make the change, make sure that the double quotes are still surrounding the text that you want to display on the console. That is because this is a string, and that needs to be surrounded by double quotes in C#. That is one of the syntax rules that you'll come to learn in this course. Let's run this again and see that our change has successfully been applied. So we click again here on the Play button. That will start our application. And there we go. We now see, "Hello everybody" being shown here on the console. Let's add some more code here. I want to show another line of text on the console. So I'll start by typing Conso, because I want to write again something to the console, and notice that Visual Studio is now showing me a couple of suggestions. What we see here is IntelliSense. IntelliSense is giving me a couple of suggestions based on what I'm typing. So as soon as I started cons, and probably the o as well, then it knows, it thinks, let's say, that I'm going to write console. And then I do WriteLine here. Again, you see that IntelliSense is showing me, is giving me, let's say, a couple of suggestions. We can do a Write and a WriteLine to the console. WriteLine will automatically move the cursor to the next line. Let's use WriteLine again. And then I want to show another message to the console, and that needs to go again between brackets. What you see here, by the way, is known as IntelliCode. IntelliCode is AI, or artificial intelligence, powered suggestions that Visual Studio now also has, and it will give some interesting suggestions as well. I want to show some text, so that means that, again, I need to enter some double quotes. So I type a double quote, and automatically Visual Studio will type another double quote. Let's enter as the text, "Please enter your name." And then to close things off, I need to end my line here, which is a statement, by the way, in a semicolon. With that in place, we can try running our application again. And as you can see, we now have the extra line of text showing here on the console. Say that we were going a little bit too fast, and I would have omitted this semicolon. What would have happened? You already see that Visual Studio is giving me this red squiggly line here. That is because it sees, it knows, that I've made a mistake here. When I actually try running the application now, it also won't work. Now, in fact, before running the application, let's do something else. I'm going to try building it, which is basically the same, but without executing the application. We're just going to build, we're just going to compile our application. So in here in the Build menu, we can click on Build Solution. And as you can now see, an error was found in our C# code. A semicolon was expected. We can click on this line here, and automatically the focus will jump to this line to the place where the error is found. So I

will enter my semicolon again. We will now have written some text to the console, and it also tried to accept some text from the console. I'm going to create a variable called name, and it's going to be a string. We'll talk later what variables and strings are, but just see them as a place to store the input that I'm going to accept from the user in. And that I'm going to get from the Console.ReadLine. Here again, you see that IntelliCode already think and I might be getting the text from the console. So I can, if the suggestion is right, just do a and automatically have the Console.ReadLine without having to type it myself. Now, when I'm actually using the Console.ReadLine, I'm going to accept input from the console. That text is then going to go into the string name here, and now we can use that to create a greeting. We can write another Console.WriteLine. Let's use Tab again, and I'm going to create my greeting. So I'm going to write here, "Hello," and then a space, and then I want to concatenate Hello with the entered name to create a greeting to the user. So I then do a + and then I use name. Don't forget to close the brackets if you haven't done that, and also don't forget to include the semicolon here. Now we've written really a small application. Let's test it out. Let's run the application once again. So now I see that the console is asking me for my name. So we enter our name here. We close with Enter, and then we get the greeting saying Hello, plus my name, so, Hello Gill here.

Demo: Exploring the Files in a C# Application

We've looked at the generated C# code. But remember that we have talked about this concept of projects. Well, let's explore that in some more depth here. I'm going to explain you the basics about the files that just every C# application will typically have, a project and a solution file. We'll also look at what gets generated when we compile our application, and that will be an executable file. Since that is a real program we can execute, we'll try to run it directly as well. And so far, we have focused solely on the editor and the code editor in Visual Studio. There's another very important window, and it is this one here, the Solution Explorer. The Solution Explorer gives me an overview of the project I'm working on and also the solution, and I'll explain the difference between these two in just a second. And so far we have written our code in this file here, the Program.cs, and that's actually the file that was open here as well. Now, in a real application, you'll have multiple files. You'll have quite a few C# files, of course, but this program file for a specific reason that will become clear later gets chosen by default to start the application. Now when we started Visual Studio, we executed a template, and that generated our project. And the project is this part here, the part that you now see being highlighted. The project

contains references to all the files that make up your application. So if we have multiple code files, which we'll have later, they will be part of the project. If you have images in there, they will also be part of the project. So all the files that are necessary for your application will be part of your project, and it's also your project that is going to be executed. It's really a container, let's say, for all the files that belong to your application to the program that you are creating. Now one level above, you see that solution. And the solution is really a grouping for projects. Now we just have one project, but it is possible to have multiple projects open at the same time in Visual Studio, and they need to be part of a solution. So the function of a solution is really the organization of projects. Now, what we haven't looked at is where are these files? It will become more tangible when we see those files physically exist on disk. We can actually go from Visual Studio and right-click here on the project and ask to open this folder in File Explorer. And this will open a window where you can see the files that make up my application. You can see here again the Program.cs, that's this file, of course, but then we also see that .csproj file. We don't see that in the Solution Explorer. Well, that is because that the project file that is actually open in Visual Studio. If you go one level up, we will also see an .sln file. The .sln file is the solution, and that contains references to the project or projects that it contains. In our case, we just have a project that lives inside of this subfolder here. Another thing that is interesting to know is, of course, where is my application? Say that you want to give the application to someone else to execute. Where is it at this point? Well, we have indeed built an application, an executable file that is located inside of the bin, Debug. net8.0, and in there you'll see an executable file. That is your application containing the code that you have written so far. This you could give to someone else to run on their machine so they can enjoy your application too.

Debugging Our Code

Now that we have written our first piece of C# code together, let us learn how we can debug C# code, a vital part of coding. Writing code is probably the most fun part of development. It's the most creative part of the job. Sadly, we are not perfect, and we will introduce bugs in our code. A large chunk of your development time you'll actually be spending on debugging the code. We might think it works fine, but it doesn't, and that's when we'll need to debug things. Debugging allows us to hook into the running application, so while running, we can poke around in the application, look at the values, and investigate why our code isn't behaving as we had expected it. Luckily, today we have a very powerful

debugger at our disposal. Visual Studio allows us to run and debug our applications with ease. I want to give you an introduction to debugging already here, since it is a task that I said you'll often do. Throughout the course, we'll also need to do this quite a few times. Now, how does debugging then work? Well, great question! I'll be showing you how to do this in the next demo. But in general, debugging works based on breakpoints. The debugger works on the running application. It gives us a peek in the running code. A breakpoint is a place where the code, so the running code, will pause. At that point, we have the ability to look around in the memory of our program. We can look at the state of the values of our application. That will give us the ability to figure out why something isn't behaving as expected. Now, while the code is paused at the breakpoint, we might want to go step-by-step, line by line, and see how things are happening in the code. That, too, is possible. We can work our way through the code and see how things are changing. All of this gives us a lot of insight and understanding of how our code is executing and will help us to fix errors, any bugs we have introduced. Adding a breakpoint is simple, and I will of course show you how to do this in the upcoming demo. In general, we can just click on the line we want the execution to pause on. We will the run our application like we did in the previous demo. As soon as that line is executed as part of the running application, execution will halt, and we can start exploring, well, the internal state of the application. You will soon see that debugging will take up a large part of your time when writing applications. It's part of getting the code in the state it needs to be in. When we run our application like we did in the previous demo, so by clicking on the button with the green arrow, we're, in fact, running the code with the debugger attached. Now, what does that mean exactly? Well, see it as follows. The debugger is sort of hosting the execution of a normal code. When something happens, a bug occurs, or a breakpoint is hit, the debugger will step in and will allow us to inspect the state of the application. When I say we're running the program with a debugger attached, indeed, you can also run the application without the debugger. When we ran the executable before, right from Windows Explorer, we were doing exactly that. There was no safety net around your application, and we basically couldn't see what happened should something go wrong. It is therefore important that we try to fix all bugs before we will ship our program to the customer.

Demo: Debugging in Visual Studio

Okay, time to return to the demo, and this time we'll learn about the debugger. You can see here on line 6 that I've again forgotten a semicolon. That is a syntax error, and that syntax error is called by Visual Studio before we can actually run the application. You cannot run a C# application and still having syntax errors in there. It will not compile, and therefore, it also can't be executed. Now, even if we solve this, there is no guarantee that our application will run fine at this point. Now, to find errors while we're running the application, we should actually use the debugger, which can help us in solving runtime errors. I will already give you a small introduction to the debugger, as you'll need it all the time. The debugger is based on breakpoints. We can actually have our application stop execution at a certain line, and then we can inspect the state of the application. And I can click here in this gray bar here on the left to create a breakpoint. I do that here on line 8 for me. I'm going to click here, and then it's going to highlight my line also in red, as you can see here. Now, I've added a breakpoint. This will stop the execution of my application, and I can inspect the state of the memory at this point. So let's run the application again and see how the debugger works. So I've just started my application as we've done before. The executing application is still asking for my name. I hit Enter, and now I'm actually hitting that breakpoint. The execution is now at this line. This is what just executed, asking me for my name. And now I can also, for example, hover over the name variable here and see that this contains Gill. So using the debugger, I can really look into the memory space used by my application and see values of variables inside of my application. I can now just click Continue, and then the application will just continue running as before. And we indeed see that it executed successfully and also showed the greeting. So let me add now a few extra lines of code, and then I can show you a few more interesting things around the debugger. I've changed a small application a little bit. It is now asking first for the first name and then for the last name. And I'm saving those in two string variables. Then online down here, I'm now writing to the console, not just hello and the name, but hello, the first name, a space, and then the last name. I'm going to put another breakpoint. I can, of course, also add multiple break points, but let's keep things simple for now. Let's run the application again in debug mode by clicking here on the green arrow, and then the breakpoints will be hit again. So now the application asks for my first name. Then I hit the breakpoint. And if I now want to go, let's say, step by step, I can actually do that. We can actually step through our application line by line, letting the application execute lines of code on a step-by-step basis. I can do that by clicking here on this Step Over button, or alternatively use the F10 button. So I'll click here on Step Over, and now we jump to

the next line. If I click that again, then my application will come up again. I need to now enter my last name, and immediately the execution jumps back to Visual Studio, and we can now inspect again the first name, the last name, and so on. So as soon as the application is in break mode, we can step through the code line by line. For now, this will do in terms of knowledge about the debugger. We'll be using it throughout the course. Again, we'll come back to it later in a lot more detail.

Building Applications Using the CLI

So far, we have explored the first option to create C# publications. We have used Visual Studio 2022 so far. That option is available only for Windows. If you don't have access to Visual Studio or you're using Macro Linux, another approach exists, and it is using the CLI in combination with Visual Studio Code, or using only VS Code even. Let us now explore this approach here, too, so that you can enjoy writing C# no matter what platform you're using. Now, I did mention this CLI thing already a few times, so time to explain it in some more depth now. CLI is short for command-line interface, and it's a command-based tool chain to create, build, run, and publish .NET and C# applications. So basically everything we can do with Visual Studio, by creating a new application based on a template, building it, running it, we can also perfectly do so without Visual Studio, but instead using a command-line interface. These tools are installed with the .NET SDK and are cross-platform. This allows us to run the exact same commands on Linux, Windows, and Mac, and compile and run or C# code no matter what platform we are on. Once we have installed the .NET SDK and we open a Command Prompt, we can write the .NET command. That is the entry to work with the .NET CLI. Of course, we need to pass it some parameters to indicate what we want to do. Now, before I forget, SDK stands for software development kit. It's a set of tools including the CLI and libraries, allowing us to indeed create .NET programs. To be clear, the .NET CLI is used, for example, through the Windows Terminal or Command Prompt. Here you can see a screenshot of when I'm using the .NET CLI through the first option that I mentioned, the Windows Terminal. If you're using Mac or Linux, this would work in the same way. Once we have the .NET SDK installed, we can write commands to perform interactions with the .NET CLI. Here you see a typical command to do so. Using the .NET CLI, we can do all common tasks, such as creating a new application, using a template, like we already saw when using Visual Studio. Take a look at the command here. The first part is dotnet, which is often referred to as the dotnet driver, and then I'm using the new command. .NET will always be the first part, since that invokes

.NET CLI. To dotnet, we pass commands. The first one here is new, indicating that you want to create a new application. Other options here include dotnet build or dotnet run. Build would, of course, compile the application whereas run would, well, execute it. So indeed, through this command, we can indicate which action we want .NET to do. Then the next part, console, is an argument. Dotnet new console will create, again, based on a template, a new console application. Console is indeed the name of the type of application we want to create. Along with the .NET SDK, several templates get installed on your machine, and I'll show you this in the next demo. Finally, I'm using -n, which is an option to specify the name of our to be created application, and that would be here, FirstProgram. Things will typically start with a new application, as you just saw using dotnet new. Dotnet build is another command which is used to compile an application. You've seen how we could do this through Visual Studio. Through dotnet build, we can do the same using the CLI. Dotnet run does the same as when we hit the Play button in Visual Studio. It allows us to run our application from source code and it will, in fact, trigger a build if changes were detected. The compiled application will then simply run. Now you may be wondering, hey, this .NET CLI thing is cool, but where do I go to edit my code then? Well, the cool thing is C# files are just text files. You can basically use any editor in combination with the CLI. Visual Studio is an integrated environment. You typically use the code editor in there as well. When using the CLI, you'd typically use more different tools. While any editor will work fine for C# code editing, often developers will use Visual Studio Code, or short, VS Code, a free editor from Microsoft. I will show you this entire tool chain in the next clip.

Demo: Building Applications Using the CLI

All right, I'm going to show you how to work with the CLI-based toolchain. I'm going to show you first what you need to have installed. Then we'll use the CLI to create a new application based on a template again. We'll use our freshly installed VS Code to enter our C# this time, and then we'll return to the CLI to compile and execute our C# program. Now to install all the .NET tooling, you will need to install the SDK. So you can go to the link you see on the screen to go directly to the page where you can download this. On this page, depending on the platform you're on, you can download the SDK installer. I'm, of course, on Windows, so I would need to select this one. If you're on Linux, you can follow the instructions over here. If you're on Mac, you can download the SDK here. In combination with the SDK, many developers use Visual Studio Code. Like I said in the slides, it is possible to use

just any code editor because C# code is just plaintext, but Visual Studio Code and the .NET CLI is an often used combination. You can download Visual Studio Code from the link you see here on screen. And this is, as mentioned, available for other platforms as well. And with the SDK installed, I can now use the .NET CLI. Interacting with the .NET CLI is done through a terminal, a console. I'm using the Windows Terminal here, but you can also use the regular Command Prompt as well. If you're on a different platform, you can use the built-in terminal there, too. Using .NET, which is the .NET driver, I can interact with the .NET SDK. If I just type dotnet and hit Enter, I will get some help on some options that it needs to continue. It can be used to execute applications, but it can also be used to create new applications. And for that, we'll use the dotnet, and we'll pass in new. By passing in the dotnet new command, I can actually create a new project based again on a template. I get a list of some of the installed templates that I have on my machine. We were using the console application before, and so now I will use the exact same, again, a console application, to show you how this approach works based on the CLI. So I'm going to now create a console application. I am already in the correct folder, so that is D:\code, and I type dotnet new, and then the type of the application, so the template I want to execute is console. I also need to pass in an argument that is the name of my application. So I do that using -n. And let's call this application "HelloWorldFromCLI," and I close the quote. And as you can see, the template console app was executed. It has created for me a new application based on that template, and it is located in the decode folder. Let's take a look at what was created. So here's my code folder, and in here we have the HelloWorldFromCLI. So there the .csproj file again created. That's exactly the same as what happened with Visual Studio, and a Program.cs was also created. The .csproj file that was generated here can also be opened by Visual Studio, and the CLI can work with the exact same file. Now, at this point, you may be asking how can I now edit that program? Of course, I can open it the Notepad, but we are going to open the application from Visual Studio Code. So I'm going to copy the path here, and I'm going to go to Visual Studio Code. In Visual Studio Code, when launching it, I can open a folder. I can now navigate to my HelloWorldFromCLI folder and select to open it. Indeed, I am not going to use the project file directly. Visual Studio Code works based on a folder rather than on the project files. As you can now see, we have the Program.cs file and the .csproj file, which are available now in Visual Studio Code. And as we now start typing, we'll also get suggestions from Visual Studio Code like we had in Visual Studio. So the experience to write the C# code is very similar in both Visual Studio and Visual Studio Code. And if I want, I can just paste in, and

I can just bring in, let's say, the exact same code as we had written before in Visual Studio as well. If I now want to execute this, I can do this directly from Visual Studio, but I can also use that from the command line from the CLI again. So if I save this and then I go back to the CLI, can now use the dotnet build command to build our application. I'm going to go into the HelloWorldFromCLI folder, and there I'm going to do a dotnet build, and the application is building. That succeeded. And now I can just execute it. I can now dotnet run, and this will look for the application in the folder that it's in. I can also specify a certain project to execute, but if I don't do that, it will look at the project file that is in there. And now you see that our application is executing as before. So this is the experience using the CLI and Visual Studio Code.

Demo: Using the C# Dev Kit

While the CLI approach works fine, Microsoft has been putting a lot of effort in making VS Code more complete and more integrated. Now since there's no Visual Studio on Mac or Linux, Microsoft is extending VS Code to be a more complete environment for building .NET and C# applications through an extension called the C# Dev Kit that we can install inside VS Code. We don't even need to use the CLI for most tasks, like we just saw. Creating a new project doesn't require the use of dotnet new anymore. We can do this from VS Code's interface, even like in Visual Studio itself. Also running and debug your application will become a lot easier. In the next demo, I'll show you the use of the C# Dev Kit in VS Code. I will show you how we can create a new project directly from VS Code. We will then add some C# code and we'll run and debug from VS Code, too. We're back in VS code here, and let's now use the C# Dev Kit. As mentioned, this approach works on any platform. So if you are on Mac or Linux, this approach will work in the exact same way. Of course, it also works on Windows, which is what I'm using here. The C# Dev kit is an extension. So in the VS Code side menu here, we can go to Extensions, and then we can search for C# Dev Kit. I have it already installed, as you can see, so I can start using it. Just click here to install if you don't have it yet. It will take just a few seconds to finish. The C# Dev Kit will make the experience for a C# developer more integrated inside VS Code. Instead of doing what we just did, so jumping to the CLI, we can now do many more things directly from VS Code, just like in Visual Studio. So, let's now do pretty much the same demo as the previous one. I'll create a small application, a console application again, and we can create a new project in two ways. In VS Code, the, let's say, magical command that opens the Command Palette is

CTRL+SHIFT+P. Through this drop-down, we get access to a very long list of commands we can choose from, and by installing the C# Dev Kit, more commands, useful for what we are doing here, have been added. You can see that there are commands in here that are installed with .NET, and one of them is .NET: New Project. And the alternative way of getting here is by clicking here and then clicking on this Create .NET Project, which essentially does the same. We'll then get a list of possible templates again to start from. I will type here console and select the Console App again. Then we'll need to select a location where we are going to place our code. I'll select this folder here. And then we can enter a name for our project. Now our project and solution are being created. Now, we can see the files again through the Explorer, but the C# Dev Kit also brings with it this Solution Explorer window, which is similar to the one we saw in Visual Studio, and it will give us a project overview in a cleaner way. It is also our entry to building and running our application. If we expand it on the project node, we see again our Program.cs. Our code editing hasn't really changed much from the previous demo, so I won't be doing that here. Now, from the Solution Explorer, we can now build our application by right-clicking on a project and then selecting Build. An integrated output window shows the output of the build, and that seems to have gone okay. Now, before we run our application, we can now also put a breakpoint in our code. Now we can run our application, and for that, again, different options exist. We can use the shortcuts from Visual Studio, and for running that was F5. Alternatively, I can right-clear here on the project and select Debug, and then say Start New Instance. The result will be the same. Our application will be running and should hit our breakpoint. With the application running, we can see that the breakpoint is being hit indeed, just like in Visual Studio. We can click this button here to continue, and we can also see the output. So as you can see, using the C# Dev Kit, the developer experience is now nicely integrated into VS Code. You have now seen the two approaches, so both the Visual Studio-based approach, as well as the CLI and VS Code-based approach. Both work very well. And ultimately, the choice is yours. You can build the exact same things using C# in .NET with either of the options. If you are more a fan of using an integrated environment that allows you to do everything from one place, the Visual Studio-based approach might be better. If you're more a fan of working with the CLI, so command-line-based interfaces, the second approach might be best suited for you. And as we have also seen, using the C# Dev Kit, VS Code offers a more integrated approach, too. Now, in this course, we'll be using Visual Studio for our demos, but you will be able to follow along and do everything I'm doing here using the VS Code or CLI-based approach as well.

Demo: Using the Docs

In this final section of this module, I want to give you some pointers on where to turn for help, namely, the docs. And we'll go straight to a demo. Let me show you what places are great sources of information when learning and using C#. Now, the documentation of C# is an indispensable resource when working with and learning C#. With the link you see here on the slide, you go directly to the C# documentation on the Microsoft Docs site. That gives you a ton of information, not only some other videos to get you started, but also a lot of concepts are explained in depth. Also very interesting is the .NET API browser. We are, in fact, working from a C# code with the .NET class libraries. As the name implies, this allows us to browse through all the .NET code that we can work it from C#. We have, in fact, already worked with the console, so we can type here Console, hit Enter, and then we see the System.Console class. That's the one we used so far. On this page, you will find a lot of information around the console class. Here you see in the examples that we're also using Write, WriteLine, and ReadLine, like we did already. Now, the console can do a lot more, in fact. By just browsing the documentation, I can see, for example, that from code, we can ask the console to do a beep, we can clear the screen, or we can change the color.

Summary

Well, congratulations! You have already created a first small C# program. While it is indeed still very small, it's a start, and you will soon see it grow into a full-blown real application. That's a promise. You're on your way to become a real C# programmer. Let us recap what we have seen in this module to summarize it. You've seen that C# is an object-oriented language and a type-safe language to create .NET applications. Both object-oriented and type-safe might not mean that much to you right now, but they will in the coming modules. C# is a popular language, and it's actively maintained. New versions are being released frequently, and it typically brings some new features for us to use. C# is definitely the most commonly used language to build .NET applications. C# can be used to create all sorts of applications in .NET. Finally, we've created our first program using Visual Studio, the IDE, as well as using the CLI combined with VS Code. Both approaches allow us to create .NET applications in C#. You have the option to choose which approach you want to use going forward. And now that you're on track to become a C# developer and you have everything installed to continue, it is time to understand the C# syntax. That's exactly what we'll do in the next module.

Learning the C# Syntax

Module Introduction

Hey, nice to see you again! Welcome to this module of the C# Fundamentals course here on Pluralsight. In this module, we will learn the basic syntax of the C# language, and we will teach you just enough so that you can feel confident with the basics. And throughout the next modules, you will gradually pick up on many other concepts. Let us dive into this very important module, shall we? Just like when you're learning French, German, or Japanese, when learning C#, you'll need to understand some very basic rules of the language. That's what we'll start this module with. We'll look at things such as statements, comments, and quite a few more. Next, we'll look at the data types we can use, some of the built-in ones first, to be more precise. Along with these built-in data types, we'll use operators, such as arithmetic or assignment operators, and that's what we'll look at next. A bit of a special case already is working with dates and time, and so I have a special topic planned for this module on learning how to work with date and time in your application. We'll come to understand one of the great mysteries of the last module, that type-safety thing that I kept referring to. In relation to that, I will show you how you can convert between types. And we'll close the module looking at how we can use something called implicit typing. A lot of cool topics that will bring you a lot further already in your journey to learn C#.

Understanding the Essential C# Building Blocks

All right, time to start with the very beginning of the C# syntax. Let's learn about some of the essential C# building blocks. It all starts with statements. The statements are the actions that make a program, and they are used for creating actions, such as declaring a variable, calculating the sum of two values, or displaying a string on the console, like we did in the previous module. That's the one you see here on the slide. Statements are executed in a certain order. By default, that would be vertically, first statement, and the second statement, and so on. The order is called the flow of execution. That flow, however, won't always be the same, though, as for example, input of the user might direct the flow to be different. Different statements might be called next, that is. A statement always ends in a semicolon. That's a first and important syntax rule. Here our statement is just one line, but statements

can actually be a series of single line statements. In that case, they will be surrounded by curly braces. We'll see that in the next module. In C#, whitespace and line breaks don't matter. This means in general that you can add a break o whitespace without making any difference in your code. You're free to use this. So what you see here will still work fine, although I do believe it is less readable. I am sure you will agree. Iin C#, we will give a name to, well, quite a lot of things. Using that name, we can identify the item going forward. We've seen in the previous module already the Console.ReadLine, which allows us to capture a piece of text entered by the user. Typically we want to do something with that value. So we'll need to give a name to that captured value. That name is the identifier used to identify this string. Many other items in C# will get an identifier. We'll see soon that we of course need to name variables, like the input here, but also classes, namespaces, and more. There's again some syntax learning here. The identifier we can use can contain letters, digits, and underscores. The first character of the identifier needs to be a letter or an underscore, however, not a digit. Therefore, this second statement, the red one, would be invalid. It wouldn't compile since the name of the identifier starts with a digit. If you want to add in our code some extra information, we can use comments. Using comments is definitely recommended. They will be helpful for others using our code. In C#, there are different ways to add this sort of extra information. In order to make the line to be understood as comment, we need to add a double slash in front of it. From then on, all text on that line won't be treated by the compiler as code. It'll just be ignored that same. Adding comments is definitely a good practice to make your code more readable. If one line of code doesn't suffice for the comment you need to add, which is perfectly fine, you can, of course, add the double slash before every line. Now, while that works, it might be less readable So a second option is available for commenting, namely multi-line comments. In this case, the comment is preceded with /* and followed by */. All content between these is treated as comment and is therefore ignored by the compiler. Just like most programming languages, C# comes with a number of keywords. Those are reserved words with a special meaning and can't be changed. C# comes with more than 70 keywords. I've listed out a few here, including if, void, class, and string. Keywords can typically only be used in certain constructs, and, for example, you can't use them as an identifier. In C#, we'll often declare variables. Variables are storage locations, and they will hold a value. Basically, it's a piece in the memory of your computer containing a certain value. And we declare a variable, we need to give it a name. That will be the identifier for the variable again, and it must be unique for the context where it is used. Using the name

of the variable, we have access to the value it holds. Variables are of a certain type, which can be one of the many types in C#. So a variable can be an an integer, it can contain a string, a date, and so on. Now, this is the first time we encounter types in C#. The type of a variable defines which value it can contain. Remember that we said earlier that C# is a type-safe language. Well, this is related here. When we create a variable of a certain type, C# will guarantee that the value is always of the specified type. More on types very soon. Creating a variable is done in a statement, more specifically, a declaration statement. Let's look at one, shall we? And let us start with the declaration of an integer variable. That's one of the most commonly used types. They contain integer numeric values. Here we are declaring an integer variable of type int, which is the name used to represent the integer type in C#. Next, we give the variable a name, and it is age here. In general, variables will get a name that starts with a lowercase. Lowercase and uppercase, however, in C# aren't different for the compiler, so h lowercase and H uppercase, as you can see here, aren't different variables. Now, while this will work, I don't think this adds the readability of the code and can actually be pretty confusing. Giving variables a good and meaningful name isn't a syntactical requirement, but it's definitely good practice. Another convention in the naming of variables is the use of camelCase. If the identifier should actually be multiple words, use this notation, as you can see here. The first word will always be lowercase, and all subsequent words start with an uppercase. That is camelCase. Now, we did say that variables are storage locations, named locations that can hold a value. But so far we have declared a variable, but we don't have a value in there yet. So, let's do that next. Here we are assigning the value 25 to the age variable. I'm using the equals sign for that, which is an operator, the assignment operator. Using this, the value 25 will be stored in the variable. Notice that this statement too needs to have a semicolon at the end. Now, since H is an integer variable, it can only be assigned integer values. We can't put, for example, a string, so a piece of text, into this variable. This again has to do with that type-safety. C# will only allow us to put a value of the given type in the variable. Now, once we have declared our variable and assigned it a value, we can also use it. Here we are creating a statement, and I am going to write our variable to the console.

Demo: Using the Essential C# Building Blocks

With these essential building blocks covered, let's return to Visual Studio and apply these already. So let us investigate some of the C# basic syntax rules, and I'm going to do that in a new project. So we'll

create a new project. We already know what a project is. Of course, we'll select the Console Application again. And this time I'm going to call the application BethanysPieShopHRM. That is going to be the name of the application that we'll be working on throughout this course. And, of course, we'll select .NET 8 as the version of .NET that we'll build this application with. As mentioned, everything will work fine with all the versions, too. When we write Console.WriteLine, we need to pay attention for a few things. When I want to write something to the console, it needs to go between a pair of brackets. Of course, when you open a bracket, you will also need to close it. When I write between the brackets a string that I want to add to my console, I need to put it between a set of quotes. This is a C# statement, and it needs to be ended with a semicolon. The semicolon is typically put right behind it, but I can, in fact, place it wherever I want. We can add spaces here or we can add tabs here. It doesn't really matter. We can even put a new line here and this will still be okay. New lines and spaces don't have any influence when writing C# code. I think, though, that it is much more readable if we put the semicolon right behind the closing bracket here. If I, for example, forget to close the pair of quotes here, automatically that bracket and that semicolon become part of that text, and therefore, Visual Studio is now flagging an error because the statement is not ended in a real semicolon because the semicolon is not part of the string. And I also see that the colors used by Visual Studio will help us to identify problems in our code. I've added here another line, which is very similar to the one we just had at the top. Now let us start accepting some input. And for that, we'll need to create a variable. I will create a variable again, like we did in the previous module, but I'll do it here again. So I'll accept text input from the console, and I'll capture that in this variable called name. It is of type string, it is text, but we'll talk about types very soon. When creating identifiers, there are a couple of rules that we need to obey. We can, for example, do the following. The identifier can contain letters, numeric values, and underscores. So this is a valid identifier. So is this one because it contains an underscore, and it's valid as well. Our identifies need to be unique, but since C# is case-sensitive, this is also valid because name2 and name_2 are different identifiers. So both can coexist in our C# code. Now, one other rule is the following. We cannot create a variable where the identifier starts with a number. This will be flagged by Visual Studio. We get a red squiggly quickly saying that this is not going to work. Now, since it's not going to compile, I can actually comment it out. Before this line, I can go and I can put two slashes in front of it. This will make the line green in Visual Studio. Again, the colors are helping us. And this is now harmless. It is not going to be compiled anymore, it is now just text, it is

just comment in our code. If you want, you can also put a number of lines in the comment, and you can go here to this Comment out the selected lines button, and then you will comment out all the lines you have selected. That is also accessible via a shortcut called CTRL+K, CTRL+C. This will comment multiple lines in one go. By using CTRL+K, CTRL+U, you will uncomment multiple lines in one go. The comments are used to indicate what the code is going to be doing. This would be a valid comment, explaining what the next line will do. Here you see a multi-line comment which is surrounded with a / and an * and closed with an * and a /. All the text between these two sets is automatically seen as comment.

Working with Built-in Types

I've referred to types in C# already quite a few times, so let's start exploring them here. Once we know exactly what they are, we can start using them, of course. And let's start again with an important statement. C# is a strongly typed language. This means that every variable will have a type specified when using it. And there's a safeguard in place as well. When we write code, the compiler will ensure that the types we use for the variables throughout their lifetime remain the same. The type is used to specify what data we can store in the variable, be that an integer value, a string, or a custom type. Of course, we'll write expressions, as we'll soon learn. These expressions, code really, will often return a value, and that, too, will need to be of a given type. Here the compiler will also check that the types will match. By specifying the data type for their variable, we are specifying the size that C# or in general .NET will reserve in-memory for our variable. But also, the data type that we're using will be a deciding factor on where the data will be stored. Now we will learn soon that depending on the data type, some data is stored on the heap, and some is stored on the stack. The data type will also give us information of the data range that can be stored. Data types will have a minimum and a maximum value. Also, based on the data type, we will be able to perform certain operations on it. I can, for example, add two integers, but I won't be able to just out of the box add two Booleans together. Basically, we can say that there are two groups of data types in C#. We have types that come with C#, let's call them predefined types. These are built into the language and can be used out of the box. We'll start working with these in this module. Next, we have the ability to create new types. Creating and using custom types is super common, and we'll start doing that later in the course. As said, for now, let us work with the built-in data types. C# comes with a number of these. The bool data type is

used for storing Boolean values, so true or false. Int is used for integer numbers. So, whole numbers, and in C#, int is basically int 32, which uses 4 bytes to store the data. The float, double, and decimal data types are used to store floating point numbers, where the decimal should be our choice where we need the highest precision. Then there's the char data type, which is used to store single unicode characters, such as the letter A. These types are built in and are often referred to as primitive data types. Now, we're not done yet. The byte type is used to store small 8-bit in teachers, basically limiting it to a value between 0 and 255. Indeed, only positive values. Otherwise, you will need to look at the sbyte. The short type is used to store 16-bit integers. These are also primitive types. Next is the object type, which is also predefined. We'll use the object type a lot when we start building our own types. Finally, another built-in type is the string type. It's a bit of a special case, and we'll talk a lot more about it later. Object and string are built in, but they aren't primitive types. Now that you have a high-level understanding of what data types are and how they relate to making C# a strongly typed language, let's start using these built-in data types. When we decide we want to store an integer value, we can create an int variable and give it a name. Here I'm doing exactly that by specifying int a = 2. A new variable of integer data type is created here containing the value 2, and it's named a. We can also write an expression on the right-hand side, as we're doing here. An expression is a piece of code that will evaluate to a value. Here, I specify that the result of the expression a + 3, which is again an integer, should be stored in a new variable of type int, and it's called b. This line of code also shows us that the expression returns a certain value. It basically decides what will be returned, and this needs to be captured on the left-hand side. The type returned by the expression should be the same as the variable on the left-hand side of the equal sign. Say we want to create a Boolean value called c. Well, then we would create a bool c and set it to either true or false. Here I'm setting it to true. In C#, we need to include the data type upon declaration. Once defined, the variable can't change its type later on. So moving from one type to another simply is not possible. Since C# is strongly typed, it will guard at all times that types are respected. And this is verified at compile time, even. Take a look at the snippet here. We are first creating an integer variable c, and I set it to 3. In the next line I'm then trying to set it to true, which the compiler simply will not allow. Why? Well, good question. The answer is true, the right-hand side is a Boolean value, and that cannot be stored in c, which was declared as an int, and it will stay an int forever. Having these checks at compile time will save you from a lot of

runtime issues which occur in other languages that do not apply type-safety. Did I mention already that I love type-safety in C#?

Demo: Working with Primitive Types

Let us return to Visual Studio and learn a lot more about working with primitive types. When we are going to create variables, we always need to start with the type because the type is going to indicate what values can be stored in the variable and also what the range of variables will be that we can store. Let me show you. Let us say that we wanted to store a value of type integer. In that case, I will write int, so that is the type that's the integer type, and it is, in fact, a keyword. As you can see, Visual Studio highlights it in blue. We've already talked about the rules regarding naming, so let us call this variable monthlyWage. I'm using camelCasing here. CamelCasing uses a lowercase for the first word and uppercase to indicate the start of any next word. It's important that we give our variables good names that cover what they are doing. Here I'm indicating that using monthly wage. I will end this statement with a semicolon. Now I have a valid statement in C#. Now, of course, typically I want to assign a value to that. I can if I do that upon declaration already. I can go back here and type an assignment operator. I have now used the single equal sign, and I can then specify a value. This value, of course, needs to be of type int, of type integer, that is, because the type of the variable is declared int. We can also declare multiple variables in one go. Say that we have months, and that would be 12, that's another variable. I can now write a comma, and then I'll specify the name of another variable, bonus, and I say that that would be 1000. Now I have declared two variables of type int in one line, and I've also in one go given them a value. Now, of course, we can create variables of other types as well. Say that I want to create a Boolean. I create a Boolean variable using the bool keyword. Then I specify the identifier, the name, and that's going to be isActive that can accept true or false. IntelliCode is already suggesting false, but I want to go with true here. And we can also use the double keyword. Double allows us to capture floating point values. So, for example, if I want to give a rating, that can be a floating point number because it's not a whole number like integer. For example, let's say that that would be 99.25, a very good rating for the employee. Now, C# is going to check that the value that we are assigning matches the type that we have defined, but it's not only doing a type check, it's also going to check that the value is not falling outside of the range. Say that we have another variable of type byte this time. Byte can accept numbers between 0 and 255. My variable is

called numberOfEmployees, and it is initially set to 125. But say that Bethany's Pie Shop is growing and it's growing a lot. At some point, we might want to assign 300. We've done a lot of hiring, and now we have more than 255 employees that we cannot store in a byte because bytes can only contain values between 0 and 255. And as you can see, the compiler is flagging an error on that saying that it cannot be converted to a byte. So let's comment this out because this is not going to work. And so far I have defined the variables and assigned them a value in one go as well. Of course, that is not always required. I can create a variable and only assign it a value later in the application. And, of course, later on, we can also change that again somewhere else in the app occasion. Now, while doing so, what we cannot do is of course changing the type. Say that we, for example, want to say that a monthly wage now all of a sudden becomes true. That won't work because the type is integer, and I cannot assign true to integer. This is type-safety. Once a variable is declared, it can never change its type.

Demo: Using Constant Values

So far, the variables we have worked with are indeed what the name gives away. There are variables. We can change the value by using an expression or an assignment. There are times, however, that we may want to work with the value that cannot be changed. Never, throughout the execution. We can avoid assigning a new value. But while the intention is good, it might be that somewhere in the code, this happens anyway. We can call in the help from C# by declaring the variable as constant. For that, C# has a built-in keyword, namely, const. You can see that being used here in this code snippet. And we declare interestRate. Using const decimal, its value cannot be changed. Trying to do so will result in a compile time error. Indeed, the compiler will check if we aren't in our code, assigning a new value to a const. If so, we'll get an error that we need to fix before we can run our application again. Let me show you the use of constants in another demo. If you want to create a constant in our code, we need to indicate that to C#. Say that I want to have the interest rate in my application. An interest rate will probably be a floating point number, so I'll use a double again, and I'll set it to 0.07. And if I don't want this value to ever be changed in the application, I can make it const. I do that by using the const keyword that I use before the type, so before double here. Now this value is const. That means that I can't change it again. Now adding code can I now say that interest rate is, for example, increasing to 0.08. If I try doing that, the compiler will already flag an error, as we can see here. We again get a red squiggly.

Getting to Know Strings

So far, we have worked with numeric and Boolean values. Apart from using it in combination with the console, we haven't worked with text really. C# of course has support to work with text in the form of strings. Strings are a special case. Not only can you do so much with them, they also behave a bit differently. That is why we will devote an entire module to them later in the course. But let's already take a look at strings here. A string in C# consists out of a series of characters, chars really. Think of it as a list of characters, really. Together they represent a piece of text. This can be a short text, even just a space or an empty string, or contain an entire book. When creating a string, as you'll see in just a minute, the actual text is surrounded with double quotes. This is different from a char, which had single quotes. When creating a string in C#, the type that we use is the string, and you'll typically use the lowercase keyword as type to create a string. Here you can see how I'm declaring a couple of strings. In the first time, I have used the string type, and I've created a string named as 1, and I have initialized it with probably the most commonly used phrase in computer history, "Hello world." Notice that this string is surrounded with double quotes, which indicate the start and the end of the string. And a string can also be empty. We could create a string using a pair of double quotes with no contents between them. That's an empty string. But as said, the string contains many helpers to enable us to work easily with strings. And one of these is the string.Empty. This will create a new empty string and assign it to s2..

Demo: Creating Strings

Time for our first demo with strings. Let's create a few and perform a few operations with them. Strings are declared by using the string keywords. Then we are going to declare again the identifier. Let's call it firstName. And then I'm going to declare the initial value for the string. As you already know, strings need to go between a pair of quotes. In this statement, I've now declared a string. The variable is called firstName, and the value is Bethany. I can, of course, declare multiple strings, as you see here. Now, so far, we've defined a value, that's what we had here, but I can also declare an empty string. By just not putting any value between the quotes, this becomes an empty string. Of course, the value of the string can also come from somewhere. We've already seen this code where we're actually getting the string from the console. We are reading the text from the console up until the user hits Enter, and the value entered by the user is assigned to this string variable name. As said, strings are extremely

important in C#, and there is so much more we can do with them. They also behave differently compared to regular variables. That's why we will come back to strings, They will get their own module, even.

C# Operators

C# comes with quite a lot of operators built into the language, and most of them are supported on the built-in types we just looked at. Now, just declaring values will not get us very far. We need to perform actions with our data. Expressions are basically what you can find on the right-hand side of the equal sign. They can be very simple, just assigning a value. But of course, they can be much more complex, too. Very often in these expressions, we use operators. C# comes with a large set of operators, and you already see a few in action here. In the highlighted line here in the snippet, you can see that I'm adding a to the product of b and c. As said, there are quite a few operators available in C#, and it's not my goal to explain them all in this course, but we'll take a look at the selection, and you'll see them being used throughout the course. First, we have the arithmetic operators used indeed to perform calculations. Equality operators are used to compare values, and logical operators will perform logical operations to be Boolean values. Assignment operators are used to assign value, as you may have guessed from the name. Here you can see a few arithmetic operators, and I'm sure you'll know most of them from math class. You can see the plus for addition, the minus for subtraction, the asterisk for multiplication, and a forward slash for division. The double plus is worth mentioning here. It is used to increment the value of the operant with 1. If I want to add a number, say 1, to the current value of a variable here, month, we can do this as follows. We say month is equal to month plus 1. This will increase the value of month to be indeed 4. Now, although this syntax is perfectly valid, a shorthand is also available, which you see here. This is used very frequently in C#, and this operator is often referred to as the compound assignment operator. It also works with the minus, the forward slash, and the asterisk. Note that most of the basic arithmetic operators will work on most built-in types, but not all. If you want to concatenate two strings, you can actually use the plus sign as we'll see later. However, multiplying two strings simply doesn't work, which was to be expected. We'll learn a lot more about strings later on, as said.

Demo: Using Operators in C#

Let us return to Visual Studio and start using these operators. Now, while we're in Visual Studio, I will also show you that based on the given data type, a default value will be assigned by C#. So let's see how we can use the arithmetic operators in C#. I'm going to bring in a couple of extra variables that we're going to be using in this demo. First, let's bring in a double to contain the ratePerHour for an employee, and it will be 12.34, and I also will bring in an integer for the number of hours worked. And I think a month is about 160, 165 hours. So, let's take 165 as our value for the number of hours worked. So I'm now going to calculate the wage. The value of the wage for this month, I'm going to capture in a double variable called currentMonthWage, and its value is going to come from an expression. In fact, a calculation is going to be the ratePerHour times the numberOfHoursWorked. And you see that IntelliCode is actually quite clever. Most of its suggestions are spot on. So now we have assigned the value of this expression to currentMonthWage. I can now add currentMonthWage to the console. Let's try that out. And you can see that the value is over here 2036.1. On my machine, you can see that it's using a comma as the decimal separator. That is because I'm using Belgian notation. It might be different for you based on your local settings. Now, C# is pretty clever and it knows about the arithmetic operations. Say that I want to bring in the bonus. We had to find the bonus over here earlier to be 1000. I can say that I want to increase the wage here with the bonus. And if you run that again, we'll see that it's 1000 extra now, indeed, as you can see here. So, C# knows about the arithmetic operations and how they should be executed. I can also increase the rate per hour. Say that we want to increase the rate per hour by 3. For that, we can use the compound operator. I'm going to use the compound operator, which is += and then 3. This increases the current value of ratePerHour, so this one with 3. And it's basically the same as saying that the ratePerHour is equal to the current ratePerHour + 3. That validated value is indeed increased. Indeed, it is now 15.34 when it was originally 12.34. Now, these are some of the arithmetic operators. We can also use the equality operator, and I'm going to use something that we haven't covered yet, and it is the if statement. The if statement allows me to check if something is correct, if something is true. I want to check if the currentMonthWage is higher, for example, than 2000. And if that is the case, I want to write something to the console, as you can see here. As it turns out, the currentMonthWage is indeed more than 2000, so indeed see that this line is being written to the console. So here we are using one of the built-in equality operators that come with C#. Say that we currently have 15 employees working for us. Sadly, someone leaves the company, someone leaves Bethany's Pie Shop. We can decrease the number of

employees with one easily by using a double minus sign. This is a shortcut saying that you want to take one from numberOfEmployees. If you want to increase the value of numberOfEmployees, we can use the double plus sign. Let's validate that this is indeed correct. Indeed, now we just have 14 employees still working for us. Now instead of writing this to the console, I want to use the debugger once. I'm going to write one more line here because I need to be able to put a breakpoint on an executable statement. I'm going to put a breakpoint here, and here. Now let's run the application in debug mode. By the way, there's a shortcut for that. You don't always have to click this button. Hit a 5, and you will also launch the application in debug mode. Currently, the number of employees is 15. If I now do an F10 to move to the next statement, I will see indeed that numberOfEmployees was decreased with 1. Let me show you one more thing. I want to show you that when I don't assign a value to a variable, it'll get a default value. So let us write the following, bool a and int b. And let's check what values they get automatically. And let's use the debugger again for that. So, a, which is a Boolean, automatically will get a value of false, and b, which is an integer, will get the value of 0. Each of these primitive types will get a default value automatically assigned if we don't assign them a value.

Members On Primitive Types

We've now seen how we can perform very simple operations, but there's so much more we can do with built-in types. They're actually very clever. When we, for example, want to work with the maximum value that an integer can contain, we can use a built-in member. A member, you may say? What's that all about? Well, members is the group name to point to data and behavior on a type. And by putting a dot behind the name, so here, int, we can access the members. When we do this on the int, we'll get, for example, access to int.MaxValue and also int.MinValue. This will assign to our variable the max or minimum value that the integer type can contain. And remember that the data type also defines what range is possible for a value, and that's what we see happening here. Of course, other types have similar members, like double, for example. Now, we haven't worked a lot with it yet, but char is a very interesting type, too, used as said to store single unicode characters. On the char type, there are quite a few members defined. Say that we want to accept input in our application, and we want to check what the user has typed. We can check if the given value is a whitespace using char.IsWhiteSpace, passing the character variable, so myChar. Note that to the isWhiteSpace member, which is what is

known as a method, I'm passing the char variable myChar, which in turn contains here the value a. Similarly, we can check if the value is a digit or a punctuation character.

Demo: Working with Members on Primitive Types

I will now jump back to Visual Studio and show you some of the built-in members for the int and char type. If you want to explore what built-in members we have, or, for example, the int that is behind the scenes in Int32, then we can go to the documentation. And in there, we'll see the Int32 type, and we will see that it has a MaxValue and a MinValue, and it also has a couple of methods. We'll talk about what fields and what methods are in a lot of detail, but these are the functionalities offered by the type which are built by Microsoft, and so we can use them in our applications as well. If we head back to Visual Studio and we do int., we can indeed see what members are available on the int type. We see indeed the MaxValue and MinValue, and we can do an int.MaxValue to grab that MaxValue. That is basically the maximum value that can be contained in an integer variable. In a very similar way we can get the minimum value. Let's take a look at these values with the debugger attached. So I'll put a breakpoint here again, and we'll run it with the debugger. So indeed, we see the MaxValue and also the MinValue. Now, we haven't worked with the char type a lot. Let's do that now, and let's see what this type offers as built-in functionality. The char can be used to contain a character. And while behind the scenes, that is, in fact, a unicode value that corresponds with a character. So let's use the char type now. Say that I want to capture the user's selection in a char this time. So not in a string or in integer value, but instead I'll use the char. So I'll create a char and I'll call the variable userSelection. And I'll set it to, assume that user has typed a, then I'll set it to a, and now notice that there is a difference between chars and strings. Chars actually require single quotes instead of double quotes, which we had with strings. Now, with that userSelection ready in a char, we can again use some of the built-in members on the char type. Say that I want to get the upper version, so the uppercase version, I should say, of userSelection. I'm going to use in that case char., and then I'm going to use toToUpper, and I need to pass in this case my char, and that would be userSelection. And with a lowercase a corresponds an uppercase A, of course. So that character will now be stored in this upper version. I'll run it in just a minute. There are some other interesting members available on char. We can, for example, ask if the userSelection is a digit or if it is a letter. Let's test this out in the debugger. Upper version of lowercase a is, of course, uppercase A, but notice that it also says 65. That is the unicode

value of uppercase A. UserSelection wasn't a, so isDigit is of course going to be false, but isLetter is going to be true. So you see indeed that char comes with a number of interesting built-in members, built-in functionality, let's say, that we can use in our application. Now, the char has some other interesting members available. Do check them out at your own pace.

Using Date and Time in C#

There is another special data type we need to discuss, and that is the DateTime and its sibling, TimeSpan. Working with dates is very common in C# applications. Because they are so common, C# comes with a few built-in types to work with them, and that type is the DateTime type. A DateTime is used to represent a certain date, a certain time, or a combination. So it could be that we create a DateTime for March 28th, 2025 8:00 pm in the evening. Another related type is the TimeSpan. This is also used for time-related work, but it's used to represent a period in time, so, for example, 3 hours. Let's see how we can create a DateTime. A DateTime is internally what is known as a structure, but let's not worry about that yet. It does, however, require that we create or instantiate a DateTime in a different way using something that is known as a constructor. Doing so requires the use of the keyword new, and that's what you see here. I'm creating a new DateTime instance, and I'm passing the date value, 2025 being the year, 03 being the month, and 28 being the day. Now, don't worry about this syntax just yet. Just focus on the time for now. All the rest will become clear soon enough. A type like DateTime has a lot of functionalities, and they are referred to as members. For example, DateTime today returns, well, the current date. To a date, we can also, for example, add two days, and a new date will be returned. DayOfWeek can be used to figure out what day it is, so Monday, Tuesday, and it's a useful number for me, as I often tend to forget what day in the week we had. Anyway, here's another one. It is possible to check if a specific date falls in the daylight savings period or not.

Demo: Working with DateTime

Let's return to Visual Studio and start working with date and time using the DateTime data type. Let us now learn how to work with date and time in a C# application. For that, we will use the DateTime. That is again built into C#, but as you notice, it has a different color than a keyword. This is not a keyword. This is a type that is built into C#, built into .NET, in fact, I should say, but we can use it from C#. Now

when we are going to represent a date or a DateTime, I should say, in a C# application, we should use what is known as the constructor to create it. Don't worry about constructors for now. It is just a different way to create a new instance in this case over a DateTime. When I want to represent the date that we hired someone at Bethany's Pie Shop, we can create a DateTime hireDate. Now, notice that the way that we are going to create that DateTime is not by just specifying the DateTime as a string or something. We are going to use, as mentioned, a constructor. And the constructor will accept a number of parameters that allows us to define the DateTime. The parameters represent the part of the day, the part of the year, part of the month, and so on. Say that we want to create a new date being March 28, 2022, and we signed a contract with the new hire at half past 2:00. Of course, we don't hire them to the second, so I think this will be okay. So this is now a new DateTime that is being represented in C#. If I just write the higher date to the console, let's take a look what it looks like. We will indeed see that this looks like 28th of March 2022. Again, this is using the date representation that I'm using on my machine, just first, the date, then the month, then the year. Let's create another date. I'm going to create a DateTime exitDate this time, and I'm going to specify in this case only the date part, not the time part. If you look in this tool tip, we can scroll through a number of options to create a DateTime which allow me to specify in a different way how I want to define my DateTime. And there is an option to create a DateTime using just the year, month, and date. That's exactly what I want. So, set this too far in the future. There we go. So now I've defined another DateTime, which is also a valid date, but in the future. And I'm not specifying a time part in this case. Now, the DateTime in .NET is actually pretty clever, and it will also know exactly if a date is representable if it's, in other words, a correct date and existing date. Say that I specify the following. Now, I have defined InvalidDate to be of the 15th month. Now, I've seen years that have been longer than others, but I've never seen one of 15 months. So let's see what happens if we run the application now. As you can see, we are getting an exception. An error is being thrown by the application, which specifies that the year, month, and day parameters describe an unrepresentative DateTime. In other words, the DateTime that we have created is invalid. This is a runtime exception. It's not a compile time exception. You only get the exception when you're executing the application. We can also do calculations with the DateTime. Say that we have a startDate, and the startDate is, in fact, 15 days later than the hireDate. We can, in fact, do the higher date at 15 days. As you can see, there are quite a few options available, these are methods again, that allow us to do calculations using dates. I can specify that I want to add 15 days to

the hireDate, and that will be the startDate of the employee. Let's run this again in the console to see the result. So the hireDate was March 28th, so the start date will be April 12th. Just like we had with the char, also on the DateTime, we have a lot of options available, again, built-in members that we are getting on the DateTime type. For example, if we want to get the currentDate, we can just do DateTime.Now. We can also check if that currentDate is actually in the Daylight Savings Time or not. Let's run this again in the debugger and see the result. CurrentDate is January 3rd, and we're currently not in Daylight Savings Time. Next to the DateTime, we also have the TimeSpan type, and it can be used to work with a certain duration. Say that we have a work day of 8 hours and 35 minutes. If you want to do a calculation to see when an employee can start and stop their working day, we can do that as follows. Say that the employee is going to start working right now, DateTime.Now. If the working day is a duration, we can use a TimeSpan for that. And I'm again going to use a constructor, so the way to create it is very similar to what we did to the DateTime. But now I'm going to specify a duration for hours, minutes, and seconds. So let's say that the working day is 8 hours and 35 minutes and 0 seconds. Now, based on these two, I can calculate what would be the end hour for the employee. I'm going to use startHour.Add, passing in the TimeSpan. Notice that this Add expects a TimeSpan, whereas the AddDays expected a double value of a number of days to be added. So there's really a difference between these two. Let's write these two to the console and let's run the application again. These values are the ones that we are using. It is now January 3rd, and this is the startHour, and so this would be the endHour. Now, we just wrote the startHour and endHour directly to the console. There are, in fact, a lot of options to format the date before using it as a string. Let me show you a couple here. On startHour, I'm going to use the ToLongDateString, and you'll see the result in just a second. On the endHour, I'm going to use a ToShortTimeString. ToShortTimeString will of course just use the time part. Let's see the result. So, ToLongDate used this date here, Monday, January 3rd, and ToShortTime resulted in this 22 hours, 70 minutes being shown.

Converting Between Types

And as in the last part, talk a bit about how we can convert between different types in C#. And you may be thinking, hey, Gill, you said earlier that it is not possible to change a variable type. Why are you bringing this up again now? Well, allow me to explain. Indeed, what you see here on the slide won't work. This will actually result in a compile time error. First, we declare an integer variable and we

assign it to value 3. Then in the next line, I assign a string to our variable. Errors, errors, everywhere! Well, at least on this line here. Although we can't change the type of a variable once it is declared, it is possible to convert its value from one type to another. And this value will then be assigned to a new variable of the target type. For this we'll use conversions, and there are, in fact, a few options to do this. We can do an implicit convert, an explicit convert, also known as a cast, or use a helper. Let's take a look. First, let me show you an implicit conversion, or an implicit cast. Say that we create an integer variable and we give it a value. If we now create another variable of type long and assign a to it, basically what we're doing is taking the value of a and passing it to our new variable. We are indeed working with the value. Now, C# will not complain about this, since the data conversion, well, just works. Longs can contain larger numbers than integers. And so we're sure that nothing will get lost. No special syntax is required to do this. Secondly, let's try doing the same for a double to an integer. If we try to do this in the same way, well, the C# compiler would actually complain, since now data could actually get lost. Indeed, a double can contain more than an integer. Therefore, we need to specify to C# that we actually know what we're doing, and we need to specify the type we want the value of d to be converted into. This is done by putting the target type between brackets before d here. You'll see that not all conversions will basically work this way, and so they can actually result in a runtime exception, causing your application to crash.

Demo: Converting Between Types

In the next demo, let me show you how to convert between types. We have worked with the numberOfHoursWorked variable before, and it was of type int, and we set it to 165. Next to the int type to represent integer numbers, we can also use the long type. Long can actually contain much larger integer values. And so that means that every integer can, in fact, be wrapped inside of a long variable. So if I say that long veryLongMonth is equal to the numberOfHoursWorked, there is no way it's quickly showing as you can see because an integer can also represent integer numbers, but long can actually present larger integer numbers. So we're good here. Now say that you want to do the following. I have a value d and it's of type double. Doubles are created to contain floating point numbers. If I now create a new integer variable, let's call it x, and set it equal to d, we'll gather it quickly again. Visual Studio, in fact, the C# compiler is complaining it doesn't want to allow this conversion to happen, automatically at least, because it will say that possibly data loss could happen if you put the value of d into our variable

x. It can contain much less than our variable d, so possible data loss could happen. Only if we explicitly allow this conversion to happen, you basically force the shop in doing the conversion, will it actually accept it. That's what you see here. Here I'm using a cast by specifying the target type int in between brackets in front of the variable d. And now it will go through. There could still be data loss, but now it's basically our own fault. The same would happen if we say that we create another integer variable, say intVeryLongMonth, and we set that equal to a veryLongMonth, then, too, possible data loss could happen. And again, the conversion wouldn't be done automatically. Only if we explicitly indicate that we want the conversion to happen will it be done by C#. Next to converting, there's also something called parsing. Parsing allows us to take a string value and convert it, or should I say parse it, into another type. It's a very common thing to do, but it's more related to strings. I will thus take a look at it in a later module.

Implicit Typing

Now, before we close this module, I want to briefly discuss one more thing related to types, and that is implicit typing. I've now already mentioned a few times that once the type is set for arrival, it's done, it's set. And so far, we have been explicit about this, meaning that we included the type name. so int, bool, in the declaration. That is the normal way of doing things. Now since C# 3, so already quite a few versions, C# supports what is known as implicit typing. As the name gives away, implicit typing allows us to ask C# to figure out the type itself based on what is on the right-hand side of the equal sign. We use for that the var keyword. Basically saying, hey C#, here is a variable, please figure out the type yourself. Take a look here. I have here var a = 123. 1, 2, 3, 123, that is, is an integer value, and thus C# will create a to be of type integer. If we do var b = true, C# will understand that true is a Boolean value, and thus b will be of type bool. You may be thinking, is C# then allowing us to omit specifying the type? Well, that would be a great question. And the answer is no, it's not. It is still creating the variable with the data type it finds. The data type is decided on on compile time, so it's known what our a, b, and c will be before the application runs. Just like before, we also can't change the type of a later on. It is still set. Using var just means that the type is inferred. C# will look at the expression on the right-hand side. And thus based on the result of that, it will know what the type of the variable will be. You can pretty much everywhere where we have used so far our explicit typing use implicit typing, although there are some exemptions. However, it can lower the readability of your code, since it is

harder to understand what type of variable will be. However, and this goes beyond the scope of this course, there are places in C# where implicit typing is the only way forward. When you will learn about the LINQ later in the C# path, you will see that in quite a few cases the use of var is required. Now, since the type of the variable on the left-hand side is inferred of what is on the right-hand side, well, if there is no right-hand side, that will be quite difficult. Therefore, when using var, we can't write what you see here. This will not compile. The compiler can't infer what type employeeAge will be. And that is needed to keep the type- safety in place.

Demo: Using var

Let us return one more time to Visual Studio and see how we can use a var. Now, so far, I've been explicit in the types I use, meaning that on the left-hand side, I've used always the name of the type I want my variable to be. But I can change that. I can, in fact, replace the int here for monthlyWage. That's one of the variables we declared earlier in this module. I can replace int here with var. Now you're basically saying I don't specify the type myself. Instead, I'll let C# decide on it, and it will do that based on what can be found on the right-hand side. That is an integer value. Hence, monthlyWage, if we hover over it, will still be an integer variable. If you do the same for this Boolean isActive, and we also replace Boolean here with var, isActive still will be a Boolean based on what can be found on the right-hand side of the assignment operator. The type is inferred automatically. And the same goes here for double. Let's replace this with var as well. That is a double, so rating will also be a double. Interestingly, remember this line here? That was faulting because of the numberOfEmployees being 300. That was too large to put in a byte. Let us actually do this, and then we'll see that C# decides numberOfEmployees should, in fact, be of type int, and then it will work fine. We can use this with pretty much all types that we have. Let's go back to our DateTime. This was the hireDate. Plus the DateTime, I can also use var here, and that hireDate will still be a DateTime because of what you had on the right-hand side.

Summary

We've reached the end of this module. Now, what a ride it has been! And to summarize what we have seen in this module, probably the single most important lesson here is that C# is a strongly typed

language. C# will guard that types are respected. Some other languages don't do this. It's definitely a plus when using C#. We have in this module looked mostly at using built-in data types, most of them primitive types. For these, C# comes with a number of keywords, such as int or bool. We then also looked at different options we have to convert from one type to another, and we've also seen the different options that exist for this. In the next module, we'll look at two important building blocks for C#, namely the ability to execute code based on a decision made in code, and loops for iterations.

Adding Decision and Iteration Statements in C#

Module Introduction

Hi there, and welcome to another module of pure C# fun! So far, the C# statements that we have looked at were executed in sequence. Now, while that is often what is needed based on certain conditions, different branches of code might need to be skipped or repeated. With the ability to let C# make a decision, we can do the first, and through iteration statements, we can do the latter. In this module, we'll learn about, oh, so important if, while, and for keywords, and quite a few more, actually. Let's take a look at what we are going to be learning in this module. We'll start with understanding a bit more about Boolean values. We've touched on these already briefly, but we haven't really worked with them. That will change in the first item here. In the same area, so working with conditions, we'll learn about the if statement and its different options. Next to if, there's another option to make decisions, and it is using the switch. I'll show you how this works and when to use either the if or the switch. And last, but definitely not least, we'll learn about iterations. In this topic, we will cover the different options we have to create loops, including while, do while, and for. Lots of interesting topics, I reckon. Let's dive in!

Working with Boolean Values

But first things first, before we can work our way through working with conditions, we'll need some more knowledge about Boolean values and how they work in C#. Let's do that here. We have in the previous module when looking at primitive types touched on Boolean values. I'm sure you remember. A Boolean value can either be true or false. And the built-in primitive type to store these is the bool. There is a backing type in C# called Boolean, but most of the time we'll use the bool keyword to create a Boolean variable. C# has quite a few Boolean or logical operators on board, of which we will explore the most commonly used ones in just a minute. You may know logical operators already. They include the negation and the and operator. Here you see something that we've touched on in the previous module. We're simply declaring a Boolean variable using the bool keyword, and its value is set to true. When we then, for example, want to show the value of the variable on the console, we can do so using Console.WriteLine, and we pass in our variable. There's really not that much to learn about Boolean variables. They can, however, become quite interesting to base decisions on. C# comes with relational operators, and these can be used to verify if two operates are equal, greater than, and so on. Here in the table, we can see a list of these. The first one is the equal to operator, which allows us to verify if the two operant are equal. Know that this is indeed the way to verify equality. A single equals sign is used as an assignment operator. So to assign a value to a variable, this operator will return true if the two operants are equal; otherwise, it will return false. If you want to check if our two operants are not equal, we can do so using the not equal to operator, which will return true if a and b, so our operants, are not equal. The greater then and the less than operators are shown next. If the value of the variable a is greater than 10, in this case, this will return true. These can be combined with an equal sign, as you can see in the last line of the table. Here in this case, this condition will return true if a is less than or equal to 5. Here are some examples of these relational operators. The first line is checking if the value of the age variable is equal to 45, and it will return true if that is indeed its value. Make sure that if you want to perform a check here that you use the double equal sign; otherwise, you're simply assigning 45 to the age variable. The second line is checking if age is not equal to 0, and if that is not the case, it will return true. C# also comes with Boolean logical operators, also known as conditional logical operators. Namely, the logical and, and the or. Both will work with Boolean expressions, and they will return a Boolean value. Let's look at the logical and first, by means of an example. Assume that we want to verify in code that the age of an employee is valid. We assume that people will be over 18, but they shouldn't be older than 65 because then they should, in

fact, be enjoying their retirement. That we can do with the code you see here. We have two Boolean expressions, age >= to 18, and age >= than or equal to 65. In between these two Boolean expressions, I have now placed the &&, which says that if both expressions return true, the statement will also return true. Otherwise, if one or both evaluate to false, we will get back false. The second logical operator is the logical or represented as a double pipe, as you can see here. This one will return true already if either one of the expressions is true. It'll only return false. both evaluate to false

Demo: Working with relational Operators

Let us in the first demo of this module go and use these operators. We'll use the relational operators and the logical and and or. Starting with this module, you will find all the code that I'm typing and also sometimes copy pasting into the editor in a snippet file. Go to the assets that come with the downloads of the course to find the snippets. And now we're at m4, so if you look in the Assets folder, you'll find the folder m4, and in there you'll find the snippets for this module. So now we are going to start working with Boolean types. We're still in the BethanysPieShopHRM application, so the same application, the same project, that is, that we had also in the previous module, but I've removed the code because that was just some testing code. So let us now start working with Boolean types. I'm going to define a new variable called age, that's of type integer, and I set it to 23. If you want to check what the value of age is, if it is equal to 23, I need to use a double equal sign, and that value of that equality check will be a Boolean. So say that I have a variable bool a, and I can set that equal to, so that is the assignment operator that I'm using here, I can set that equal to the following expression. The result of age being equal to 23 or not. Here I'm using the relational operator. This will return true if age is effectively 23 and false if age is not equal to 23. Now, since age is defined to be 23, I think the result will be true, but let's verify. Indeed, age is equal to 23. In a very similar way, we can use b, and b can be assigned to the result of age being greater than 23. If you add this to the console, I think you already know that the result will be false because age was 23 and it was not including because I didn't put the equal sign. If I would have put greater than or equals, then it would have been also true that we got back. Now, say that I want to check if age is between 18 and 65. I say and, so it has to be above 18 and below 65. Then I can use the Boolean logical operator, the and operator, in that case, because I want to only return true if both are true. So what I would do then is I would say bool c is equal to the result of age being greater than or equal to 18. And so I use a && here, age also needs to

be less than or equal to 65. And the result of that is, of course, true because age was set to 23. So remember that the && or the logical and only returns true if both operants result in true. I've pasted in a little bit of code, which you can find in the snippets. I've now used age1 = 16 and ag2 to be 64. I'm again using two operants, ag1 greater than 18 and age2 less than or equal to 65. For age1, that's going to result in false, of course. That is, that I'm using logical and and logical or over here. Here you can see the result. The first line resulted in a false because this operant is false and the entire expression will result in false. If we use the logical or, hence this results in true.

Making Decisions with the if Statement

Now that we know more about the Boolean operators, we'll use this knowledge to work with conditions in combination with the if statements, one of the most important tools in your C# toolbox. Although we have written already some C# code in this course, it's been simple to follow. Basically, the program flow, so the flow of execution of the different statements that we have created, has so far been sequential. First, the first statement executes. Then, the second one, and then the next one. Now, in such a case, every execution of the application will probably be the same as well. But in many cases, that is not what's going to happen. Perhaps we need to ask the user for a value. Based on that value, you might need to take a different action, or perhaps different logic needs to execute. Conclusion, while so far the flow of our application was a straight spot in terms of statements being executed, that won't be the case in most real life applications. Assume that you have been tasked with the creation of a C# program for Bethany's Pie Shop where the user can enter employee details. When entering the employee details, the application should ask the user for the age of the employee, and if the person is under 18, we can't hire them and the application should show a message to the user. Similarly, if the employee's age is above 65, we also need to show a message again saying that we can't hire them. Such logic we can include in our application using an if statement. Using the if statement, we can decide between two statements which one will get executed based on the value of a Boolean expression. You can see the skeleton of such an if statement. First, we use the if keywords, and then between brackets, we use our Boolean expression. If the result of that Boolean expression evaluates to true, will the if basically be triggered and will the statements between the curly braces get executed? If, however, it evaluates to false, the statements in the second block, the else block, will get executed. Else is another C# keyword, and it can only be used in combination with the preceding if.

The entire else block is, in fact, optional. You can just use an else block. Now let us look at a few examples. Here you can see the first real example of an if statement. We want to display a message if the value of the age variable is less than 18. For that, we are going to use the if here, passing between brackets the condition, age less than 18. If that evaluates to true, the code between the curly braces will get executed. If the condition evaluates to false, well, in this case, this code will simply be skipped, nothing will get executed. I actually may want to display a message to the user stating that all is good. We only want to show this if the value is, in fact, 18 or higher. We can therefore wrap this inside an else block. As you can see here, the else block as part of an if/else statement is optional, and you'll often encounter just the if part. One more thing. Between the curly braces, we can, in fact, put multiple statements. We're not limited to just one single statement. If, however, we have just one statement like we actually did, it's possible to omit the curly braces, as you can see here. Only if there is just one statement, we can omit them. And in that case, the statement right behind the if belongs to the if, and the one below the optional else belongs to the else statement. While this works, I'm often a fan of using curly braces all the time, since I believe it improves readability. And, of course, maybe in the future while you're improving your application, you might bring in more statements, which would require adding the curly braces anyway. To make things clear, this will not work. You'll, in fact, get a compiler error. Notice what I have tried here. I have two statements under the if, which is followed by an else. Only the first one belongs to the if, but since we have an else following the if, only one statement can be placed in between unless we bring in curly braces. Now, while we're at it, this also won't work. Can you guess why? Pause the video if you want to think for a minute. The answer is we're not using the equality check here, but we're using a single equal sign, and that is the assignment operator. The if statement expects a condition that results in true or false, but assigning 100 to age won't return that. So, this too will result in a compiler error. Next to even else there's also else if. Should we want to combine multiple condition checks, we can combine these into an if-else if-else statement, as you can see here. The structure of the if is the same as before, but now I have added another condition check, else if, that too requires a condition. We can bring in as many else ifs as we wish. Finally, the optional else can also be included here. As before, no condition is passed to else. Here we can see the code for our earlier requirement. If the age is below 18, we have the message we created earlier. If this condition is met, the rest of the statement is skipped, the other code is not evaluated. Then if the age is above 65, so another condition we wanted to check, we also want to

display a message. That's now in the else if, and that gets a condition, too. Again, if that's a hit, the rest of the code will be skipped. Finally, we have else again that will get executed if neither the if or the else if have been executed.

Demo: Using if Statements

In the next demo, we are going to work with if and else, and then we're going to bring in multiple conditions as well. So using an if statement, we can change the flow of the execution of our application. Instead of just executing statement after statement after statement, we can basically diverge into multiple, let say, branches within our code. And this can be as a preprogrammed, or it could also be that it is decided based on what the user has entered that a different branch is followed, and that's what I'm going to show you here. So assume that we are registering a new candidate for Bethany's Pie Shop and we need to have the age of the employee again. And I'm going to assume that the user is a good user and just enters a possible value, so a value that can actually be passed into an int, and that will be saved here in this age variable. I'm going to start by creating the first if statement here, and I'm going to check, again, using an expression if that value is less than 18. So if the value is, in fact, less than, smaller than 18, let's say, then I'm going to write to the console a message saying that person is is too young to apply. If that age is higher than 18, well, then basically, we're good to go. Notice in this if else statement I'm including an else. That is, of course, not necessary. So if the age is less than 18, this part is executed; otherwise, this part is executed. Also note that I haven't included any curly braces here, nor for the if block nor for the else block because it is just one line. Now, omitting the curly braces can indeed only be done if we have just one statement. So, for example, say that we also need to send an email to this candidate, and we have a second statement. This will give an error because now we basically have an else that doesn't have a corresponding if, and you see the word squiggly already appearing in Visual Studio. I need to surround this block here with curly braces. And now we can include as many statements that are only executed if the age is less than 18. And, in fact, even though this else only has one statement, I will also include the curly braces. I'm typically a fan of always including them because often you will come back to the code and you will need to add them anyway, and I think adding the curly braces here also makes the code more readable. So now we're only checking if the age is lower than 18. If I also want to check if the age is higher than 65, I also have a condition I want to check, and I cannot pass that to the else

block. I need to use the else if in that case. Again, my else if is followed by a set of brackets, and I'm going to check if the age is above 65. I'm going to continue what I've done here. I'm going to include curly braces, and I'm going to add something to the console. So in the else if block, we also need to include a condition. Let's run the application and see the result. So, we'll be a good user, and we'll enter a valid age. So I'll enter 25. And then it says here, great, you can now start with the application, so the age been validated. That means that it didn't fall into this bucket. It also didn't fall in this bucket, so we hit the else block in this case. Now, just to finish this demo, I also want to point out that it is not, of course, required to use only integers in combination with if. We can use just any expression that evaluates to a Boolean value, so true of false. Here I'm using, for example, a DateTime. I'm going to check if the day, so the day of the month, that is, of today's date is the 20th. If that's the case, well, then we hit this Console.WriteLine. In the else if block, I'm using the logical and operator. I'm going to check that if the day is higher than 25 and the end of month payments haven't started, then I need to show something on the console as well.

Using the switch Statement

The number of if statements you write in your C# development life, well, there will be a lot of them, I can tell you that. But in some cases, it might make sense to use the switch statement instead. Assume using the if-else if-else combination, you end up with something like you see here on the slide. Here we have a lot of conditions to check, and possible different bits of code that need to execute. Now while this will work, it might not be easy to read. Also, if some code needs to execute, if different expressions would evaluate to true, it also would be a bit harder. Introducing the switch statement. Things start with a switch keyword, which receives again an expression. This expression needs to get surrounded with brackets, and it is evaluated. Once evaluated, the execution will jump to one of the case statements. For the different cases, we can have a constant expression. So in that case, there must be an exact match between the value of the switch expression and the case constant. Alternatively, we can also use a relational expression, such as greater than. This expression is called the case label. Notice the syntax for different cases. After the case, we will have a caller, and then we'll have one or more statements to execute. Each case will then end with a break, yet another C# keyword, and that is used to indicate until where the execution should run for the used case. If no match is found in the different cases, the statements below the default are executed. Just looking at

this syntax structure might be a bit confusing. Let's make it more tangible with a real example. Here we are again going to evaluate the age value, and based on its value, a different block needs to be executed. Age is the expression that we're going to try and find the match for. It's going to be just a variable of type int here. Now a match is going to be searched. The first case will be selected if the age is below 18. So here the case is using a relational expression, less than 18. If that's the case, the statements here are executed until the break, and then we continue below the switch. If age is greater than 65, then the second case is executed. If age is exactly 42, the third case is used, and that one is different. It's using a constant expression. If no other case is a match, then the default will be used, which in our case would be the replacement of the else block we had before. While the switch can replace the if structure we saw before, it won't always be the case, though. The switch, in fact, has quite a few rules that apply. For starters, the switch can be used with every type. It works for most of the primitive types we've seen, excluding float and double. The case label, so the expression for each case, uses a pattern. This can be a constant like 42, or relational, like greater than 18. Each case must be unique as well. And as soon as the hit is found, so a matching pattern, execution will stop. Cases are indeed evaluated from top to bottom, and so the first match is executed. Default is optional, as said, and, in fact, doesn't have to be at the bottom. It will, however, always be the fallback if there are no other matches.

Demo: Using the switch Statement

Let's return to the demo and learn about the different options we have with the switch statement. So now I'm going to transform this if-else-if else combination into a switch statement. So I'm going to start by writing the switch keywords, and then I pass the expression. We are going to do the switch based on the value of age, so that is going to be my expression here. Then I need to include a set of curly braces. In between those, I'm going to write my different cases. The first case that I'm going to write is going to replace this if block, and the first case is going to be that age is less than 18. In that case, I'm going to write this line here. I'm going to just copy that over here and bring it over here. Now, of course, I can write multiple lines here, but when I'm done, when I'm done with this case, in fact, I do need to include a break here. This case will replace, well, this else if. And finally, in the default, I'm going to add this to the console. Let's copy that over and also include a break here. The default is, let's say, default true. If we cannot find another match, then default will be used. And that is, of course, the

same as the else in the if-else if-else combination. We can also omit the default entirely. If no match is found, then execution will continue after the switch block. It's just the same as if we can omit the entire block in an if else combination. So let's now run this, so let's put this one in comment so that we are sure that we are running our switch statement. So let's now again enter 25, and we are in the default now. So now I've used relational expressions, I can also use a constant expression. Say I want to say that if the age value is exactly 23, I want to do something special. I've now added another case with the label 23. So in this case, age would be equal to 23, and that will then be handled differently. Let's now run this again, as I now enter 23, and now we have an exact match, and that case is now selected. Now we have multiple conditions that we would like to handle in the same way we can share the code. Say that, for example, below 18 and above 65 needs to be handled in the same way. By saying in both cases that the age is not within the range we are looking for, what we then can do is, in fact, take this case and move it together with the case of below 18. I do need to remove this break. And now notice what happened. If the age is now below 18 or above 65, it will be handled with the same code. So I can share the code between these two cases then. Let's take a look at another place where we can use the switch. This code you can again find in the snippets. In the application that we are working towards by the end of this course, the user will be asked to make a selection out of the menu. And that menu we can construct easily using a switch statement. Assume that we want the user to make a selection between a number of options. Based on the option, we can invoke different codes. Of course, we could also do that with an if else block, but using a switch in this case is cleaner. So first, I'm writing to the console the different options, 1 for adding an employee, 2 for updating an employee, and 3 for deleting an employee. Based on the input of the user, we will then handle things differently. Let's see this menu in action already. So here I'm asking the user which action they want to do. I'll select one to add an employee, and then we can start the code to add a new employee. That functionality we'll add later in the course.

Adding Iterations

Now that we know about the most commonly used decision statements in C#, it's time to look at the other group, the iteration statements. And before we look at what C# has on board to work with iterations, let's first understand what iterations really are and what they are solving. While creating applications, there will often be the need to execute one or more statements repeatedly, so in a loop.

That will continue most often until a certain condition is met, that condition can be a counter value reaching a certain value, so effectively, until the loop has executed a number of times. Another option would be that we keep asking input to the user until they enter a certain value. Then too, we keep executing code until a certain condition is met. A loop can also be that we read recursively all files within a folder and its subfolders. So as you hopefully understand, the concept of a loop is very, very common, and we will encounter them very often when writing applications in C#. C# comes with a few options to create a loop. We have the while statement, the do-while, which is a variation of the while, and the for statement. Let's explore these in detail. And let's start with the while loop. It's definitely the simplest one of the three, although they're all pretty easy to understand. We will, of course, use the while keyword. That was to be expected. You pass in a Boolean expression. So that's the condition again. As long as the condition evaluates to true, the statements between the curly braces will execute. Only when the condition becomes false will the execution stop. That means indeed that we can from within the while loop change the state of the condition so that the execution will stop. When working with the while, the conditions of the Boolean expression will be tested before any statement is executed. Only if it's true will the statements get executed. Once the statements have finished will the condition be evaluated again, and if still true, they will get executed again, and so on and so on. I think you get the picture. Just like with if statements, a set of curly braces is only required if there is more than one statement you want to execute as part of the loop. Oh, and yes, using a while statement, it's pretty easy to create an infinite loop. If the condition simply never becomes false, your application's execution will be stuck forever in a loop. That is, until of course you close the application. Here I'm using an actual while loop. I'm using a counted variable here named i and I've initialized it to 0. I say counted, but it's just a regular variable, to be clear. Then I'm creating the while loop and its condition, and I'm checking if i is less than 10, if that's the case, which it will be initially, the statements within the while loop will execute, surrounding the value of i to the console first and then increasing the value of i. So i will now become 1. Then the condition is checked again, and it will still be true, so the loop would execute again until i reaches 10. Since then, the condition won't be true anymore, and the execution will continue below the while loop. In total, the loop will have executed 10 times.

Demo: Creating a while Loop

Visual Studio time! Let's go and create our first iteration using a while loop. And while we're at it, we will also create a nested loop, something that we haven't looked at yet. Let us first take a look at the while loop here. I'm going to start by asking the user to enter a number, and we're going to read that again into a variable. Again, we're assuming that the users are only entering numeric values. I'm going to write a loop in which I'm going to execute a little bit of code a number of times, and that number is entered here by the user. I do need another variable, and I'm going to call that my counter, let's say, and typically we use i as the name for the counter in the loop, but again, you can use whatever you want here. I'm going to set that to 0. That will be my counter that keeps track of the number of times my loop has executed. Now, I want to execute a piece of code a number of times, and that number is your max. I'm going to use for that of course a while loop, and so I'm going to execute my code as long as i, my counter, is lower than that amount max. That's the amount of times I want to execute the code in my loop. So as long as i is lower than max, I'm going to execute my code. The code that I want to execute is simply writing at this point the value of i, the counter, to the console. And lest you forget that I do increase the value of i because no one else is doing that. I need to do that in the loop so that I keep track of the amount of times the code has executed. And so as soon as i is equal to max, the while loop will exit, and we're going to arrive here. So we'll write something to the console to know that the loop has finished executing. There we go. We'll enter value at say 15, and we see that i is initially 0. The loop executes again, and i becomes 1. The loop executes again, and so on and so on, until i is 14. Then it becomes 15, and the condition is no longer true. Hence, we exit the while loop and we arrive here on line 105, which is going to write "Loop finished!" to the console. Now it doesn't always have to decrease. Say that i, the counter, is 10 already. We can, for example, loop until i is 0. So as long as i is above 0, it will execute, we write i to the console, and then we deduct 1 from i. The result is what you see here. We start at 10 and we go all the way to 1, in this case, because we stopped at 0. When i was 0, the loop has not executed anymore. I want to come back to what we had here. In the previous demo, we had written this small menu in which the user could make a selection to then diverge into other parts of the application that we are going to create later on. Now, we basically hit an endpoint. As soon as the switch statement is executed, we don't have the ability to go back to the top unless we repeat the switch statement. But how many times do we need to repeat it? Well, of course, we can use a loop in combination with this. Let me show you what I've done here. I've now changed my application a little bit so that I have an extra option. 99 would exit the application. Now, I've used a

wire loop around the switch statement, and that loop will continue running until the selected action is equal to 99. In other words, as long as it's not 99, it will continue executing. So if I enter 1 here, we will enter here, and then we enter here in a switch statement, and we go in case 1, and then I show the menu again, selectedAction gets updated, and we go back to the top of the while loop. If the value is still not 99, we again based on the selection execute code, and we show the menu again. And this loop will continue executing until the user types 99, which will exit the application because then we exit the while loop and we come over here. Let's try this out. So first as user, I want to maybe add an employee, then I want to update an employee. I want to add another employee. And you see that I get that same menu all the time, as long as I do not enter 99. When I enter 99, we are closing the application. So this already gives you a bit of a real feeling of an interactive console and interactive terminal that allows the user to enter the value, execute code, and continue doing that until they enter 99, in this case, to close the application. I want to show you a couple of small extra things around the while loop. Loops can actually be nested. We can nest them as deep as we want. Notice that I now have two nested loops here, and I also have two counters. The first while loop will run as long as i is below 10, and the second, the internal, the nested loop, will run as long as j is below 10. I'm going to write something to the output, the value of i and the value of j. Then I increase the value of j in the inner loop, I set j back to 0 because otherwise it stays on 10, and then I increase the value of i. This j = 0 is important here because that inner loop will otherwise only execute 1. Now, in total, we will have 100 executions. Let's try this out. There's quite a few lines that have been printed here, 100 to be exact. So first, i was 0, j was 0, i was 0, j was 1, and so on until j was 9. At that point, that condition for the inner loop was no longer true, j is reset to 0, i is increased, and we start the outer loop again. One last thing, if I want to create an infinite loop that keeps on doing something, we can do it as follows. You can say while (true), print the DateTime to the console, and this will keep on running until I close the application. If we run this, we'll see that continuously dates are being printed to the console.

Using the do while and the for Loop

Closely related to the while loop is the do-while loop. I must say that I don't use it that often. But let's look at it briefly. I'm sure you remember that we saw with the while statement, the condition is checked before any execution of the loop. As so if the condition is false to begin with, the loop will not be executed at all. The big difference with the do-while statement is that its body will get executed at least

once, since the condition is checked after the run of the loop statements. Here, you can see the structure of a do-while. Notice the new keyword here, do. There's no condition behind that. And because of this, the statements between the curly braces will get executed. Then we will create the while statement, and that again comes with a condition. If true, we will go back to the beginning, and an execution of the loop is start again. If false, we will continue below the do-while. But as you surely have noticed, the statements inside the body have executed at least once. Here you see an example of a normal do-while loop. We have our int i again, and it's again set to 0. Then we enter the do-while loop and the body gets executed and i will also increase. Then the condition is checked. And since it will be 1, it's all still good. The loop will start again. This will happen until i is 10, then the condition will be false. And so the output of this do-while and the while we saw earlier is exactly the same, although there is one difference. After completing the while loop, i is still 9, whereas using the do-while it is 10. Can you figure out why? I'm sure you can. Say that we now initialize our variable as 10. Because the condition will only get evaluated after execution of the loop, the loop will have executed exactly 1. The condition, however, is false upon first check, but at that point, the body has already executed. We'll see the do-while in the demo in just a minute. Before we head back to Visual Studio, let's look at a for loop. Technically, there is very little difference between the while and the for, but I use for more frequently. The reason is that it's more compact, since it does everything in one go. With the while statement, we typically have to initialize, then do the condition, and then inside of the loop perform the increment. With a for loop, all of this is done typically with the setup of the for statement. Here you can see the structure for a for statement. Of course, we use the for keyword, and that now receives the initialization, condition, and iterator, all semicolon-separated. The statements to repeat are again placed inside the curly braces. Since we have for, all these need to be specified. It is less fragile. You, for example, can't forget about the iterator, since it is required in the initialization of the for loop. That is an error you'll often make with the while loop. I can guarantee that. I've been there myself quite often. Here's the for loop in action. First, we write the initializer, int i = 0. This is, to be clear, only executed once. This is initialization. The variable is declared and initialized inside the for loop. Note you could also declare it outside of the for loop, but if the purpose is acting as a counter for the loop, you would just use it inside of the loop and declare it here. Then we have the condition expression that determines if the loop should run again. If true, the statements in the body will execute. That will again

be a Boolean expression. Finally, we have the i++. That's the expression that defines what needs to happen at the end of each loop here, and that's increasing the value of i, so our counter.

Demo: More Loops

Time for the final demo of this module. In this demo, we're going to work with the do-while and the loop statements. I'm also going to show you some more options around loops, namely how to break out of them while they're executing, using yet two more keywords, break and continue, and finally, I'll show you how to debug loops, as they sometimes can be a bit harder to do so. Do you remember this code from the previous demo? So first we showed the menu to the user, and then based on the input, we were going into the while loop. If the selected action was not 99, and then based on the selected action, we went somewhere in the switch, and then we show the menu again. But notice I have the menu in here twice. If I need to add a fourth or fifth option, I need to enter it here and here. That's not really nice. We can, in fact, do a better job. What we actually want is to always show at least once the menu, and that we can actually do with a do-while loop. Let me show you an updated version. Notice now I have a do-while loop. In that do-while loop, I'll write as the first thing the menu. I capture the selected action. If it is 99, we basically won't find a match in the switch. We will then hit the condition check on the while loop, and we will exit it immediately. If we do find the match, well, then we go into the switch, and we execute one of these. And then we go back because selected action is not 99, and we go back to the top, let's say, and then we show the menu again. So the criteria to choose between a while and d-while is the following. If you want to make sure that your code in the loop executes at least once, then you go for the do-while loop. To close things here, I'm going to show you the for loop, which is, in fact, the loop I tend to use most. The main reason is probably its compactness. Let us write one together. So we start, of course, with the keyword for, and then we have brackets. In the first part, I'm going to define our counter again, and we can do that directly in line here. I can just write for(int i = 0. Basically, my counter has now been created. Then comes the condition part. And let's say that we want to create a for loop that we want to execute also as long as i hasn't reached the value of 10. So as long as i is then smaller than 10. And then we go into the iterating part. I need to make sure that my counter also gets increased. This code will only get executed after completing the execution of the for loop. And in this case, I will just use i++. Notice the semicolons between these different blocks here. Now we use a set of curly braces again, and then we can just write i to the console. If we run

this, we'll see a simple counter that starts at 0 and executes until i is 10. And again, the loop hasn't been executed as soon as i is 10. And to finish this demo, I'm going to show you two other keywords which can be used in combination with the loop statements. So they can be used with the for, but also with the while and the do-while. And they are the break and continue, and they influence how the loops are executed. Let me show you. Here I have again a loop which is going to run, but it does contain an if statement, and it's going to write if(i == 5) something special to the console. But notice what I have below there. Continue. Continue basically means that all other statements within the loop, within the current execution of the loop, I should say, are skipped. We basically go back to the start of the for loop. I'm asked to enter a value. Let's enter 10. Notice now what has happened? First, 0 is shown. That is because of this line. Then 1 is shown; same thing. As soon as we find 5, if i == five, then we write this line to the console, and we call continue. This line is not executed anymore. We are basically skipping all the other statements within the current execution of the loop, and we go back to the start of the loop. And we can also break out of the loop. If we want to, for example, stop execution of the loop entirely when we find the value 5, we can use its sibling, break. Now, as soon as i == 5, execution of the for loop will stop. Let's run things again, and let us again enter 10. And now you will see that we hit 01234, then 5, and then we hit the break statement. The Console.WriteLine is not executed anymore, and the loop is also stopped. Execution continues below the for loop in that case.

Summary

Yes, smarter again! In this module, you have learned that code execution won't always follow a straight path, but we sometimes, in fact, often, branch into different sets of statements. This can be done using decision statements using the if and the switch. If there's code that we will need to repeat a number of times using a loop, C# comes equipped with a number of statements, namely while, do-while, and for. We've looked at the differences between these different options to create loops here. In the next module, we will dive into working with methods.

Using Methods in C#

Module Introduction

Writing all of our code in one large block won't be very readable when building real-world applications. It might be okay for a small demo, but once we are past that stage, there are definitely better ways. Let's in this module understand methods in C#, a way to group and reuse code statement. I'm still Gill Cleeren, and I'll be guiding you through this module called Using Methods in C#. Let's see what we will be covering in this module to kick things off. First, I'm going to introduce you to methods. What are they, and how do you create them? As we have created them, we also need to learn how to invoke them, and that we will see here as well. Next, we will do some refactoring in our code, that is, reordering our code for the good, and we'll move things into a helper file, and I'll explain why that is when we are there. We'll also learn how C# finds the correct method. And while doing so, we'll understand method overloading. Next, I'll discuss briefly this thing called variable scope, since that comes into play when using methods. We'll dive a bit deeper into other aspects of working with methods. And to end the module, I will introduce you to a method you have, in fact, already been using without knowing it, and that is the main method. Lots of cool stuff to learn, I think. Let's get started!

Understanding Methods

As always, first things first, let's understand methods in all their glory. So far, the code we have written is, well, pretty sequential. I think you'll agree with me on that. In the previous module, we have the ifs and the whiles that allowed us to jump between different branches or repeat certain parts of the code. But still, in general, our code is one large block. Say that we have written code that calculates the wage of the employee. That would be code that we may wish to call several times while running the application. It wouldn't be the best idea in the world to write that code several times now, would it? Because wage calculation it'll change over time. If we'd have that code several times in our application, we'd have to make a change in several places. The solution is pretty straightforward. We'll write the code in one place and call it. That place is a method, and while the name methods might not ring a bell, maybe the word function or subroutine does. Methods are pretty much the same. But so what are they really? Methods are very common to use in C#. They are code blocks which contain a number of code statements. They have a name, and by calling that name, we can invoke the method, and thus the code that lives within that method. Methods can be any size. They can contain just a single line or contain a whole bunch of statements. Frequently methods will define that they require

one or more parameters in order to run. And thus, when we call the method, we need to pass in a value for these parameters. The values that we are passing in are called arguments. Methods can be used to define functionality to do all sorts of functionality, and based on that, they can optionally also return a result to their call. By placing our code in methods and thus not creating a long list of statements indefinitely improve the readability of the code. But not only that, methods also help with code reuse. By creating applications, you'll often do the same thing as I mentioned before. By putting this function at the inside of a method, you can invoke that method from different places within your application, hence improving code reuse. Finally, methods always need to be declared inside of a class or a struct. You may ask, what are they? Well, they are very important building blocks for C# applications that we will look at soon. Methods are, in fact, always declared inside of a class or a struct. That will become clear soon enough. Just keep this in your brain tucked away for now. That'll do. Methods are always created with pretty much the same structure. In general, you'll always see the same items come back when defining a method. On this slide, I've captured the general structure of a method. A method has to have a name and requires two brackets behind this name. Then the actual method statements, so the body of the method, goes between two curly braces. Essentially, the items in orange are required. In fact, what is also required is the return type, which indicates what type of result will be returned from the method. It's possible that the method isn't returning anything. Then still we need to indicate this with a void return type. More on that soon. A method can optionally also define it requires one or more parameters. These need to go between the brackets. Essentially, the parameters are declared as if we are defining variables using a type and their name. When two or more parameters are defined, the parameter list needs to become separate. And if there aren't any, we still need to add the brackets, but just leave them empty. Finally, it can optionally also have an access modifier defined. Now we will talk about access modifiers later. For now, remember that this can be used to indicate from where this method can be invoked. All of this together is what is referred to as the method signature. Here is a first method. They almost don't come any simpler. The method's name is AddTwoNumbers. It has an int return type and it is publicly available. Pretty simple, isn't it? Now a method named AddTwoNumbers which doesn't receive the two numbers to add, well, that doesn't make a lot of sense, now, does it? That's why next I'm going to add two parameters. Int a and int b are these parameters. Now, the method signature defines that when someone wants to call this method, they will need to pass in two integer values, which will go in this a and b variables. Just to be sure,

notice that we need to specify here the type of the parameters, so int before a and b. If you would try to compile this code, it would actually give us a compiled time error. Any idea why? Well, the answer would be the method has a return time specified, the int here. But when doing so, the method must do what it says. It should actually also return an integer value at the end of the method body. With this line added now, return a + b, this will be the case, and the compiler will be happy. Return is another C# keyword that essentially stops the execution of the method and returns the value here to the caller, and this also the execution returns to the caller. Am I saying the caller? I mean the code that will call or invoke this method. And we'll see this in just a minute. Most often the return statement will be at the end of the method. When a method defines its returning int, in fact, all code execution paths should do this. Again, otherwise, you will get a compiler error. It is possible that the code inside of our method has multiple execution paths due to, for example, an if statement again. Say that we indeed include an if statement, like you see here on the slide. There's a possibility that no value is being returned from the method that would be if b would be greater than a. C#'s compiler won't allow this to go through, and it will raise an error. You should also include code here that returns a value for the case a is less than b, at least. But now what if the code in the method just performs the task? Say show text on the console. What should it return then? Well, great question! It is not required for a method to just return anything. The only thing C# requires is that we then indicate this. For this, you have the void return type. This is an indication that the caller shouldn't be expecting anything to come back from the method, and the method now also doesn't need to include a return statement. And once we have defined the method, we can also invoke it or call it. That's the same. If I have that DisplaySum method, so the void method I just showed you, we can now invoke it by calling DisplaySum. And then between the brackets, we now need to pass in two values of type int, since that was in the method signature. These values, so the values that we pass to a method, are the arguments that we use when invoking the method. For each parameter defined in a method declaration, an argument must be provided when invoking the method. The arguments provide the values for the method parameters. The different arguments are comma-separated, as you can see here. Oh, and one more thing, the method name has to match exactly. So, including the casing. C# is case-sensitive. I do understand that calling DisplaySum basically changes the flow of execution again. Instead of continuing to execute statements over here, the statements in the DisplaySum method are executed. And only when the DisplaySum is finished executing will the flow of execution return back here. Now our DisplaySum had

the void return type, so it's not returning anything to the caller. And that it is different for the AddTwoNumbers method, which was returning as the sum of its two parameters as an integer value. So, we can assign the returned value of invoking AddTwoNumbers to a new variable here called result. Result will contain after execution here the value 99 and can then be used locally here. Instead of directly passing values as arguments, we can also pass in the name of variables. Here you can see that I have defined two local variables, p1 and p2. We're now calling DisplaySum like before, but now we are passing in the variables instead. This is something you'll do often.

Demo: Creating and Using Methods

All right, time to head to Visual Studio for the first time in this module. We already have a basic understanding of methods, so we'll use that to create a few methods. I'll show you how we can add parameters and return a value. Because just creating a method won't do a lot, we also need to invoke them, and that I will show you here as well. So far, we've written all our code directly in this Program.cs file. And not only have I written all the code in this file, it's also in a sequential mode, meaning that all these statements will be executed top to bottom. And if there's parts in there that we need to repeat multiple times, well, then we need to write them also multiple times. That is not an ideal situation. So in this module, we're going to fix that and we're going to start introducing methods. So I'm going to start with removing all the code that we've written so far. In fact, it was all small snippets. It wasn't a real application just yet. So we'll start working on that right now. What I want to do in this demo is write some logic to calculate the wage for someone. We've already written a bit of logic for that, but now I'm going to write it inside of a method, and that method we can then call from multiple places within our application, and logic will be in one place. And so if we need to change it, we can just change it in one place and we don't have to make a change in multiple places. So, let's do that first. I'm going to start by writing our method while still within the Program.cs file. I'll come to that, I'll change that, in fact, in the next demo, but now I'll write our first method still within the Program.cs file. So let's start with writing our first method, and, as said, it's going to be a method that calculates the yearly wage, so I'm going to give it a meaningful name, CalculateYearlyWage. Now, the static I'll come to later. That is a keyword that we need in this case, and I'll explain why in a later module. Void is the return type of our method. So our method is going to contain code, and that is going to perform some logic, some functionality. And in this case, the method is declared as void, meaning that it will not

return anything. We'll see later on, of course, that methods can also return something. Then we have the name of the method, and then we need to add a set of brackets. We always need to add a set of brackets. That will contain the parameters. And even if we don't need parameters, we still need to put the brackets in there. The method body, so the logic that our method will contain, needs to go again between a set of curly braces. So here's our first method. It doesn't do anything at this point just yet, so let's change that. So our method is called CalculateYearlyWage, so we're going to have to include some functionality that at least calculates that yearly wage. And it can be called from multiple places within our application. So that means that this method will be called, will be invoked, let's say, from multiple places within our application with different values. And so I need to capture these values that are going to be passed into my method using one or more parameters. So as said, a method can, but doesn't require parameters. But in this case, I'm going to include a couple of parameters. Two, in fact. I'm going to pass in the monthly wage, and that's going to be of type int. So you see that I'm declaring what looks as a variable, so including a type int and then a name, again, in camelCase. If I add multiple parameters, they need to be separated using a comma. So I'm going to also pass in the number of months worked. So now I have two parameters of type int that my method is getting in. These parameters are available within the method body, not outside. I will need to pass a value into these parameters when I invoke the method, of course. Now, in the method body, I'm going to have to do my calculation, and the functionality is going to multiply monthlyWage with numberOfMonthsWorked. Very simple. And I'm going to write that to the console. You see here that I'm doing indeed that calculation, and I'm then writing it to the console. Now you do see here a syntax that we haven't covered yet. We'll look at it in the next module. This is called string interpolation. Now, string interpolation allows us to while inside of the string also use values, and these values need to be between a set of curly braces. If you put a dollar sign before the string, these values will be replaced at runtime. This is a very common way of creating concatenated strings like we're doing here. All right, our first method is ready. We can now start using it. We can invoke it. So I'm going to go here, so outside of the method, but again, inside of the program file. I'm going to declare a couple of variables, again, of type integer, amount, and months. These are now variables available to use inside of my application, and I'm going to pass these to my methods. I'm going to invoke my method, and I'm going to pass these in. Now, invoking the method is basically nothing more than writing the name of the method, CalculateYearlyWage, and then I always need to use a set of brackets, and I need to pass in

values for the parameters, and these values will be arguments. And I will use for that the variables I just declared. MonthlyWage is amount, so I'm going to pass in amount to my method, and it is going to arrive here in monthlyWage. So, argument 1 will be passed to parameter 1. Argument 2 will be months, and that argument is going to be passed to this parameter here. Now, here you see also the importance of creating methods. If I now need to invoke this CalculateYearlyWage functionality from multiple places in my application, I just call it multiple times, and I can pass in different values for my parameters or different arguments. For now, the number of arguments that I'm passing in needs to be the same as the number of parameters that my method accepts. What's also important is that the arguments that I'm passing in need to be of the same type. My method requires here two integer parameters, and the values that I'm passing in, so the arguments that I'm passing in, are also two integer values. I'm going to add here a Console.ReadLine. I've now added a few breakpoints, so we can step to our application with the debugger attached. Let's do that now. So first our application hits the CalculateYearlyWage. We will now click on Step Into, so F11, to jump into the method. Indeed, now this method is being executed. We can now do an F10, and then we will get our output. After the method is done executing, control will return to the CalculateYearlyWage, which has then executed, and then the flow continues over here, and we see the result on the console. Now, in this case, my method has just given me some output. The chances are that in the caller, so from here, I will also need the result of that calculation. In that case, my method, which is now not returning anything, it's void, will need to return a value. For that, I need to also declare that this method will, in this case, return, for example, an integer value. Now we get a red squiggly because Visual Studio say, well, this method says that it's going to return an integer value, but it is not. I need to change that. So therefore, instead of just giving me some output, which I'm going to comment out for now, I'm going to now use a new keyword, the return keyword, and that is going to return the multiplication of monthlyWage times numberOfMonthsWorked. That is now going to be returned to the caller. At this point, the caller is not doing anything with it. If we hover over CalculateYearlyWage, we indeed see that it's going to return an int, so that means that I can now do int yearlyWage, and that yearly wage is now going to get in the result of invoking our CalculateYearlyWage method. We can now in the caller use the result of invoking the method, and that's what I'm doing on this line here. The value that's being returned by my method needs to be of the same type that it is declaring here, and therefore also here because of type-safety, I know that I'm going to be, in this case, be getting back an int. Now let's make one more

change to our method for now. In this case, the method is very simple. It's just doing that multiplication here. But it can, of course, also contain extra logic. We could, for example, include an if statement here that checks if the numberOfMonthsWorked equals 12. In that case, we are going to return monthlyWage times numberOfMonthsWorked + 1. And otherwise, we continue with the old return statement. So as you can see, we can have multiple return statements. As soon as we encounter a return statement, the execution of the method returns to the caller. So that means that after we've executed this return statement, we immediately return to the caller. If we run the application again, we see that we get back a different yearlyWage because the number of months that we passed in was 12.

Demo: Adding a Helper File

So far, we have placed all code in the default generated file. So, the Program.cs file. Now, while that works, even for smaller applications, we typically have multiple C# files in our application. I'm going to show you here how we are going to start moving our application's functionality to a separate file, which will contain a class. The specifics of this concept of a class will be explained very soon. We're going to do this directly in a demo. I'm going to do what is called refactoring. We're going to modify our code, and we'll make sure that the functionality of the application stays the same. I'm going to move the code we have to a separate file, let's say, a helper file. And we're going to move our method as well, and we're then going to invoke our newly created method again. The goal is thus that the functionality of the application remains the same. In the previous demo, we already created our method here. That is a way to reuse code. But leaving everything, even multiple methods inside of the program, also won't make for a viable application. So we are going to refactor our application's code so that we move things into different files. We'll still leave the code that will execute and start up in the program file, and the methods will now move to a different file, which is going to be a different class. So let us refactor our code, and so refactoring is a term that points to reorganizing our code, but making sure that the functionality stays the same. It's a process that you will often do when writing applications. So I'm going to add a new file here in my project. As mentioned before, the project is a container for all the files that you need in your application, so if you have another code file, we'll add it to our project. So we can right-click on the project and go to Add, New Item. I will move our code into a new class. I'll explain later in a lot of detail what classes are for. Now, we'll see them as a place to put our method in. I'm going to call this file, this class, Utilities. So a new file, Utilities.cs has now been generated, and it

contains a class Utilities. It also contains namespace and using statements. We'll talk about those later on. For now, I'm going to use this Utilities class as a place to put my method that we already created. So I'm going to go to my program file and cut my method that we created earlier, and move it into the Utilities class. Make sure that you put it inside of the body of the Utilities class. Now, I do need to make one change to the method. I'll explain later what it is. We'll need to make the method public, so it is, in other words, reachable from the outside. Let's go back to our Program class. And indeed, now we see here that CalculateYearlyWage can't be found anymore, which is pretty logical if you think about it. It is now inside of a different file inside of a different class. We're getting an error on that, getting errors quickly on that. So what I need to do now is say to Visual Studio that CalculateYearlyWage can be found within Utilities. And I now I do a dot, and then CalculateYearlyWage. And I'm still getting a wage quickly here. If I now hover over Utilities, Visual Studio is suggesting me a potential fix. Let's click on that, and it says here I need to bring in using BethanysPieShopHRM. That's, in fact, bringing in a using statement for the namespace image the new class Utilities lives in. We'll talk about namespaces also later. For now, make sure that you have that using statement there at the top, and then Utilities will turn green, and CalculateYearlyWage will also get a different color. So now I'm basically invoking from within the Program class CalculateYearlyWage that lives here in the Utilities class. Let's run our application again to make sure that everything still works as before. And indeed, the yearlyWage is the same as we saw in the previous demo.

Finding the Correct Method

So far, we had just a very simple example where the method was, well, pretty easy to find. But how does C# actually find the method that we intend to use? You have seen that the method requires a name, a return type, and 0 or more parameters. When we ask C# to invoke a certain method, it will start searching for the corresponding one. To search for the correct method, it doesn't just use the name. No, indeed, a few things are considered to locate the correct method. The method name is, of course, the first one. Then it will also see if the call to the method is passing the same argument types as the method requires. The method may say it needs a string parameter, but if we try to invoke it passing an integer argument, the C# won't match it. This, of course, is a good thing. It's the type-safety of C# that will make sure at compile time that not only the name matches, but also the parameter types. Also, the number of parameters is taken into consideration when searching for the

correct method. In conclusion, we can see that three things are used to find the correct method, the method name, the parameter type, and the number of parameters. Only when this combination is found can C# invoke the method. And it also brings another thing, it is this combination that needs to be unique. Not just the name. Indeed, we can have multiple methods with the same name. As long as this combination stays unique within a class, we're good. The fact that multiple methods with the same name can coexist within a class is called method overloading. The name can be the same, but either we need to define a different number of parameters, or different parameter types, or a combination of course will also work fine. Take a look here on the right. I have two methods with the same name, both called DisplaySum. The first variation has two integer parameters, and the second one has three parameters. C# will accept these. These are now overloaded methods. And I now invoke DisplaySum as you can see on the left with just two parameters. The highlighted version will be selected. C# will manage this method, and this one only here because of the number of parameters.

Demo: Using Method Overloading

Let's go back to Visual Studio and take a look at the method overloading in action. When I said here CalculateYearlyWage, C# went out looking for a method called CalculateYearlyWage. But not only does it go out and look for a method with that name, it will also look for a method that has a corresponding parameter list to what I'm passing here to integer arguments. So when we invoke a method, C# will not only search for a method with the corresponding name, but also the number of parameters, and the types of parameters should match with what I'm invoking. Now, that means that I can also create overloaded methods. So instead of always creating a new name for a method to always finding a new name, let's say, I can also create methods with the same name, but with a different parameter list. The parameters can be different in terms of name or in number. Let's try out method overloading. I'm going to go back to my Utilities class here, and I'm going to bring in a new method with the same name, but with a different parameter list. Notice I now have two methods with the same name, CalculateYearlyWage. This one is an old one, and it had two parameters of type int. The new one has three parameters of type int. We have an additional int bonus parameter that I'm also using here inside of the method body. This is method overloading, and I have two methods with the same name, but a different parameter list. If we remove the bonus parameter, we're not only getting errors within the method body because bonus can't be found, we are also getting an error, a

red squiggly, on CalculateYearlyWage. That method is now not unique anymore because the name is the same and the parameter list is the same. So let's bring bonus as a parameter back in here. I can create another overload. This time I'm using again the same name, but now I'm using double parameters instead of integer parameters. That, too, is possible. The methods can be different in terms of number of parameters or type of parameters. So these are now different overloaded methods. Let's now go back to the Program class. I'm going to bring in int bonus. I'm going to set it to 1000. When I now call Utilities., you will see that CalculateYearlyWage shows up in this list. And when I now type the first bracket, you will see here that I get this list of overloaded methods, which differ from each other in number or types of parameters. I'm going to add here bonus as the third parameter. I'm going to put a breakpoint in each of the methods, so we can see easily which one gets called. There we go. Let's run the application. And as you can see, the one that has the bonus parameter is the one that's being invoked because indeed, over here, I invoked the method with three integer arguments. If I do the following, I create a few double variables, and I now invoke utilities to CalculateYearlyWage. What will happen, do you think? Well, let's try it out. And let me put this one in common before we run things. Well, indeed, the overload that expects double parameters is being invoked because we invoke the method passing in arguments of type double.

Understanding Variable Scope

Now one thing that you may not have noticed yet is this thing called scope. Scope is about where variables once defined are accessible and thus known. Let us take a look at variable scope. Next, we have declared our first methods. In these methods, we have declared variables to store values in the memory of our application. Here you can see the method called SomeMethod, and it's similar to the one we have already created. Now in there, I have a variable called value, and I've given it a value of 0.04. What happens is that when this statement is executed, that variable will get created in-memory. Statements below it can also work with it. It is, let's say, alive at this point. When the code hits the return statement, the method will be finished, and at that point, the variable value will also cease to exist. It is defined within the method SomeMethod, and only within this method after its declaration can it be used. The variable now has what is known as local scope. Scope refers to the region where the variable is known, and that is here indeed, the method. The variable is defined between the opening and closing braces of the method. Hence, it is known within the method only, and thus its scope, its

region where it's known, is the method. This variable is also often referred to as a local variable. If we would try to compile the following code in our application, the C# compiler wouldn't let things go through. See what I'm trying to do here? In AnotherMethod, I'm also using value. Our variable, however, was declared inside the other method, SomeMethod, and it is locally known, so within the method, but not outside of it. It has local scope only. So, method scope. Because value is only known here within the method it is declared in, we can, in fact, in AnotherMethod define a value again. Each method can define a variable with the same name. They aren't different variables, to be clear.

Demo: Using Variable Scope

In the next demo, we're going to work with the scope and see its effect. Let me show you scope in action. We are again here in the Utilities class, and we have our methods that we created earlier in here as well. Say that I create in the CalculateYearlyWage a local variable. Let's call it int local = 100. This variable is now known within this CalculateYearlyWage method. The curly braces define its scope. It is known in this block and this block only. That means that if I now go to this second CalculateYearlyWage, and let's say, for example, local = 150, we'll get an error. Visual Studio is simply saying the name local does not exist within the current context, so within the current scope. Because it's defined here, it's not known over here. Now, although we saw earlier that identifiers need to be unique, they need to be unique within their scope. So that means that I can create another int local within this method with the same name. That's, of course, possible. That shows you what local scope or method scope means.

More Options with Methods

We've already touched on several interesting topics around methods. When working with parameters, which we often will do, there are quite a few options, too. In this part, I'm going to show you a couple of these, let's say the most commonly used ones. More specifically, these topics we need to take a look at. First, I'm going to show you optional parameters. Then, we'll look at named arguments. And finally, I will show you expression-bodied syntax. So first, optional parameters. What are they all about? Can we just randomly leave out an argument by calling a method? Well, yes and no. Let me explain. Using optional parameters, it is possible for us when creating a method to specify a default

value in the parameter list for one or more parameters. When we then invoke the method, it is possible to leave out the optional ones, and the method will rely on the defaults that we have defined. Let me show you a small snippet here with optional parameters. In the snippet on the left, I have defined an optional parameter. Notice I have added a default value for the third parameter. That is now an optional parameter. We don't need to add another keyword or something to make it optional. Just by adding a default value, C# knows that this will be its default value and should be omitted. Now on the right, I'm calling this method. In the first call, I'm just passing in two arguments, meaning that the default value for C will be used. And in the second line, I'm just calling it regularly, so with three arguments. In one go, I also want to explain you named arguments. Named arguments have been around for quite some time. They were added in C# 4 already. Now, what are they then? Normally when invoking a method, the order in which we specify the arguments matches the order in which the parameters are defined in the method declaration. Argument one matches parameter one, argument two matches parameter two, and so on. That's what we have seen so far in this module, right? However, it's possible to specify the name of the parameter when using the argument, and then the order isn't important anymore. Matching then happens based on the named arguments instead. Let me show you this. In the snippet on the left, we now have our regular method again, nothing special. And on the right, I'm calling the method, but notice that I now specify first the name of the parameter b, and then a colon, and then the value for b. I'm just passing in first b, and I can only do so by stating this explicitly when invoking the method. I'm doing the same here for a as well. Now, why would you use this?

Demo: Using Optional Parameters and Named Arguments

Back to Visual Studio. I will see these features in action. First, I'm going to show you optional parameters, and then I will show you named arguments. We are back here in the Utilities class and have added this new method here. It's called CalculateYearlyWageWithOptional, and it's very similar to what we already had. It will return the monthly wage times the number of months worked, plus the bonus. But let us now assume that the value of bonus is in most cases 0. We can, in fact, define a default value in the parameter list for the bonus parameter, and then we can invoke it without passing a value for bonus. It therefore becomes an optional parameter. We can do that by just specifying here = 0. There we go. Now, bonus is optional. We can invoke the method now with two or three

parameters. Instead of invoking this method directly from the program, I'm going to introduce a bit of a different way to do so. I'm going to bring in another method, UsingOptionalParameters, and that will in turn invoke CalculateYearlyWageWithOptional, as you can see here. But now notice I'm invoking it with two parameters instead of three. That is possible because bonus has a default value here, it's an optional parameter. If I want, I can also pass in a different value. And still the method is found now with three parameters. Let's remove this 100, and let's put a breakpoint over here. And I'm now going to invoke UsingOptionalParameters from Program. Let's put all of this in comment, and let's called Utilities.UsingOptionalParameters. Let's now run the application and see how optional parameters work. So we're now here in CalculateYearlyWageWithOptional, and notice that bonus is 0. We haven't passed in a value over here, so its default value is going to be used inside of the method. One more thing, the optional parameters do need to be placed at the end of the parameter list. You cannot mix, let's say, the optional and the non-optional ones. Let me now show you those of named arguments. I've added another method here, CalculatedYearlyWageWithNamed, but on the method itself, nothing has changed. It's simply using a couple of integer parameters, as we did before. Now, let me invoke the method. I've added another method here, UsingNamedArguments, and I've already brought in a couple of variables. Let us now invoke the method. I'm going to invoke our CalculateYearlyWageWithNamed method. Again, the method itself is nothing different from what we had before. But now I'm going to pass in the parameters in a different order. Now we already saw that argument one matches parameter one, argument two matches parameter two, and so on. But if you want to pass them in a different order, we need to pass in the name of the parameter first. So that would be bonus. That is this bonus. And then I specify colon, and then the value. And it would be bonus, so that's this bonus here. Then we can do, for example, monthlyWage, again, passing in a colon, and its value is amount. And then we do numberOfMonthsWorked:, and then months. So what I've done, I've invoked the method, and for each parameter, I first specified the name of the parameter, and then colon, and then the value. And then I can pass them in in whatever order I want. Of course, all required parameters need to get a value. And so I invoke this also from our Program, and if I just replace this here, there we go, let's run the application again and make sure that everything still works as before. And here we go, we get our output as we did before.

Demo: Using Expression-bodied Members

Let's now learn about expression-bodied syntax, a syntax to make short methods, well, even shorter. But I can imagine the first time you see it, it might be a bit confusing, so I want to include it here. Expression-bodied syntax can actually be used for methods where the body consists out of a single line. If it's a void method, it can be just one operation. And if it does return a value, that operation must also return a value of that return type. Let's take a look at expression-bodied syntax. I have another method here using expression-bodied syntax, which is going to call CalculateYearlyWageExpressionBodied. We don't have that one yet, so I'm going to write it, and that's going to be the one that uses expression-bodied syntax. So I'm going to write the method here. It's going to be again a public static method. It needs to return an int. So that stays the same. The return tab needs to be defined here as well. And its name was this one. Let's copy that over here. So the method signature hasn't changed, and that includes any parameter that I have. So I'm passing here monthlyWage, numberOfMonthsWorked, and bonus. But then instead of writing a set of curly braces, I'm going to use a very concise way. It's going to be a statement that returns an integer value. And before I write that statement, I use what is known as the fat arrow, and that consists out of an equal sign and then a greater than symbol. After that, I can write the code, and that will, in this case, be the calculation of the yearly wage. And I need to finish with, again, a semicolon. There we go. These are the same as using a full method body, including a set of curly braces. This is a concise way that you see often being used, and it's called, as said, expression-bodied syntax.

Introducing the Main Method

So far, we have written our initial code in the Program.cs file and the other newly created file. It is clear that the execution of our application starts in Program.cs. Now, why is that, you may be asking. If you look at the Program.cs here on the left, it's a pretty empty file, and we added our functional code, well, directly in there. However, in our Utilities.cs, there was some more ceremony going on that we mostly skipped so far. There's the namespace, the class, and quite a lot of curly braces. The reason that they are different is that since C# 9, in one file, we can use what is called top-level statements. When using top-level statements, all ceremony code is left out, allowing us to write our own code from line 1. Our own code is at the top level, and a file that uses this will be used to start our application, and that is why so far Program.cs was automatically used as the startup for our application. Our top-level statements weren't available before C# 9, and it is, in fact, only since C# 10 that they are fully used by

Visual Studio and .NET when creating a new application using the templates, like we used the console application template. Before that, the generated Program.cs contained what we see here on the right. You'll recognize a lot of the ceremony from our Utilities.cs fund. There's a Program class and the Main method. Indeed, one single method, Main, gets created by default. Now, both approaches work fine. You can use the "old" approach still with the Main method. Now, while the templates now generate what we are using in this course, you will probably when looking at existing applications encounter the Main method in the Program class. That's my main reason to explain it here. Otherwise, you may get confused. And the Main method was and is still, in fact, the entry point for our application when executing. C# will search for that method, and that is the code that gets executed when starting the application. Now I did say is, but we didn't create a main method, now, did we? Correct. Behind the scenes, I've been compiling our code. The code in our Program.cs actually gets wrapped inside of a Main method inside of a Program class. So, although we don't see it, it's actually still happening. The compiler is playing some tricks on us here. Like I mentioned earlier, the top-level statements are now the default. And that doesn't mean that you can't write the Main method yourself. If you create it yourself, the compiler won't wrap your code. Only one main method should exist in the end. Either we have the compiler created, or we create it ourselves, and then the compiler doesn't generate anything extra.

Demo: Exploring the Main Method

In the final demo of this module, we are going to explore the Main method in all its glory. I have now created a new application, again, a console application with Visual Studio. But this time I have selected .NET 5. So before .NET 6, this is what was generated. Here I can see the Program.cs, but now notice, apart from the Console.WriteLine("Hello World!"), it contains more code. The Main method I just mentioned in the slides is there. The Console.WriteLine is, in fact, inside this method. This method is inside the Program class. We'll understand classes very soon; don't worry about that. And then this code is all wrapped inside this namespace here, which allows us to group classes. Again, a concept we'll learn about later. Pre-.NET 6, so this is what we got. And in fact, without knowing, this ceremony code is still there. It is just hidden for us. And when launch our application. NET will search for a class program, and in there, the Main method, and that method is the entry point of our application. There can only be one of these. So the code we want to launch when starting our

application needs to go in there. And from there, we can launch the rest of the application. For example, also going to the Utilities class, if we want. Now, since .NET 6, the default template uses a C# 9 feature called top-level statements. Using top-level statements, we can omit this ceremony code, and that's what we see here corresponds with this. But the namespace, the Program, and the Main method signature aren't there. But behind the scenes, these are generated, and therefore, only one file in our application can have these top-level statements. When using this approach, the file using the top-level statements approach is the one that's used to launch the application, since it will get the Main method generated. And we will continue using what we have used so far, so top-level statements. Now, it's important to understand the rules of the game. If you use these, there can be only one file that contains them, and that file will become the entry point for our application. There will be an implicit name generator for you, and that, in fact, is the real entry point. Both options work fine, so if you should start working on an existing application that uses an explicit Main method, you can continue using that.

Summary

Methods are a very important aspect when writing C# code. Nearly all of our code that we write will live inside of a method. They allow us to make our code more manageable, since otherwise we would have a large block of code. Through methods, you will also be able to reuse a lot of the code. Methods will accept parameters. Through parameters, we can pass data from the caller of the method to the actual method. We then also looked at the Main method, which is now, let's say, implicit, but it is still the entry point for your application. Wow! We're again a lot smarter. In the next module, we will explore the many facets of texts and strings. See you there!

Working with Strings

Module Introduction

Text, the written word. What should we be without it? On average, we see many thousands of words on a daily basis; in a newspaper, on our phone, on the bus that drives by. And, of course, also in

applications, words and written text are very common. It's quite understandable that a major programming language, such as C#, also has a lot on board to help us work effectively in our applications with text. Welcome to this module titled Working with Strings, part of the C# Fundamentals course here on Pluralsight. Let us see why I have decided to dedicate an entire module to strings. We've touched on strings already briefly earlier in this course, so I'll start with a quick recap of what we have seen there. Once we have gone through the short introduction, we'll then start working with strings. We'll also look at how we can compare two strings. So, checking if two strings are equal. Finally, we'll take a look at how we can convert, or should I say, pause, a snippet of text into another type. Exciting stuff! Let's get started.

Demo: A Small Recap on Strings

First, a small recap on what we already know at this point about strings. We're going to continue using the setup that we already used in the previous module. So, I'm going to create in my Utilities class a small method to do a small recap on what we already saw on strings. We could define a new string using the string keyword that defines a variable of type string, and it's called firstName. The string value itself needs to be surrounded with double quotes. We can define a string and give it the value immediately, or we can define it and assign the value later on. Implicit typing also works with strings, so using var, I can also define a string here, and it also will be of type string. The string type defines a large number of functionalities and methods, such as ToLower. We can also define an empty string using a pair of double quotes with no content. And it is identical to calling string.empty. That is what we already saw in terms of strings before. Let's now learn a lot of new stuff around strings.

Working with Strings

We're now back on track with strings. So let's now start exploring what we can do with them. In C#, the string type has a lot of functionality built in. It has many members available for us to use, so built-in methods and other C# constructs, such as properties. Through members, we can use the data and functionalities exposed by the type. In other words, Microsoft has given the string type a lot of functionalities and behavior that we can use. Since a string is a bit of an array or a list, we can ask a string for its length. For that we have the Length member, and it's actually a property, but more of what

properties are later. Notice that I'm using the Length property on a string I've created called myString. The returned value we store in an int. Next, I can also ask C# to convert a string to all uppercase by taking a string and calling ToUpper onto it. And this is a method. We've seen the methods already. This is the built-in one, so we can use that here like we've done with the CalculateWage method, for example. What is the difference with properties, though? This one has brackets at the end. Similarly, we also have ToLower, which will take all characters and replace them with the lowercase version. I can also ask C# to check if a string contains a certain other string that we can pass in. Using the Contains method, again, indeed a method, I can ask if my string contains, for example, the string "Hello." The latter one, Hello I'm passing in as an argument. This will return true if the exact string can be found. You can also ask C# to replace a value in our string. Say that we want to replace all a's with b's. Well, then we'd use the replace method, passing in the value we want to replace, a, and second argument, b, is the value we want to replace it with. This will return a new string with a replaced value, and that I'm assigning to a new string s. In the last snippet, I'm asking C# to take a part of this trick. Now, like I said, strings behave like an array, and we'll talk about arrays later on in the course. But arrays just like everything in C#, is 0-based, meaning that we start counting at 0 instead of 1. I'm just asking here for the substring that starts with the second and ends with the fourth character. A frequently used operational strings is concatenating string variables. So, taking one string and attaching another one to it. Or another one. And another one. You get the idea. It is so common that there are actually different ways to perform this task. A first option is very simple, and probably the one you'll use most frequently. As you can see, I'm now using several concatenated strings and I'm including the variables in the coat. And I always have to close the string wire, the plus sign and then open it again. Now it does work. But there are definitely easy ways a first solution is using what is known as string format to string format are now passing or let's say original string as the first argument in this string. Though notice I now have zero and one between curly braces and I want to put the variable values. Secondly, I have the list of variables. I want to substitute the first one. So index zero will be replacing the zeroed parameter in the string. The second one will be placed where the one is we. Now the result is the same. Now there is another more elegant solution and it is called string interpolation. It works by adding a dollar sign in front of the string. First notice it is outside of the string. So not between the quotes then between curly braces, we now use the variable names instead of

zero. And one, we saw it string dot format at runtime C# will replace the placeholder with their value. And again, the result is the same

Demo: Working with Strings

We've already covered a lot of ground to learn about strings. Let's go and try this all out in Visual Studio. I'm going to show you how we can work with strings, perform concatenation, and string interpolation. The string type has a lot of functionality already built in. I'm going to show you a selection of commonly used functionalities here. I've already gone ahead and prepared this ManipulatingStrings method again in the Utilities class, and I'm going to use that to show your functionality on the string type. So I already have the first name and the last name. What if I want to create the concatenated version in a variable called fullName? I can create that concatenated version by just using the plus operator. I can do firstName, then I do a +, then I need to do a space, I can do a space using double quotes with just one space in between, and I can add the lastName. Again, as you could see, IntelliCode was already pretty clever, and it's suspected that that was what I wanted to do. I really like how this IntelliCode is helping me out writing my code. But we, of course, have to learn it ourselves, as this is a simple way to do string concatenation. Behind the scenes, this is wired to do String.Concat. That String.Concat, we can also use ourselves. Say that we want to create the employee identification, and we set it to String.Concat. And again, IntelliCode is smart enough to understand that I might want to concatenate first name and last name, so it will then do it. String.Concat is a method on the string type that concatenates all the arguments that I've passed it. Let's run this already. Now, I do need to call using manipulating strings from the Program.cs. So I'm going to call here Utilities.ManipulatingStrings. There we go. Let's put a breakpoint in here. I'm going to put the breakpoint this time on the closing curly brace so that I have an overview of all the code that has executed. So as you can see, fullName is the concatenation of firstName, space, and lastName. An employeeIdentification is just firstName, lastName without the space, because I didn't specify a space to be added there as well. Now, as said, the string type contains a lot of built-in functionality. Say that I want to create the empId, and that would be the first name. And when I do a dot, I will get an overview of all the available functionality on the string type. I can do, for example, a substring, I can do a count. What I want to do here is I want to do a ToLower. ToLower is a method built into the string type that gives me a lowercase version of my string. I may want to concatenate that with a dash, and then I also

may want to add a lastName ToLower. Now, if we look at these methods, ToLower, they will also return a string themselves. It is possible, therefore, to use what is known as method training. I can call a method directly on the value that is returned by another method. Let me show you. Say that I on the lastName may want to remove any leading or trailing spaces. I can do that using the trim method. Trim method is another built-in method that indeed says that it will remove all leading and trailing whitespace. That trim method returns a string, and I can't pass that string to ToLower. So what I'm doing here is training methods. If I want to figure out how long the empId is, I can do empId.Length. Now, ToLower had these brackets. Length does not. That is because Length is a property, and it will get me the number of characters in the string that are passing it. If you run this again, we'll see what empId looks like. I will also see that the Length is 13 characters. Now we're learning a lot about strings. Let's continue. I may want to check if a string contains another certain string. For example, here, fullName. I want to check if that contains beth lowercase or Beth uppercase. And if so, and I write something to the console. Next to checking if a string contains something, I also may want to take a part of a string. For that, I can use the subString method. Here, we have the fullName. I'm going to take the subString out of that fullName, starting at position 1, and the Length, the number of characters I want to take, is 3. Now, C# is 0-based, so that means that 1 here would actually be the 2nd character. So that means that I'm going to take characters 2, 3, and 4. Let's verify that that actually is correct. So first we see that it is Bethany. FullName did contain Beth. And then we took that subString, and you see that I'm returning here, eth, so I skipped the B, and then I took the three characters, eth. Now, finally, I want to show you a bit more about string interpolation. Another way to do string concatenation that we, in fact, already used in the previous demos when we used the string prefixed with the dollar sign, it's, in fact, my favorite way of concatenating strings, and let me show you it in detail. So if I to do string interpolation, I first type the $ before the set of double quotes. And then between these double quotes, I can just type a string. But then between curly braces, I can use variables, and these will be evaluated at runtime, and the returned value will be pasted in at that specific spot. So let's try one. I can do here firstName, so that's a variable, and I put it between curly braces. Now everything that is not between curly braces is just a string. So if I do a dash and then I do lastName, again, I want the value of lastName to be injected here, then I do lastName between curly braces, as you can see here. And I find this a very easy way to do string concatenation.

Using Escape Characters

The strings we have used so far are pretty basic. They start and end with a double quote. And in between, we just had some regular characters. But what if we want to include a new line, for example, or what if we want our string to include a double quote as part of the string? Surely that won't work with what we have looked at so far. For this purpose, C# strings can contain escape characters. It is, in fact, a combination of a backslash and a character, and together, they will have like a special meaning. Take a look at the string we have in this snippet here. I want to show part of the string on a new line, as I can achieve using an escape character, \n. Between the first name and the last name, there's a backlash key. That is another escape character. This one will add a Tab. In the demo, we'll use these and other escape characters. But first, one more handy trick with strings. Say that you want to represent a file path in the C# string. A file path itself contains backslashes, typically. Now, to represent a file path with backslashes, we therefore need to escape the backslash itself. And this we can do by adding a double backslash. The first one says, hey, now comes an escape character. And the second one is the one that's actually being escaped, so that one will be shown. And you see what such a string looks like here on the slide. Now, while this works, I don't think it's very easy to read. In general, strings that contain a lot of escape characters tend to have this. All the escaping makes it harder to read. There's a way to say to C#, well, for this string, just ignore any escape characters so that you don't see the backslash as the start of an escape character, you just see it as a regular character. That's where verbatim strings come in. If you prefix the string with an @ sign, C# won't see the backslashes as something special; it'll just ignore them. Take a look at the second line. Don't you agree, this just looks a lot cleaner. I am definitely a fan!

Demo: Using Escape Characters

All right, we have a few new things to try. Let's go back to Visual Studio and bring in escape characters, and then we'll also give verbatim strings a try. I've gone ahead and already prepared another method in utilities called UsingEscapeCharacters, in which we are going to play with escape characters, or maybe I should say escape sequences because they are often a sequence of multiple characters. I'm going to create a string, a greeting, where we welcome the user. So let's create the greeting. And I'm going to again use string interpolation. So let's first type "Welcome!" Then I want the name to be on a new line. And I can use for that an escape character. It's going to be \n. Then I want

to show the firstName. And then I'm going to use another escape sequence or escape character, \t, and it's going to insert a Tab between, in this case, the firstName and the lastName. So I want to give some special instructions like a new line or a Tab we use, in this case, an escape sequence. That is recognized by C# to be a special sequence. So let's write this to the console. Here we go. I have already in my program called UsingEscapeCharacters, so we should see our new string. There we go. We see that Bethany Smith is on a new line. There is, in fact, a Tab in between them, but you don't really see that on the console. So, escape sequences are recognized because they have that backslash in there. Now, what if we in our string also need to use a backslash? For example, say that we have an Excel file that we need to load into our application that contains all the information about employees. That file will be in a certain path, a certain directory that also contains backslashes. By default, they would be seen as escape characters, and that wouldn't work. So we need to do something special here. We basically need to escape the escape character, and we can do that as follows. So here I have a file path where the Excel file is in a certain directory. But notice I'm getting red squigglies under \d and \e because C# is saying those aren't correct escape sequences. It sees those as escape sequences. That's not I want. So what I need to do, I need to escape the escape character by adding a second backslash. So what I'm saying to C# is don't treat this backslash as an escape character; it is part of the actual string. And now the string to our path is valid. Now, a similar problem occurs when we need to have a quote in our string. Say that we have a marketing string, "Baking the best pies ever." That's of course, a tagline for Bethany's Pie Shop. In there, we have best pies between quotes. If I would use the quote, it would end the string. Not what I want. Therefore, I can also escape the quote using escape character, again, the backlash that goes in front of it. The final thing I want to show you is a verbatim string. Now, while this works, it might not always be readable. If you have a long file path, having all these double slashes in there might not be great for readability. We can, in fact use what is known as a verbatim string. If we put an @ sign in front of it, then escape characters are not considered anymore. So basically, now we can remove the double slashes here. And now the \d and \e that we had before are not considered to be an escape sequence. In many cases, when you have a string like this, making the string a verbatim string will improve readability.

Testing Strings for Equality

We're doing well on our trip to learn about strings. One thing that we haven't done is testing strings for equality. Let's look at that next. In C#, we can compare two strings for equality using the double equal site. Take a look at the snippet you see here on the slide. I have a string, Bethany, with a capital B. I can check in the second line if firstName == Bethany, again, with a capital. When we run this, this will return true, so we've used to double equals sign here. If we, in the third line, that is, comparing a lowercase bethany, now by default, this will return false. I say by default, since that is what C# will do. It will compare each character in the string, and when they are not exactly the same, it will return false. It's exactly the same in C# to use the Equals method on a string, passing in the string you want to compare against. So both methods are used and do, in fact, the exact same thing, they compare the value. Say that you now want to compare a string, but ignore the casing. There are again several ways of doing this, but there's one way that I typically use. What I'm going to do here is I'm going to make the two strings I want to compare all upper case. Then, no matter what the casing is, we're comparing the exact same value. Of course, ToLower w do the same thing.

Demo: Comparing Strings

Another demo coming up. Let's see how we can compare strings in C#. New demo, new method. I've now brought in UsingStringEquality, in which we are going to play with equality of strings. I have two names here, Bethany with a capital B, and name2, which contains BETHANY in all uppercase. First, I'm going to check if these two name strings are equal. I'm again using a double equal sign. That will return us a Boolean. Let's see the result. They are not equal, as you can see. When I check the equality of strings, C# will do a character-by-character comparison, and, of course, in that case, these are not the same. If I want to check if name is equal to Bethany, where I check the equality of name1 and Bethany, then I will get back true because indeed name1 was equal to Bethany. And instead of using the double equal sign, you can also use the equals method. As you can see here, on name2, I'm calling Equals, and then pass in the string I want to compare with. Name2 is all uppercase, so this will also return as a true. And you can see that here. Now, if you want to do a case-insensitive check, so we basically don't want to take capitalization into account, we can, for example, use a ToLower again. So that's what you see here. On name1, I'm calling ToLower, and I'm comparing that with the lowercase version of bethany. That will also return true. Of course, it would work the same with the ToUpper.

Parsing Strings from Other Types

When working with strings, we can actually parse a string value into another type. Let's take a look. Say that we allow the user to enter a value for the wage of an employee using our console application. The captured value will always be, indeed, a string. I will probably want to do some calculations with that entered value, so we need to create perhaps a double from the entered value. So we need to convert it from a string to a double. We're going to parse the string into a double value here. And this we do using the Parse method on double. If the value can be parsed and it can actually be wrapped into a double, which will now contain the double equivalent of the entered string. Now, of course, not only the double type offers this capability. For example, we can do the same thing on bool, as you can see here as well. It might be that the user has entered an incorrect value, the one that cannot be parsed, for example, into a bool. For that reason, we have the TryParse methods. So if we're unsure if the parsing will succeed, it's best to give it a try using TryParse and let C#, let's say, try the parsing first. If all goes well, the value will be available in the variable parsed to TryParse. So that will be the parameter b here. TryParse is definitely your safety net. If you call parse on a value that cannot be parsed, it will result in a runtime exception. That will not happen when using TryParse.

Demo: Parsing Strings

Last demo of this module. Let us play with parsing and also bring in support for TryParse. All right, so now we are going to parse some strings. I'm going to ask the user to enter the wage for the employee, and I'm going to read that from the console. And the Console.ReadLine is always going to return as a string. So we capture that indeed here in string wage. Let's say that I want to use this value in a calculation. I need to make it into an integer or double, depending on the type of value the user has entered. So let us for now create an int wageValue variable, and I'm going to use the int.Parse method, and I'm going to parse, indeed, the wage. The parse method on the integer type is a method that will take a string, and it will parse the value passed in into an integer value. Now, this will work as long as the user enters a numeric value. For example, 1000. But if user enters abc, well, then we'll get an error because C# won't be able to parse abc into an integer value. Therefore, we can ask C# to do a test to see if the parsing would actually work, and for that, we can use the TryParse method. Let's use that instead here. And using the TryParse approach is a little bit more complex. I'm going to define the wageValue again. But now, as you can see, I'm not using parse, I'm using TryParse, and I'm

passing in, again, the value of that string that was entered by the user. And if the parsing works, the value will be put in this wageValue variable. And I also pass in that variable wageValue needs to be defined, and it also requires that we use the out keyword here. And out keyword we'll talk about later in the course. If the parsing succeeds, well, then we'll write this to the console; otherwise, we'll say to the user that the parsing failed. Let's try this out. So first, we'll add 1000, and that, of course, succeeds. If you run the application again and we'll enter abc, then we'll get a parsing failed. Now, parsing also works for other types. See that user has entered the higher date as 12th of December 2022. That's, of course, also a string. We can then using the DateTime.Parse method parse that string into a DateTime. So if that string represents a valid date, a new DateTime will be created. And that's what I'm writing here to the console. Let's try this out. Of course, we still need to enter the wage, and indeed, we see that also the date parsing has succeeded here..

Summary

We're at the end of this module. I hope you've understood the importance of strings in C#. Nearly every application you'll write will work with them. The string type, therefore, also comes with a lot of functionality built in. For example, to concatenate and to parse strings into other types. We've already mentioned them a few times, but we haven't started looking at them in detail, classes and objects. In the next module, you'll learn about these oh-so-important concepts in C#.

Creating Classes and Objects

Module Introduction

C# is an object-oriented language. That means it is easy to represent structures in the language and let our code represent real-life models and their interactions. To do this, we most often use classes and instances of these, which are objects. Welcome to this module, where I will teach you how to create your first classes and objects. I'm still Gill Cleeren. I didn't think it would be a good idea for me

to change my name in the middle of a course. Important module alert. Indeed, creating classes and objects is a task that you'll be doing all the time when working with C#. So first, I want to give you a bit of background on classes. Then, throughout the rest of the module, we'll be creating the Employee class. Of course, creating it is one thing; we also need to use it. And that's what we'll do in the next part called Using objects. All right, this will be great fun, but as said, also really foundational. So, no time to waste!

Understanding Classes

So, let's understand classes first. So far, we have mostly used variables. Int a = 3. Well, that's a great first step, but just regular variables won't get us to the moon and back. They are pretty loose. We need a way to represent structures, models, or entities that we have in real life. What we will need is to bring together data and functionality on that data to represent the model, and for that, we'll have to create a custom type. Models or entities are used to represent an item. Typical examples include an employee, which is what we'll use later in this module. A customer is similar. It's a real-life model that we want to represent in code when writing an application that deals with customers. A message is perhaps a more abstract model. It can represent a piece of data going from one system to another. A transaction could be a model we want to represent when working with a banking application, and it's again a bit of a more abstract type of model. When creating a system, you'll be spending time thinking of the models or entities that are needed in the process you're building, and then you need to think of the data and the functionality this will need. After that, we'll create types, custom types in our C# application. Creating custom types is a very common thing in C#. In fact, almost all of our code will live inside of a class, the most commonly used type in C#. Classes are reference types, and we'll learn in the next module what this really means. Classes are, as said, very common. You've seen earlier that we have moved our code to a separate file, and that was also a class. We've also seen that in the Program.cs file, implicitly a Program class was created, which, in fact is called when the application executes. So most of the code that we have written and that we'll write in the next modules will live inside of a class. Next to the class, the struct is another custom type that we'll look at. It is very similar to a class, but has a few specific properties. Since C# 9, a new custom type was introduced, records. Internally, they are classes, but they have some specific advantages, and we'll look at them later, too. We are going to focus on the class in this module. So first, what's a class really, then? A class is a C# language

construct you'll use most often to represent a real-life concept or model, like a car, an employee, or dog, and so on. A class is really the blueprint of an object. Keep this sentence in mind when you're not familiar yet with the concept of object-oriented programming. A class is the blueprint of an object. So what does that mean? We will create classes that represent a concept, and we will instantiate an object of the class, which represents a single instance, really. Employee will be a class, but Bethany, Bob, and George will be objects. They are real employee instances. A class contains data to represent the structure, and it also defines methods which expose the functionality of the class. Creating classes is done using, well, the class keyword, as you may have guessed. We will soon see how this is used. And yes, classes really are the building block of just about anything in C#. Classes are really the foundation of object orientation, also known as OO. OO is about getting classes, and it's what we typically do in C#, and we will learn a lot more about object orientation in a later module. As already mentioned, all code we will typically write, or maybe I should say nearly all code to be correct, will live inside of a class. Indeed, in C#, you can't put a method just anywhere in your code. It has to be inside of a class. The only exception was the code that we saw in the program notes, yes. But remember, behind the scenes, a class was generated there for us as well. The same goes for a variable. I can't just place it randomly where I want. It needs to be inside of a class. Okay, what does the class look like, then? Good question. Let me show you its structure. First, we have the class definition using the class keyword followed by the name of the class. Here that would be MyClass. I've prefixed MyClass with an access modifier, here public. And that makes the class accessible from pretty much everywhere. We'll talk more about that later. Then, again, between a set of curly braces, I have the class body. In there, we'll have the code of our class. Then in one go, also creates the scope. When creating a field, which is like a and b here inside of the class, that field is available inside of the body of the class. Now inside the body of the class, we can write our codes, and we can use fields. I've mentioned the term fields before. They are basically class-level variables that will hold data. We will also include methods, which will contain a functionality, and they will work on this data. Later in the course, we will learn that our data will typically live in properties, which wrap the fields. But like I said, more on that later. Also, a class can contain events. We're not covering events in this course, though. I'm sure you now already understand what classes are about. Let us start going practical now and create one. I'm going to take you through the steps of creating our first class, the Employee class. In the context of BethanysPieShopHRM, our application will work with employees, probably. If you think

about an employee in real life, you will have a number of things that define them. An employee will have an identity. The name, of course, being probably the most difficult thing here. An employee will also have other attributes that define them, the age, their wage, and probably dozens more. Also, an employee will do things. They will hopefully do work, and they will get paid accordingly, I may hope. And again, I'm sure you can come up with many other behaviors, as well as attributes for an employee, but let's keep it simple for now. So let's apply what we have learned and create a class that represents the employee. The logical choice would be creating a class using the class keyword, and I think Employee would be a good name for the class. On the class, we'll need a way to store its data. For this, we will use fields, and I have mentioned before what they are. They are just variables on the class, and there we will store values. Let's first add a few fields. Here you can see that I've brought in fields that will contain the attributes of the Employee class. I've used camelCase again as the naming convention, meaning that we start with the name of the field with the lowercase, and then we capitalize each subsequent word. Notice for now also that I have added to these fields the public access modifier. That means that these fields will be accessible once we start creating objects. A class that contains just data probably won't be very useful. It's like having a car with a steering wheel and a poke, but no way to drive it. To add functionality next to our class, we'll add methods. Methods are used to perform actions, and they will typically do this on the stored data in the fields. The result is often a change in the state, meaning a change in the values of one or more fields. So what can an employee do? Well, perform work, as we said before. That's a functionality that we'll want to add to our class, so that will be a method. Here you can now see the method perform work, which was added. For now, this method is not returning anything, so its return type is set to void. Our method, too, received the public access modifier. Public will make this method usable from the outside. It means that when we will be using our class, this method can be invoked on objects of this type. It is part of the public interface of the class. With interface, I mean how our class can be used. Think back of the other analogy that we've used a couple of times already before, a car. A car will expose to you the functionality to drive. It is a public functionality. Private means that the method or field is only accessible from within the car. Never will you be able to interact with it from the outside. It is typically used for methods you don't want anyone to use directly. It is inner workings of the class. Again, thinking of the car, this would be something internal to the car. The car itself knows how to inject fuel into the engine, so that is a functionality, but it's typically not something you can call on to while

driving. Driving would become very hard to do so. Protected means that the functionality is available for the class and its inheritors. We'll learn about inheritance later in the course, and we will revisit the protected access modifier there.

Demo: Creating the Employee Class

All right, time to go and create our class. I'll go and create the class, bring in fields, and then we'll add methods. For the purpose of this demo, I'm going to start with a new application. We've already touched on many of the basic principles, but now we are going to start working on a real application, and that I'm going to do from a new project. So I'm going to go again to Visual Studio 2022, and I'm going to go here to the wizard to create a new project. So we'll select C# Console Application again, click on Next. I will name our application, again, BethanysPieShopHRM. And I have the option to select the version of the framework that I want to build my application with. And we'll again select .NET 8. So next, we click on Create, and our application template is executed, and we get, again, a new clean application that again starts with that "Hello, World!" that we already in the beginning. The Program.cs is again here. We'll leave it as is for now. We'll first focus on creating that new class, and once we have that, we'll use our class from the program here. So, let's go here and add a new class. We've done it before, and we created that utilities class, but now we're going to create a real class, a class with real functionality. So I'm going directly here on my project and select Add, New Item. You can also go to Add, Class here, and then I select class, and in here, I'll then give the name of my class. So I'm going to create the Employee class. It is typical that we place a new class in a new file. So let me create the Employee class here. So this will execute the template for the class. And as you can see, some code has been generated for me already. There are some using statements, and they are grayed out. And I'll explain why that is later on. Then we have a namespace. We'll talk about namespaces also later. But now let's focus on the class part. We see here that a class has been created for us. It is marked as Internal, and then we have the class keyword, then the name of the class. And now we have a pair of curly braces. And in between these, I'll write the code of my class. That will be the body of my class. Let us now focus on the creation of the Employee type, the Employee class. Now, what does an employee have? Well, they have a first name, a last name, they have an email. So that will be data that we need to store about the employee. So we're going to store that in variables on the class level. Those are actually fields. Let me create those first. Also note their

notation. Typically we'll use camelCasing. So, first name has a lowercase for the first word, and then all subsequent words like name will start with the capital. And there are some other things I need to store about the employee. I want to store also the number of hours they have worked. I want to also store the wage and the rate they get per hour, and that will go in this hourly rate. I also want to store the birthday for each employee. So I think we're done. We have all the data I want at this point to model my employee. Now, of course, just having data doesn't make it a real class. We sometimes have classes that just have fields that just will contain data, but we actually want to model an employee, and they will also have functionalities. So functionality we're going to model using methods. And will work with the data. Let us create our first method. On the method, I will also use an access modify. We'll talk about access modifiers later on what their meaning really is. Public basically means everyone can access it. The method I want to use here will be the PerformWork method. That's a regular method. We've seen those already quite a few times. It's not returning anything. It's a void method. It's called PerformWork, and it doesn't have any parameters for now, so we have an empty set of brackets. And between the curly braces, we can create the body. Now what happens when someone does work? Well, then the number of hours worked will increase. I must set it so that it increases with one, so we'll use a double plus operator here. I'm also, in order to see this happening, going to write something to the console. I'm going to use, again, string interpolations, saying that firstName, lastName has worked for the total numberOfHoursWorked. Now, one thing that is different is that this method isn't static. Why that is will become clear later in the course. But do note that I haven't added the static keyword here. Now, maybe we also want a version of PerformWork where we can actually pass in the number of hours worked. We are applying method overloading. Methods with the same name, but a different number or different types of parameters. So that's what I'm doing here. I'm going to pass in the number of hours, which will then increase the number of hours with that passed in number of hours. And that's what you see here. I'm now going to increase the number of hours worked with the value passed into the parameter, and we're going to also write something to the console. Let me add a few other methods on my employee. Hopefully my employee can also receive a wage. You see here the ReceiveWage method, which has a return type, double, and it also gets a parameter, resetHours, which defaults to true. That is an optional parameter. The wage is going to be the calculation of numberOfHoursWorked times the hourlyRate. I'm going to write something to the console. And if that parameter is true, I'm going to reset the number of hours worked to 0. I'm also

going to return the calculated wage. In the DisplayEmployeeDetails, we use again string interpolation using escape characters \n, \d to display the employee details. Now, going back to these two methods here, they are, in fact, very similar. You see that I'm going to increase here the numberOfHoursWorked with 1, and here I increase it with the numberOfHours that is passed in. And the rest of the method is, in fact, exactly the same. What I could do is let this method call the other method. I can, in fact, remove this line here and instead call PerformWork passing in 1. That would be the same thing, right? Because now this method invokes this method and it passes in 1. In that case, we should also remove this Console.WriteLine because otherwise we'll have the output twice, once from here and once from here. It's still something that I don't really like. I don't like to have fixed values like 1 directly in my code. I may want to make this a constant. So let's do that. We've already seen what a constant is. It's a value that cannot be changed. So I can go and create a constant here of type int minimalHoursWorkedUnit. That is then set to 1. Now, I can use that named value and use it from PerformWork. What I've now done is basically named 1, and I'm calling PerformWork with that value 1. So now we have a well-designed class. Let's do a build. That seems to work. We're getting some warnings. That's quite normal at this point. But everything seems to work fine. We have a class, but we're not doing anything with it. So let's change that in the next part.

Using Objects

We are now the proud owners of a class, a real custom type in C#, but we aren't doing anything with it yet. This module was called creating your first class and objects. So now is the time to start working with the class and start creating objects. A class is a blueprint of a concept, of a structure. It's like the plan of a house created by an architect. On the class, we have defined perhaps a field called Color, but that is on the class. Now we want to create off of that blueprint of that class if you will an instance. That instance is an object, and it can give a specific value to the color field, like orange. We can based on a class with multiple objects. So, a second instance or object would be the one where the value of color is blue. It's physically a different object. There are different spaces and memory. And we can continue doing that. I've added another object. It's important to see here the difference between classes and object. Classes are blueprints. Objects are the real instances, and they will give different values to the fields defined in the class. Creating objects is a very common task when writing C# applications. Objects are often referred to as instances, and you'll hear me use the words

interchangeably. Saying that we are creating an object is the same as saying we're creating an instance. And so that process is also often referred to as instantiation. So how do we create an instance or an object then? Since it's such a common thing to do, let's break down this process in a few steps so you are fully clear on this. We'll work with the new keyword, so as you can see here. And so we're seeing to the new keyword which class we want to create an object from, or in other words, which class I want to instantiate. Here I'm newing up an employee, since that's the type we are creating. In other words, saying new Employee will create a new object of type employee. Notice the brackets at the end. Between the brackets, now,I don't have any values defined like we did with methods, but as you will soon see, we'll often do that. So the pod on the right of the assignment operator has now created the object. Let's now focus on the part on the left of it. Now that will do what we have done already a few times, it will create a variable called employee. So the lowercase and the type of that variable will be employee, our newly created class. And what does this variable then contain? Well, with primitive types, like integers, the variable contains the value itself. Classes are reference types, and we'll learn about what reference types are in the next module. It means that the variable will contain a reference, a point, if you will, to the object. Once we have created the object, we can invoke members on it, which I will show you in just a minute. We have used the new keyword to generate a new instance of the Employee class. What we're actually using is called the constructor. It is a special type of method that is used to create an actual object. When we new up an instance, that will be done through the use of a constructor. And I'll show you how I can use them in a minute. C# will always use a constructor to generate a new object. And for that, it can use what is known as the default constructor. This means that if we do not define a constructor, it will generate one for us. Very often, though, we'll create our own, and I'll show you that as well in just a minute. Using the constructor, we will be able to pass initial values for the new instance. And let's take a look at them in some more depth. Here is our Employee class again. And as you can see, it looks as if I've added another method. But take a closer look. This one doesn't have a return type. And there's something else, which is special. Its name is the same as the class itself. Indeed, these are two characteristics of a constructor, no return type, and the name is identical to the class name. It can right here accept parameters, but that is not required, and it'll typically also have an access modifier specified. The highlighted code here is the constructor, and it is called to create the object and set some of its initial values. In the constructor body, you can see I'm using the constructor parameter values to set indeed

initial values for my fields firstName and age. How do we now call that constructor, then? Well, as said, the constructor is invoked when the object is created using the new keyword. Newing up an instance will always use a constructor. You don't call the constructor explicitly by a name or something like you do with a method. Based on the list of arguments we pass, the correct constructor will be invoked. Indeed, it's possible to overload the constructor. Multiple combinations of parameters can be used for this. Here you can see that I'm now using the new keyword to generate a new object of the employee type again. And to the constructor I'm now passing two arguments, the value for the firstName and for the age. Just to be sure, here's what the constructor in this case will be, let's say responsible for. From my Employee class, that contains the constructor. I'm actually creating new objects, and those are created using the constructor when I called new on that. Different objects are being created, so an object with the name Bethany, another one for George, and another one for Mohammad. Different objects, so different instances in-memory, span of the same class that offered this constructor. But wait, initially, we didn't have the constructor, and we were still able to create a new employee. I'm glad you noticed. Do we actually always need to include a constructor before a constructor of a given type can be created? Well, yes and no. Let me explain. There will, in fact, always be a constructor, but we don't have to create it. If we omit constructors all together for a type, C# will generate what is known as a default constructor, just basically the code you see here containing, well, pretty much nothing. It's also a parameterless constructor that doesn't contain any code, but it's actually what will be called when C# news up your object. You can also include it yourself and even include your own custom code inside of its body. Now, do note that C# will generate the default constructor only if we have no other constructors defined. Once you include just any constructor, so with or without parameters, C# won't generate it anymore. So it's not always the case that an empty parameterless constructor will be generated for you. This only happens if you have no other constructors defined in your class. A handy and sometimes used shorthand to instantiate an object is the one you see here on the slide. Notice what's different? The name of the type, so employee, isn't repeated on the right-hand side. We're just writing new and then the pair of brackets, optionally including the parameters, of course. It's definitely a bit shorter, but it depends on personal preference if you like this syntax more than, let's say, the complete one where you repeat the name of the class Employee on the right-hand side. Now, I'm a fan of being complete, so I tend to default to the full syntax, as you'll see throughout the demos. On objects, we can now invoke methods defined on

the type, as well as access its fields. Here we have our Employee instance again. Now I want to have our employee do something. We have defined methods for that on the type, so on the class, such as PerformWork using the dot operator. So by putting a dot behind the name of the variable, we can access its members. We have done this already a few times, even very early in this course when we used, for example, int.MaxValue. I hope you remember. That was the same thing. Using the dot notation, we get access to the methods and fields defined on the class. Methods are used to perform an action, such as PerformWork here. But I can also change the state of an object, and that's what I'm doing here, I'm setting the firstName to Bethany, again using the dot operator. In the last line here, I'm also invoking a method ReceiveWage, and that will return a value, which I then capture in the wage variable. One more thing, note that it's possible that depending on the access modifier used in the class, so public, private, or protected, some members actually won't be accessible to use. Only public ones are usable this way. With C# 12, a new alternative syntax was introduced called primary constructors, and I want to touch on them here briefly. Take a look at the snippet here and notice what's different. This time, there's a number of parameters defined directly on the class instead of on a constructor. These parameters are the same as with otherwise defined in a constructor we'd write ourselves, but now this is being done for us. We don't even need the write constructor anymore. I can now use these values inside of my class to name an age value as we could otherwise also do when we would write the constructor ourselves. Now I won't be using it in our demos here, but know that if you see this being used, it is basically a different concise syntax for doing the same thing.

Demo: Creating an Object

We have learned a lot about objects, so it's time to head back to Visual Studio. First, we will create a constructor, and then we'll instantiate an object. We'll again use the dot operator to then invoke members on that object. So we have our class that we created in the previous demo, but we're not doing anything with it just yet. So that will change right now. We're going to start creating objects. We are going to construct instances of our employee. So, constructing, that means that we'll need a constructor. But there is, in fact, already a constructor. Here it is. Can you see it? Well, neither can I, but there is, in fact, what is known as a default constructor. By default, C# will always generate a default constructor that it will use when we create a new object of our employee. If we do not bring in our own constructor, C# will create a default constructor, which is used to create an instance of our

Employee class, so to create a new object. So instead of letting C# create a default constructor, I'm going to create my own constructor. It will again start with an access modify. We'll use public for now. Then, I use the class name. Notice I do not have a return type. Constructors are similar to methods, but they do not have a return type. Their single purpose is creating a new instance of our type, of our Employee type, that is. Then we close it with a set of brackets again, and it will also have a body. What you see here is exactly the same as what a default constructor looks like. It doesn't initialize anything; it is just used to create a new instance. Typically, what we'll do in a constructor is setting up our new instance or new object, initializing it, so to speak. And so we typically will also accept a set of parameters that we can then use to initialize the fields, the data of our object. So let us convert this constructor into a real constructor that accepts a number of parameters. So I'm adding here to my constructor a number of parameters, one that will contain the firstName, the lastName, the email, the birth date, and the hourly rate. Now, just passing in those parameters doesn't do anything. I also need to assign the value of firstName to the value that's passed in. That will be first. I will do the same for the other fields. So now when a constructor is used, you will need to pass in values for all these parameters, and those we can then use to initialize the data fields of our object that we're creating. Now, just like with methods, constructors can also be overloaded. I can create multiple constructors which differ from one another by having a different set of parameters. Let me add another constructor here. What have I now done here? I've added another constructor, one that this time only has four parameters. Instead of also giving it a body that sets the data fields, I'm going to use another keyword here, the this keyword. Now, the this keyword is used to from this constructor call this constructor. It is basically calling another constructor, passing in the values for the parameters that this one requires. Notice that this one doesn't have a value for the rate, so we're just passing in 0. So now we're really ready with our blueprint, with our class. We can now create objects. We still haven't done that yet. So let's do that now. I'm going to go back to the Program class. And let us remove this default code here. I'm going to add some Console.WriteLine, and then we are really ready to create, to instantiate an employee. Let's do it now. I'm going to create a new employee. We have created the Employee class, so the type will be Employee. If you look at what IntelliSense is suggesting us, it doesn't seem to find the Employee. There was, however, a light bulb that appears that shows me a couple of quick actions. You can see here that it says we need to bring in a using statement. Indeed, a using statement is required, so let's add that. Now, why is that? Well, that is because employee is inside of a namespace,

BethanysPieShopHRM. Now, what is the namespace? We haven't talked about those yet. Well, namespaces are really groupings of classes. We'll talk about them later, but since Employee is inside of this grouping, inside of this namespace, we need to also make that grouping known here in the Program class. Again, we'll talk about this later on in much more detail. So just like we did with other variables, we have defined the type. Now we're going to specify the identifier. So let us create Bethany. And Bethany is the name of our variable, that's the identifier. And I'm going to assign it to be a new Employee. So now I'm going to use the constructor. As soon as I call new, behind the scenes, the constructor will be invoked. Now, our constructor required a number of parameters. As you can see here, we have two options. Either we use a constructor with four or one with five parameters. I have now defined a new employee, passing in a firstName, a lastName, email, a birth date, and the hourly rate. I'm using for that the constructor with five parameters. Let us just do a quick test. Is that default constructor is still there? That default constructor, the one without parameters, cannot be found. Indeed, Visual Studio says here, I cannot find the constructor that does not take 0 arguments. As soon as you create one other constructor, we've created two, in fact, the default constructor is not available anymore. So we can't use this one for now. We finally have created our first object of our Employee type. Now, there are a couple of things that I want to show you here. Of course, we can use var if we want. That's, of course, still valid. We can still use implicit typing. But there is another way of writing, of instantiating the initialization code. Notice this icon here below Employee. Visual Studio gives me another light bulb here that says, well, you can use this new, simplified way of initializing your object. Notice the difference here. We can, in fact, omit writing the type again on the right-hand side. This is, again, personal preference. I prefer to use the full-blown syntax, but you can make things a little bit simpler. Leave it as is for now. Now we're going to work with Bethany. We have an object. We can invoke methods on it. We can ask to display the employee details. Notice what I've done here. On the object, I've used the dot, and then I can invoke the methods on it. This is this method that I'm going to invoke. It's a void method, so I'm not expecting anything to come back. Next, we can also ask Bethany to perform some work. We had two overloads of PerformWork, one without and one with a parameter of type int. Let's use the one without for now. Let's ask Bethany to perform some more work, and we can use for this one, for example, 5 hours. Now, Bethany has done some work, so thank you, Bethany! We can now ask Bethany to receive a wage. That, if we think back of the Employee type, returned as a value, a double value. So let's use that now. So I'm going to create a variable of

time double receivedWageBethany, and I'm going to set that to bethany.ReceiveWage. And I do want to reset the hours. That is an optional parameter, so I can, in fact, omit it entirely, but we'll be explicit and I will make sure that we pass in a true as well. We'll write their wage paid to the console by using, again, string interpolation. With that out of the way, we have a fully functioning application that will create our object before methods on it and then also return a value from it. Let's run this now. There we go. The application has executed. Let's look at the output. First, the employee details are being shown. Then we have asked Bethany to perform some work, that did that 1 hour, that constant of 1, a couple of times. We also invoked it with 5, and then again with 1 hour. Then we called the ReceiveWage method. That also gave me some output, but also returned the value, and then also showing here on the console. I hope you start seeing the difference. The Employee class, the type contains a blueprint of the employee. And when I actually instantiate an employee, that's what I'm doing here using the constructor, I'm going to give values to these fields. The objects are using the actual values.

Demo: Working with Several Objects

So far, we have worked with a single object. Let's now work with several ones in the next and final demo of this module. So far, we worked with one object, Bethany, but we can create another employee, another instance of our newly created type. Let's create another employee. I'm going to create the employee George. Creating is very similar. In this case, I've now used that new notation where we do not repeat the type on the right-hand side. I can then also on George, on our second employee, invoke the DisplayEmployeeDetails. I can ask George to also perform some work. And finally, George can also receive a wage. I'm using the var, so implicit typing here again, but ReceivedWage will still be of type double. Let's execute things to make sure that everything still works. So we have the output from Bethany that we had before by creating a new employee, George, and he has done some work, and also received a wage. So far, I've used the methods to work with the values of our object. Indeed, if you think back of the PerformWork method, that worked with a numberOfHoursWorked, for example, to change data on our object. We can also directly change data on our object. For example, I can go here, and I can say that Bethany, the firstName, should be John. And that for Bethany, the hourlyRate is changed to 10. So using the dot notation, I can also directly access the fields, as you can see here. And that will change, that will set the data on my object. It will

override the other values. I can then invoke again my methods PerformWork and DisplayEmployeeDetails, so let's try that out. So we show first the employee details for Bethany. The object is still Bethany, but I've changed the first name to John. That's what you see here. From then on, the object's first name is John, and you see that here also in the output of PerformWork and ReceiveWage. In the other way around, we can also, for example, ask the firstName. We can do Bethany, ask what is the first name? That's what I'm doing here. I'm putting the bethany.firstName on the right-hand side now, and I'm getting the firstName value from my object. That's how we can work with multiple objects at the same time.

Summary

Wow, that was another exciting module! I'm happy you followed it, and you now know a lot more about classes and objects. Classes really are the cornerstone of C#, even this module created a class, and we've added fields to store data, and we have added methods to work on that data. Classes are the blueprint to create objects. And when creating objects, we saw that constructors are being used. In the next module, we will dive a bit deeper into how classes work, since they are what is known as reference types, and we'll learn about the difference between reference types and value types.

Understanding Value Types and Reference Types

Module Introduction

Types are important in the C# language. Right from the start of this course, we've been looking at the different types that come with the language. Types in C# can actually be split in two large groups, value types and reference types. Welcome to this module, Understanding Value Types and Reference Types where we will dive a lot deeper in what these are exactly, how they behave. And then, we'll also look at other options besides classes to create custom types in C#. This module will be a serious upgrade of your knowledge about working with types. So, fasten your seatbelt. We will start the

module with the main topic of this module, and that's understanding the difference between value types and reference types. Once we understand how reference types work and behave in C#, we will apply this in two places. First, we'll see that we can, using references, also pass data to methods. And next, we'll revisit strings once again and understand that these two are in effect reference types. That explains a lot of the, let's say, more special behavior of strings that you saw before. We'll finish this module by looking at two other custom types, the enumeration and the struct.

Value Types and Reference Types

So first things first. Let's understand the difference between value types and reference types in C# and .NET. In the previous modules we have already used a number of types. We have used some of the built-in primitive types, such as int, decimal, and datetime. And in the previous module, we have created our first custom type, the Employee class. In .NET, and thus also in C#, there are two categories of types, value types and reference types. These two categories basically behave differently. All types fall into one of these categories. We have, without really thinking about it at the time, worked with value types from very early in this course. All primitive types, like int, float, double, and char, are value types. Based on the selected type, the compiler will allocate memory for this on the stack. The type itself determines how much memory is needed to contain a value of that type. The actual value is placed in this memory location, so the value is directly stored in the memory location, hence the name value type. Let's see this inside of a diagram. Here we create a few integer values, a = 1 and b = 5. What's essentially happening here is that these variables are created on the stack, and they immediately contain their value. If we could look in the memory of our C# application, think of it like a box with the label a. And when you look in that box, it'll contain the value 1. Reference types are different. They are allocated on the heap. The heap is the mass memory of the machine. In this case, the value on the stack will not contain the actual value. It will contain a pointer to the memory address on the heap where the object is allocated. Indeed, what gets allocated on the heap is the object, so classes follow this pattern. Classes are indeed reference types, the most important ones, that is. Here's our diagram again. The actual object will be created on the heap, though the variable doesn't contain the actual object. Rather, it contains a reference to the actual object on the heap. If you have been working with C++, this will look very similar to a pointer, and that's exactly what it is. However, C# is a managed language, meaning that we don't need to clean up our objects manually to free up

memory space. Let me prove to you that classes are reference types by means of a small code example, and we'll use our employee again. I'm first creating a new instance, and then I'm setting the name. Employee1 is a variable pointing to an actual object created on the heap. I then create a second variable, employee2, and I set it equal to employee1. At this point we have two variables pointing to the same object, and that I can prove. If I now, through the employee2 variable, change the first name of our object and I set it to George and then I use employee1.firstName to get the value of our object's first name through the employee1 variable, we'll get back George. Why is that? Think about this for a minute. Maybe pause the video. In the next slide, I will show you a diagram that explains this in a bit more detail. What we have done is we have one object on the heap, and we have two variables pointing to that same object. When I thus change the field's value, it is changed on the actual object, which is allocated on the heap. Employee1 and employee2 are just variables that point to the address of our actual object, and it was the latter that was changed.

Demo: Working with Value and Reference Types

Let's return to Visual Studio and make sure that we understand value types and reference types in full before continuing. I'm going to continue using the application that we finished the previous module with. And as you can see, I've commented out most of our code. We'll need to come back to that code as well. Now I'm going to show you first the behavior of value types. I'm going to create a variable called a, and I'm going to give it a value of 42. Next, I'm going to create aCopy, and I'm going to set that to a. Now, since integers are value types, that means that a copy of the value of a, 42, will be created and will be placed in aCopy. We'll basically have now two variables with the same value. Let's verify that. I'm going to add the value of a and aCopy to the console. And indeed, both are now 42. But they are value types. They contain their value themselves. If I change aCopy to 100 and I do the same, I write them again to the console, we'll now see that a remains the same but the value of aCopy of a has changed to 100. In other words, there are two different variables, and each contains its own value. That's typical for value types. Let us uncomment bethany here. Bethany is an object, and Employee is a custom class so it is a reference type. And let me play around with that. Let us create a new Employee variable. Let's call it testEmployee, and let's also set it equal to bethany. Now I have two variables, bethany and testEmployee, and they're both pointing to the same object in memory. If I now, through the testEmployee reference, change the first name to Gill, what will now happen? Well

since they're pointing to the same object, we are, in fact, changing the value on the object on the heap. In other words, the two references pointing to the same object will display the same first name. Let's verify that. Let us do testEmployee.DisplayEmployeeDetails and bethany.DisplayEmployeeDetails. And we'll see that Gill is the first name in both cases because we're indeed working with the same object. It is changed on the object, and the two variables that point to it will give me back the same results of course.

Method Parameters

Now that we know about reference types and references, I would like to tackle two topics that would have been too complicated to understand without knowing about references. The first one has to do with passing data to methods. In C#, it is possible to pass parameters to methods in two ways. So far, we have actually used the default, and that is by value. The second option is passing parameters by reference, which will require us to use a new keyword, the ref keyword, on the parameters we want to pass this way. Let's explore these two options in a bit more detail. As said, the default way of passing is by value, and it is what we have been using so far without giving it too much attention really. With this approach when calling a method and passing it arguments to values of parameters, a copy is created for the method to work with. The method can do whatever wizardry it wants on these values. It is a working copy of the value, and thus the values in the caller aren't changed. They are not being touched. Again, let's see this in a diagram. Say that we have our AddTwoNumbers method again that we used earlier. And I've defined the variable a to be 33 and b to be 44. I am passing these to the AddTwoNumbers method. Essentially, a copy of these values is created, and that is passed to the method. If the method now changes, the passed-in values, so here we're adding 10 to the b parameter, the original value of b, so in the caller, is not influenced. It keeps its original value 44. The method works on a copy. That is passing by value. The second option to pass them is by reference. As the name gives away, we are not passing a copy. Instead, we're passing a reference to the original value, so the value that lives from where the method is invoked. No copy is made. That, of course, has a big influence on the behavior. If we now make a change to the parameter value in the called method, the original is changed as well since indeed we have passed a reference. For this to happen, we need to include the ref keyword, saying to C# we actually want to pass by reference. So passing by reference is definitely intentional. You'll want to do this when you want the method to actually work on

the original values. Here's our diagram again. Notice that I have now added the ref keyword in two places. In the method declaration, I have added ref before the int b. Now our method can still be called by passing in b but now as reference. And when doing this, so when we call the method, we now also need to specify the ref keyword over here. Executing the method, which does add 10 to the current value of b, will now have an effect on the value in the caller. Normally, a method can just return a single value. By using ref, it is possible to let a method work on different values and change the original values of the parameters. One more thing, before the method is invoked, all ref parameters must be initialized.

Demo: Passing Parameters by Value and by Reference

In the next demo, I'm going to show you the difference between calling by value and by ref. First, you'll see that our default way of working doesn't change the original values. So that is using by val. Then we use by ref parameters, and we'll see that this will change the values in the caller. So first, we'll test by val. In fact, we have already done that without paying any attention to it, but let me show it to you in a bit more detail. I'm going to go back to my Employee type, and I'm going to create a method on it to calculate the bonus. Let me paste in the method quickly. It's a very simple method as you'll see. In here, I calculate the bonus. I pass in an initial bonus. That is this parameter here. If the number of hours worked for that employee is more than 10, we'll multiply the bonus by 2. We'll write the bonus to the console, and we'll also return it. As you see, I'm passing in the parameter just as we've done before. It's just int bonus. Let us now use our employee. I'm going to first let Bethany perform some work, and we'll set it to more than 10. So I'll set it here to 25 so that the bonus is multiplied. We'll set a minimum bonus, and we'll set it to 100. That's a local variable at this point of type integer. The receivedBonus is set equal to the result of bethany.CalculateBonus. I'm passing in the minimumBonus. We'll then write the minimumBonus and the receivedBonus to the console. Now taking a look at the CalculateBonus, the value of bonus will be multiplied in here, and it will be returned. However, this value is coming from this parameter, minimumBonus. So let's see what now happens with that value. So that employee got a bonus of 200. That is the multiplied value that we have here on Employee. The value has indeed been changed. However, in the caller, the value has not changed. That is because minimumBonus was passed in, but a copy was passed in to the CalculateBonus. So we are using here by val. A copy is passed. I'm going to create another method

on Employee. This time, I'm going to create a method called CalculateBonusAndBonusTax. Notice now that I'm passing in again the bonus by file, but I'm passing in the bonusTax by ref. In other words, the value of bonusTax is a reference. It is shared. It is the same one between this method and its caller. If this method makes a change, it will also be visible in the caller. In the method body, I'm again multiplying the bonus by two. But then, if the bonus is greater than or equal to 200, I will set the bonusTax, which is equal to 1/10 of the bonus, and I deduct the bonusTax from the bonus. The bonusTax value is thus also changed in this method. I'm only returning the value of bonus. I'm going to call it in a very similar way. Notice I have, again, a value for minimum bonus and bonusTax, which is set to 0. I've initialized it to 0. ReceivedBonus is the result of just invoking on Bethany the CalculateBonusAndBonusText method, passing in the minimumBonus and, by reference, passing in the bonusTax. If the bonusTax is now changed in here, which it will be, that value will also be available here. Let's test it out. Bethany has began to perform some work. Then we call the methods. The employee got a bonus of 180, and the tax is 20. And more importantly, the value of bonusTax is also changed in the program, as you can see here. And the value was changed by the method that we evoked because we passed by reference.

Demo: Using out

Next to ref, there's another keyword in this area, and that is the out keyword. It's pretty similar in that it's also passing by reference. The ref keyword required us to ensure that the variable was initialized. Using out, that is not the case. We can actually pass in an uninitialized variable, and the requirement with out is that before the method exits, the out parameter must be initialized. Since this is also by reference, the original value, if any, can also be changed from within the method. While out works very similar to ref, you'll frequently use out when you want your method to return multiple values to the caller. By default, only a single value can be returned from a method. Using the out keyword, it becomes possible to return multiple values. And yes, that works with ref too. But remember that using ref, the parameters must be initialized. Let's head back to Visual Studio and see how the out keyword works exactly. I've gone ahead and created a variation of the CalculateBonusAndBonusTax method, which is now I have an out parameter in here. Out will do the same as ref. The only difference is that the argument does not need to be initialized in the caller. In the method, there's very little difference. The only difference is that I'm setting here the bonusTax inside of this method because before the

method exits, all out parameters need to have a value. I therefore need to set bonusTax because it could be that otherwise I will not initialize it. Let me show you. If I comment this one out, we'll get an error. As you can see, the out parameter must be assigned before we leave this method. So that's why I'm assigning it here in the method itself. But that value is also shared with the caller. I can now simply make a change here and to change the ref keyword here with out. And since I'm now using out, I don't need to specify a value here anymore, as you can see here. This will also work and it will also change the value of bonusTax in here in the caller. Let's verify that. There we go. The value of bonusTax was also received in the caller.

Strings Are Reference Types Too

I mentioned there were two things that I couldn't explain to you before we knew about reference types. The second one is what we'll look at now, the fact that the string type is, in fact, a reference type, too. Let's start again with a small snippet of code here. Say that we have a string that contains Hello. I then create a new string b and I set it equal to a. Then, in the next line, I add to b using +=, space, world. Then we ask to see what a is. The result is Hello. Can you figure out, based on what we already saw when I'm talking about reference types, what happened here? Strings are very foundational concepts in C#, but unlike int, float, and char, they are not value types. They're reference types. They point to the actual string in-memory, so on the heap. I'm going to show you in a diagram what has happened, and then this will already become a lot clearer. So first, you create a and b, and a will contain Hello. A points to the string Hello. That's important here. Now, we set b equal to a. B doesn't contain a copy; it just points to the same memory location as a. Now we create a new string, space, world. And then I set b to the concatenated version of what it currently contains, plus this new string. Notice what now happened. B, and b only, points to the newly created string. A doesn't know about all this. It is just still happily pointing to the original Hello string. That is because strings are reference types. They aren't created on the stack, they are created on the heap. That's another thing to notice here. When we created this new string, space world, and suffixed b with it, an entirely new string was created. The original wasn't changed. This is because strings are immutable. Once created, a string cannot be changed. Now, you may be thinking, hey, we saw earlier that we could, for example, replace a character within a string with a new one. Surely then we are changing the original string, aren't we? Well, I must disappoint you. Every operation you do on a string will result in a new one being created.

So also your replace operation will return a new string with the values replaced. Now, why am I telling you this? Is this important? Well, yes and no. No, because, in fact, C# handles all this. It'll return a new string for you, and if needed, it'll clean up the old string for you, too. But doesn't all this copying and creating of new strings have an impact on performance? Well, that would, or maybe I should say, could be the case. In regular circumstance, by no means, do you really need to worry. C# will handle this for you. But say you are doing some string operations inside of a loop or you're doing a lot of concatenation operations. This could result in a lot of strings being copied over and have a negative impact on performance for your application. Time to start worrying? Absolutely not. C# has a solution on board, and it's called the string builder. As its name gives away, it is used to build strings, but in a more performant way without all the copying. So if you find yourself in a situation where you need to concatenate a lot of strings for which each individual operation would result in a string being copied, the string builder will be worth its name in gold. Take a look at the snippet you see here on the slide. I'm creating a new stringBuilder using the new keyword. Indeed, that's a type, and we're using the constructor. Then on my new stringBuilder, I'm calling the Append method to append the passed in string. I'll do that a few times using AppendLine, which will append the string and passing in on a new line. Using the stringBuilder, none of this will result in the string being copied over multiple times, which would be the case for regular strings. Finally, I can ask the stringBuilder to give me back the actual string value, and we can do so using the ToString method on a stringBuilder instance. Very easy, isn't it?

Demo: Strings Are Reference Types

Time to head back to Visual Studio and make sure that we understand both string immutability and the StringBuilder. I'm going to show you the string immutability in the same Program class. It is not related to what we were building before, but let's do it quickly here. Now, without actually knowing, we have been working with string immutability or we have at least been encountering it already, well, since the very beginning, since we used string. When we were, for example, using ToLower, we were getting back a copy. When working with strings, we're always getting back a copy. The original is never changed. And let me show you that because that is string immutability. Let's create a few strings, let's create string name and set it to bethany, and let's create string anotherName, and let's set that equal to name. So I have two variables of type string and they're both pointing to Bethany in-memory.

Because strings are reference types, they are stored on the heap rather than on the stack. If I now do name +=, so I use the compound operator again, and I say, space, smith, for starters, this creates a new string in-memory, but then using the +=, so basically using the + operator here, is going to create a new string in-memory, which is the concatenated version of Bethany and Smith. Let's verify that. So I'm going to write to the console name and anotherName. Let us test this out. So as you can see, name contains now bethany smith, that is the concatenated version. And although we had another name point to name, it hasn't changed. It is still pointing to the original value name because that object hasn't changed. Name, however, did change. It is now pointing to a new string in-memory. Strings are reference types, and they are never changed. That is what you see here, and that is what we get with string immutability. Now I can show you string immutability also, for example, with the ToUpper method. I'm going to call ToUpper on name. Let us verify what name and upperCaseName will be. As we can see, name hasn't changed. That is still bethany smith. Although, I did call ToUpper on it. It hasn't changed. A new string, uppercase name, was returned, and that is what we see here. The original string again hasn't been changed. Now, as said in the slides, this will actually create a lot of strings. Basically, every change that you make on a string will create a new string. Now, that can actually result in a lot of strings being created. For example, take a look at the following code. I have an empty string here, indexes, and I'm going to create a loop that runs 2500 times, and I'm going to attach the counter to the value of indexes. Every time that I do this, a new index string is going to be created. In other words, 2500 copies of indexes are going to be created in-memory. That might not be an ideal situation. Although C# will clean up, we'll talk about garbage collection in the next module, it is not an ideal situation to use this approach. And it might actually be better to use the StringBuilder because using the StringBuilder, we are not creating new copies every time. We are appending to the original string. Let me show you the use of the StringBuilder. I'm going to create a new StringBuilder. As you can see, it can't find it. We need to actually bring in a namespace for this. I'll talk about namespaces in the next module. For now, just over here over the light bulb and click on using System.Text. We'll create our new StringBuilder. We'll use its constructor. There we go. And now the StringBuilder has quite a few methods to work with strings. And internally the StringBuilder works with a list which makes for better memory use instead of creating always a new copy. Say that you want to create a string, and I'm going to use for that Append. Append will let me append to the original string. I'm going to say here last name, and a colon and a space, and I do builder.Append again, and I use

here lastName. In fact, maybe I should use AppendLine here. Then I'm automatically going to include a line break. I'm going to do the same for the firstName. And when I'm ready appending strings, then I can actually get the final string by saying on the builder do a ToString, and that will get me back the complete string. This is in terms of memory use a much better approach. Now, for small concatenations, you won't feel this anyway, but if you have a lot of strings, it might make sense to use this. Again, talking about that loop, doing the loop this way will actually be a lot better. I'm again using the StringBuilder, I'm again using the loop, but now I'm going to append to the original string using the StringBuilder. And in terms of memory use, this is a lot better.

Working with Custom Types

Now that we know more about reference types and value types, let's dive a bit more into custom types. When talking about types in .NET, we can create different categories of types. We have already seen how to create a custom class, and that is by far the most commonly used custom type. But there are others. In total, there are five categories of types. So we have already worked with the class, but we also have the enumerations, struct, interfaces, and delegates. These different categories are in one of the two large buckets of types we have looked at before, value types and reference types. Enumerations, which I will cover in this module, and structs, also covered in this module, are value types. They will indeed contain their value and are created on the stack. The other three categories, classes, interfaces, and delegates, are reference types. When we used integers, bools, string, we were actually using predefined types. Those are types that come with .NET. Indeed, the .NET class library is itself a huge collection of types, but predefined value types and predefined reference types are included, quite a lot of them, really. Literally thousands. Next to the built-in ones, we can create our own. When we created the Employee class, we've created a user-defined type, and since that is a class, that will be a reference data type. In .NET Core, we have the base class library, aka the BCL. The BCL is a large library that we can use when writing C# applications. It contains a huge number of custom types, mostly classes that we can use from our own application code. The BCL offers all kinds of functionality already baked in so that we don't have to write it again. Going from working with files, drawing on the screen, looping over items in the list, connecting to a database, and much, much more. Since there are so many types in .NET, throwing them all in one big bucket would be quite a mess. Probably there would be name collisions, too. Multiple types within .NET could share the same name,

and by default, that is not allowed. That's why types are grouped in namespaces, another heavily used concept when working with C#. It's a way to organize types in groups. You can think of it like a folder with subfolders and the types live in the folders and subfolders. Because of this organization, we can have multiple types with the same name, as long as they live in a different namespace. Take a look at this hierarchy, since that is really how namespaces are organized. At the top, we have the System namespace. System itself contains types. Then there are other namespaces whose name starts with System. Then a dot, and then, for example, Web. In that namespace, there will be all kinds of types that have to do with wrap-related functionality. And the nesting can go deeper, a lot deeper, as we can see, for example, with the System.IO namespace. The types in .NET are placed in namespaces, but also our own custom types we'll place in namespaces. By using namespaces, we avoid naming collisions between different types with the same name. Of course, when using a type that lives in a given namespace, you do need to point out to .NET where it should look for the type. And we can do this by bringing in a using statement for the type's namespace at the top of the file. By using a namespace for one of the namespaces we just saw, System.IO, all types within that namespace become available for use in our code, including here, the FileInfo type. Since some using statements are so common, since .NET 6, we actually have what is known as global using. As the name implies, these are using statements which are globally set, so for the entire application, and as such, we don't need to include in our own code the using statements anymore. Upon compilation of our code, Visual Studio will generate a file that contains the code that you see here containing a list of global using statements. By making these global, it's not required that we add a using statement ourselves in our own code for the types in these namespaces. Adding it won't cause an error though. It is just not needed anymore.

Demo: Custom Types

Visual Studio has a handy tool to look at existing types called the object browser. It's easy to see what types exist and what namespaces they live in. I'll show you how we can use a type that lives in the base class library in our own code. And I'll show you how we can bring in a third-party library containing, again, custom types. I'll also show you the use of global using statements. The .NET class library is huge. There's a lot of functionality already built in that we can use in our applications. We've already used the System console, but there are so many things. It's actually rather difficult to get an

overview of everything that is in there. Of course, documentation that you've looked at before can help you with that, but also the object browser is a useful tool within Visual Studio to browse the class library. You can see it by going to View, and then Object Browser. Now, in this object browser, we have different views, and I actually switch here to namespaces by right-clicking here and say View Namespaces. Then we indeed see BethanysPieShopHRM showing here. And indeed, in there we have Employee, and we also can see everything that Employee offers, our fields, as well as our methods. But this we can also do for other types. For example, we have worked a lot with the System console. So why don't we expand on System, and now we browse to Console. And we see the many operations that we have available on the Console class. You indeed see here that Console is a static class that is part of System. You can click on the WriteLine. We've used the WriteLine passing in a string quite a few times. That would be this one. We indeed also see a summary, we see its parameters, and also possible exceptions that can be thrown by the WriteLine. I'll talk about exceptions by the end of this course. I want to show you also another one. Let us talk about collections. The System.Collections.Generic namespace contains classes that were used when working with lists of data. I will do that in one of the next modules as well. In fact, when I expand here on the System.Collections.Generic namespace, I'll see all the classes which are part of that namespace. Indeed, here we see the list of T. And we haven't talked about what this list of T is, but it is, in fact, a class built into the .NET that we can use to work with lists of items. And you can see that this, for example, supports an Add, a Clear, a Copy, and so on. So, typical things that we would do with the list. And once we have found the list, we can actually start using it in our application. For example, let me go back to the Employee and create a temporary method in here. Now in here, I will start using the list, for example. We'll create a list of strings. Again, this notation we haven't talked about. We'll do that later. I'll create a list of strings first. Now if you go back to the object browser, you will see here that the list of T is part of the System.Collections.Generic namespace. So technically, I would need to bring in a using statement. We didn't do that. Why is that? Ah, it was already there, but it also seems to be grayed out. In fact, I can click here and it says here remove the unnecessary usings. Isn't that contradictory because we do need the list here, which is part of the different namespace? And we don't need the using statement. When we, for example, bring in the StringBuilder, I do need to bring in the System.Text namespace with the using statement. Why is there a difference? Well, there is a difference because there is now something called global usings. If we

build our application and we go to the generated files and we navigate to obj, Debug, net8.0, you will find in there a file called BethanysPieShopHRM.GlobalUsings.g.cs. If I open this in Notepad, you'll see here some global using statements that Visual Studio will add automatically. GlobalUsings, as the name gives away, are generated automatically, and they are prefixed with global, making them globally available. We, in other words, don't need to bring in using statements for these namespaces. Since System.Collections.Generic is in there, we could just use List of string without bringing in an extra using statement. System.Text is not in there, and that was needed for the StringBuilder, so we needed to bring in a using statement ourselves in order to work with the StringBuilder. There is a lot of functionality already built into .NET, but of course, Microsoft cannot create everything. That is why we have libraries often created by the community available as open source packages. We can, in fact, bring in functionality written by someone else into our application. Say that we have an extra functionality for Bethany's Pie Shop where we also need to export an employee as a JSON string. JSON stands for JavaScript Object Notation. And it's a format that is often used for exchanging information between different systems. Say that we want to convert our employee in a JSON string. We could write that code ourselves, but surely someone has already done that. It is not a very unique functionality. We can right-click here on the project and bring in functionality written by someone else by going to the Package Manager built into Visual Studio, and that is NuGet. If you right-click here on the project and we select Manage NuGet Packages, we will open the NuGet Package Manager. At this point, it says we don't have any packages installed, but we can browse, and we can now search for JSON. In fact, you already see that with a package called Newtonsoft.Json, which is shown here as one of the most popular ones. You can click on that. You can read a description. I'm not going to go in too much details here, but I can install it into my application. Once this package is installed, you can go to Dependencies, and under Packages, you will find Newtonsoft.Json. Of course, there's documentation for this package, but I don't want to go too deep into it. What I want to show you here is I can extend my application with functionality written by someone else by the community. Now, I can use that package also by bringing in that functionality into my employee. Let us add a new method called ConvertToJson. What I want to do here is create a JSON representation of my current employee. I'm going to use for that a class which is part of Newtonsoft.Json. Now, I can actually go back to the object browser, clear my search here, and I can also see that Newtonsoft.Json is added here. I can start browsing what is in here. And one of the classes that is available here is JsonConvert.

And JsonConvert has functionality to serialize my object, my current object, that is. What I can do here is now use JsonConvert. By default, it cannot be found, so I'll again need to use my light bulb, or alternatively use Alt+Enter, and I'll bring in a using statement. Now I can use the SerializeObject method here, and Visual Studio is already suggesting me that I use this here. Now, what is this, in this case? Well, this is another keyword that points back to, let's say, the current object. In other words, I'm going to convert the current object, its data members, into a JSON string, and I'm passing that to the SerializeObject method. I will then return the JSON. Now I can go back to my Program class, and from here, I can now convert my employee to JSON. Let's comment out the code that we've written before in this module. We have Bethany here. I'm going to convert my object into JSON. For that, I'll use the ConvertToJson method, and that will write to the console. Let's take a look. And there we go, there is the JSON representation of Bethany. So what we saw was that we can easily bring in extra functionality already written by someone. We can go to NuGet, we can search for a package that does what we need, we can reference it, and it appears here as a dependency. And once we have done that, we can use it in our code by bringing in a using statement that will make the code that is in our package available to use in my class.

Creating Enumerations

We've already created the class, but we can also create other custom types. Let's learn about creating an enumeration next. So, what is an enumeration, really? Imagine in your code that you have a numeric value that you want to assign a name to, simply because the numeric value itself makes the code harder to read and understand. A good example would be an EmployeeType. Maybe in your application, you want to distinguish between employees of type 1 and EmployeeType 2. But what are these values, really? What meaning do they have? One and three doesn't ring a bell for me, but we can create an enumeration which then allows us to give a name to these different employee types. From then on, for example, we can use the manager enumeration value, part of the employee enumeration type, to refer to EmployeeType 1. We can create our own enumeration types, as we'll soon see. C# will store them internally as a value type. Indeed, they are, in fact, nothing more than an integer value with a name. And to create them, we will use the enum keyword. Here is a custom enumeration. It's the EmployeeType enumeration I was referring to just a minute ago. If we want to, in our case, refer to an EmployeeType, we can create an enumeration that contains one or more

enumeration members. Creating the enumeration is done using the enum keyword, followed by the name of our enumeration, and that would be EmployeeType here. The members are the different values, different options, let's say, for the type, and they are strings. Here I have defined sales, manager, research, and store manager. Behind the scenes, C# is actually assigning default values for the int type for these different enumeration members, starting with 0 and increasing with 1. So, in our enumeration, sales would actually get the value 0, manager 1, and so on.

Demo: Creating Enumerations

We're going to create our first enumeration. Then, I'll show you how you can actually use it in a useful way. And finally, we can also see how we can access the value that sits behind the enumeration number. And the purpose of this demo is updating our ReceiveWage method. We are going to change the calculation that is being used based on the type of employee, or it could be that in the database the type of employee is a number, an integer value like 1 or 2 in a column EmployeeType. Now, we could use that value, that an American integer value, also in code. But while that works, it's not very easy to work with numbers in code and start checking if the value is equal to 1 or 2. When you come back to this code after a couple of months, it will be hard to understand what that 1 or 2 or whatever value meant. For that, we'll use an enumeration, and an enumeration is basically nothing more than giving those values a name. It is, in fact, a custom type. So we'll create a custom type different than a class. In this case, we'll create an enumeration. So, let us do that next. I'm going to right-click here again on my project, and I'm going to go here to Add, New Item. Now, I will need to select class. There is no template by default available to create an enumeration. It's actually pretty simple to do that. So let us create an EmployeeType enumeration. So I'll just stay here EmployeeType. As before, it will create a class. I'm going to change that into enum. So I'm going to create an enum here, EmployeeType. Now, enumerations are pretty simple. We cannot add functionality on them. They are just named values that we can use in our code later on. And the enumeration values that I want to create are Sales. That is, for example, an EmployeeType that I have in my database. I also want one for Manager, I want another one for Research, and StoreManager. So I have four known EmployeeTypes. Those are representing a number, an integer value. By default, this corresponds to 0. This would actually be 1, that would be 2, and this one would be 3. So behind the scenes, C# lets us correspond with a number. If we actually have other numbers, we can also define them ourselves

here. But let us keep things simple here, and we'll use just the values that C# is giving them. So now we can actually make our employees of a certain EmployeeType. I can go back to my Employee class here, and I can now create a field of type EmployeeType. Let's call it employeeType, lowercase since it's a class variable. So that now becomes data that we store on our class. I can also change my constructors. I have used the constructors before to actually set all the values to initialize the values, and I'm going to do the same with that employeeType. Let me update my constructors. Notice now that my full constructor here uses an EmployeeType. It gets it in as a parameter, and it will set it to this local field, and the other one uses a default and sets that to StoreManager. Now, since this is a value, since this is a field that's my employee, I can now use this in my calculations. As mentioned, I want to change the ReceiveWage method so that it takes into account the EmployeeType. I'm going to replace this line here with a small if else check. If the EmployeeType is equal to manager, I'm going to add 25%. In other cases, I'll just leave the wage to be the numberOfHoursWorked times the hourlyRate. If we now go back to the program, which I cleaned up while you weren't looking, I do get some errors. I haven't done a good job, it seems. This one is complaining that it cannot find a value for the EmployeeType, which is a required parameter. So I need to go and change my call here. I need to pass in an EmployeeType. Let us make Bethany a manager. And this one used George, so I'm going to make George a researcher. There we go. And the code now compiles again. Let's now run the application. I'm going to actually put a breakpoint here and see what the EmployeeType would be. So we're executing the code now for Bethany, Bethany is a manager. So indeed, the code for manager is executed. That seems to be okay. Now we go here the second time for George, and George is in research. So then we will go into the else statement, and he is not getting that 25% extra.

Working with Structs

As the final topic for this module, let us now explore the struct in C#. The struct keyword is really already saying what the struct is going to be doing. It's going to allow us to create a custom data structure. Structs are really used for lightweight structures, so with limited amounts of data. Behind the scenes, a struct is a value type. We can instantiate a struct using the new keyword, just like we did with classes. A struct can contain data, but it can also contain methods, and even other members, and it is in that they are very similar to a class. But they have less options. While it's technically not correct, you could say that there are lightweight classes, as structs are created on the stack instead of the

heap, and it mostly makes sense if the class you are about to create is too simple to warrant the creation of a class. Here you can see a struct, and I have modeled here work performed by an employee as a struct. To create a struct, we write first the struct keyword, followed by the name we want to use. So here, WorkTask. The body of the struct is again between curly braces just like with the class, and inside the struct, I have now defined two fields, description and hours. In a struct, I can also add a method. This method adds functionality to the struct, and typically in the code inside the method, we will again work with the data of the structure.

Demo: Creating a Struct

In the final demo of this module, we will learn how we can create a struct and call it from our application. So let us now create a struct. A struct is typically used for simple situations, and it is also a value type, meaning it's taught on the stack rather than the heap, which happens with classes, so with reference types. So let us create a struct next. I'm going to go again to my project, right-click, and say Add, New Item. In here, again, there is no template for structs, but let us create a struct ourselves, and I will create a struct for a very simple representation of a task, a work task. So I'll call my struct WorkTask. So by default, again, it becomes a class, but I'll change it to be a struct. Just like with classes, we can store data on it. We can add a string for the description of the task and an integer value for the number of hours. Now just like classes, structs can also contain functions, methods, that is. That's what you see here. I have a PerformWorkTask method that simply outputs to the console that that task has been performed. Now with the struct created, we can also now start using it. Let's go back to our program. Let's now create a new WorkTask. Now, we can use the new but it's not really necessary, since it is a value type. We can just say that we create a WorkTask, and we can then access its fields. We can set the task.description to baking delicious pies, and we can set the number of hours for that to, for example, 3. So we then have to renew it because it is not requiring an object to be created on the heap. It is all created on a stack. That also means that I can now do task.PerformWorkTask. So without doing new, this will still work. Let's run this and verify. So here's the output Task Bake delicious pies of 3 hours has been performed. That is, this method had executed here. Now, to be honest, I use most of the time, even for simple data representations, a class. I'm not often using a struct.

Summary

There we go. Another important module done. We have covered in a lot of detail the difference between value types and reference types. We've also seen that our beloved strings are also reference types, which explains their behavior. .NET itself is a huge collection of types, and we can create, of course, our own. Most of the time, we use a class which is a reference type, but we've also seen how we can create enumerations and structs, and they are value types. Since types and mostly classes are so important in C#, in the next module, we will dive deeper in working with custom types.

Doing More with Classes and Custom Types

Module Introduction

Nearly all code that we write in C# will live inside a class or another custom type. Classes are the most important custom type in C#. In .NET, the base class library itself is built mostly out of types. And we will, when writing C#, be creating custom classes all the time. In this module, we will build on what we have learned so far around types. I'm Gill, and I will be guiding you through this module, too. So what will we be looking at in this module? Well, it's a full agenda again. First, we're going to learn about creating namespaces for the clauses we have in our application. We looked at using namespaces already briefly, but now I will show you how we can create our own. Next, we will look at static and what this is all about. Then, we'll discuss the null value, also an important aspect when working with classes. Then, and this is related to null, we'll learn what garbage collection is all about. It's a magical feature of .NET and C#, to be honest. I will also show you how we can use a class from another library. And finally, I will touch on records, a feature which was introduced with C# 9. Let's dive in!

Grouping Classes in Namespaces

Time to look at namespaces again, but now from a different angle. We're going to see how we can group our own classes and other types in namespaces. Now remember what we saw in the previous

module. The .NET base class library consists of many custom types, and to avoid naming collisions, these custom types are organized in a structure that resembles a folder structure. I say resembles because it's a virtual organization. Real folders aren't required. But it's an easy analogy to think of them. Namespaces are used to allow us to create multiple classes with the same name. Without namespaces, every type, so every class, should have a unique name for C# to uniquely be able to identify the class. Now using namespaces, only within a namespace the type name needs to be unique. .NET itself uses them all the time. We've already seen that in the previous module. But we as C# developers can and should organize our classes in custom namespaces. We can easily create a namespace. It is just a string, really, and once created, a class belongs to that namespace. From then on, the fully qualified name, so the namespace and the class name combined, identied the class, and that identification should be unique. If we then want to use a class that is part of a certain namespace, we can access that class by making its namespace available in our code. And for that, we're using the using statement, which will go at the top of our code. Here you can see that I have created a namespace. This is it, really. We just use the namespace keyword, and then we define the name of the namespace, which is just a string. Typically, you'll use the dot to indicate a sub-namespace. Visual Studio will typically create what is known as a root namespace, which is by default your application name. Then I've added HR, which is a custom namespace, and in there, I'll place all my classes that are related to the HR part of my application. The Employee class is now entirely wrapped within the namespace. It is surrounded by curly braces. Of course, more than one single class, so spread over multiple files, can be part of the same namespace. That is the whole idea here. As in C# 10, an alternative syntax was introduced called file-scoped namespaces. On the left, we see what we just discussed, and that is, let's say, the traditional approach, where the namespace is followed by a set of curly braces, which then results in the fact that all code between these now becomes part of that namespace. All code is invented because it is inside the namespace. Now, when we look at the snippet on the right, we can see a file-scoped namespace. Notice the semicolon at the end. Now, and that's where the name comes from, all code in this file automatically becomes part of this namespace. What we're now gaining is horizontal spacing, and perhaps also a less clutter. It's again just an alternative. They do exactly the same thing. In this course, I use a traditional approach going forward. It is just a personal preference which one you use.

Demo: Working with Namespaces

Let's head over to Visual Studio for the first time in this module and see how we can create a few more classes and group them in namespaces. Then we'll use these classes, and for that, we'll need to bring in the namespace using a using statement. As we can see here, our employee class that we've now been working with for a while is by default in a namespace called BethanysPieShopHRM. That is not a coincidence. If we go to the project properties, we will see indeed that there is a default namespace set, and it is set to BethanysPieShopHRM. That is a default. You can change that default, but it means that every new class that you put in the root of the project, that is, is going to get namespaces BethanysPieShopHRM. So that means that EmployeeType, WorkTask, and Employee are all in that namespace BethanysPieShopHRM. Program, however, is not. That is why we needed to bring in the BethanysPieShopHRM using statement here. That gave us access to the types that live within the BethanysPieShopHRM namespaces, so that is why that using statement was required here. Now, in most applications, we won't place all of our classes, or of all types, I should say, directly in the root of the project. We'll create some subfolders, and those typically correspond with the namespaces. Say that, for example, all the classes that we've made so far have to do with HR. We could very easily go here on a project, right-click, and add a new folder, and let's call that HR. The default with Visual Studio will do, and that's typically also what I follow, is that all the time I create within this HR folder will be in the namespace BethanysPieShopHRM.HR. so the concatenation of the root namespace with the name of the folder. And this can even include subfolders, and we just place a.dot in between them. If we just let Visual Studio create a new class in here, let's create a class called Test just to test this out, you will see that the namespace becomes BethanysPieShopHRM.HR. Let's remove this again because I don't want to clutter my code with anything I'm not going to be using. So what I'm now going to do is I'm going to clean up our application a little bit. I'm going to move the Employee. I'm going to pick it up and move it into this HR folder. And I'm going to do the same with EmployeeType and WorkTask because all these classes seem to have to do with HR-related functionality. Now just moving them didn't do any changes. If you look back at Employee, we'll still see that they are in the BethanysPieShopHRM namesspace. The namespace is really a grouping here. I want to therefore also reflect this grouping in code, and I'm going to add .HR here to let the namespace correspond to the folder. Again, it's not required to have them correspond with the folder; I tend to typically do that in most of my applications. I can do that manually here on the EmployeeType as well, and I can also do

that here on WorkTask. And while we're at it, we can also let Visual Studio do it automatically. As you can see here, we can click on the light bulb again. You can let the namespace be changed automatically by Visual Studio. If we go to the Program, we'll see a lot of red squigglies. Our employee and our EmployeeType can't be found anymore. And using BethanysPieShop.HRM is grayed out. So we can probably remove that. Now, Employee was in the BethanysPieShopHRM.HR namespaces, so I'll need to bring that namespace again with a using statement. So I can again go in here on the light bulb and bring in the using statement, and everybody is happy again. Now say that we also in our application are going to build some functionality around accounting. Well, then we can create another folder here, of course. Let's call that Accounting. And I'm going to bring in a few classes. After I brought in two other classes that have to do with accounting, Account and Customer, and they are in the BethanysPieShopHRM.Accounting namespace. If we go back to the program and we want to create a customer this time, we can again do what we did before. We can start typing Customer. By default, Visual Studio won't find it. We actually need to bring in the using statement. We've seen that. But what we also can do is fully qualify the type name. I can say here BethanysPieShopHRM.Accounting.Customer, so then this is the fully qualified name. The fully qualified name means that I'm going to include not only the type name, but also its namespace in the name. And I can also create a new variable of that type, and that's going to be a new customer. There we go. Most of the time you won't be doing this, but if you see this fully qualified name, it basically means that the using statement is not being used, and we name the class with its full name, including the namespace.

Introducing Static Data

Let us now discuss that static keyword that we have been seeing in quite a few modules already. You have now enough knowledge to understand what its function is. When we create a class, we're creating a blueprint, I hope you remember. We can on the class include a field such as the name on our Employee class. And we then create a few instances. These will use the blueprint. And each object, each instance, can give a different value to that field. As you can see here, we have three employees, each with a different value for the name. They don't share anything really in terms of data values. However, we can now also define on the class a member, such as a field, using static. Let me give you an example. Say that we plan on giving our employees a bonus, and that bonus will be

calculated based on a fixed percentage, so the same for all employees. If you would create the bonus percentage on each employee, the objects get this value and can change it themselves. Imagine being able to adjust your own bonus percentage. How great would that be? So that's why we can actually define the value using static. The value will not be instantiated. It's defined on the class level, and it is shared for all objects of the given type. Of course, we can still change the value, but in that case, it is changed for all objects. Defining a value as static basically means it's defined on the class level rather than the object level. So how do we make something static in code? Well, we can in our code create our bonus percentage field using static. At this point, we will now create an employee. It doesn't receive the bonus percentage field itself, since it is defined on the class, not on the object. An important side note, static fields can't be accessed through the objects. You always have to go through the class, and I'll show you how to do this in the demo. Creating static members isn't limited to creating a static field. Here you can see that I'm creating a static method. Now this is an interesting aspect. Why would I do that? Well, remember what I said about being static. It is defined on the class, not on the object. This means that this method itself is available to the class, so I can use it without instantiating an object of this class. That's exactly what you see happening here in this snippet. Here I'm calling the static method. Notice I'm calling it on Employee, so the class name, the type, without instantiating an object first. That is not what we have done so far. Normally we need to create an object first. When defining a static method, you call the method on the class directly. One more thing. The methods we created in our Utilities file, so the Utilities class, were all created static. Now, why did I do that? Well, we called into these methods from the Program.cs file, which contained top-level statements. But behind the scenes, there was a static main method, which was implicitly created by the C# compiler. Indeed, a static main method. So in order to call methods from a static method, these methods do need to be created using static, or we need to create an instance. Since we didn't know about creating instances at the time, I went around and need to explain and just made them static.

Demo: Using Static

Now, static can sometimes be a bit confusing, so let's head back to Visual Studio and see things in action. I'm going to show you how to add a static field, and also a static method, and then we'll use these static members. Making something static means it's basically shared between all objects that are created based off of this class. So the value is stored on the class level and not on the object level.

I'm going to show you static in combination with the taxRate. It wouldn't be very useful to have a taxRate to be stored directly on the Employee instances because it would be the same, I guess, for all employees, right? So that is why I'm going to create it on the class level and not on the object level, it's going to be shared that way. So again here in the EmployeeType, and I'm going to create a static variable. It's going to be the public static double taxRate, and I'm going to set it to 0.15. Now this tax rate I'm going to use inside of the ReceiveWage method. So let's go there. And in this ReceiveWage method, I'm going to make a change so that the tax rate is included. I'm going to paste in the code and take you through it. All right, so here's the updated Receive Wage. And as you can see, I now have a wageBeforeTax, which I set to default 0. If the EmployeeType is manager, we still do that 25% bonus. Otherwise, we do the regular calculation. And that now goes in the wageBeforeTax. Then I calculate the taxAmount, which multiplies the taxRate with the wageBeforeTax, and I deduct the taxAmount from the wageBeforeTax, and that will be the final wage that we are returning. This is now using from within the ReceiveWage method the static taxRate, as you can see. I'm also going to make a small change to the DisplayEmployeeDetails method. I'm going to in here also display the taxRate for a reason that will become clear in just a second. Let's now go to the Program. I've again cleaned it up while you weren't looking, and we still have Bethany and George, our employees, and nothing has changed to them really. I'm going to let my employees do a little bit of work, and then pay them, of course, for it. So as you can see on Bethany I invoke PerformWork a few times, and then I call ReceiveWage, and I do also display EmployeeDetails. And I do the same with George. By the way, notice what I've added here, a Region. I haven't seen that. A region is actually a way to collapse code. When I type #region and then a string, I can actually collapse this, and everything between region and #endregion will then be collapsed. It can make your code a bit easier to read. All right, back to where we were. Let us run the application now and take a look at the result. Now, before I run things, I'm going to go to my ReceiveWage method again. By the way, I can put my cursor here and then hit F12, and it will automatically navigate to the definition of the method. How handy is that? Now in here, I'm going to put a breakpoint so that we see what taxRate is going to be used for our different employees. All right. Time to run it. All right, we're here in the ReceiveWage method, and I'm calling ReceiveWage for Bethany. The taxRate is 0.15. That is a taxRate which is stored on the class level, but we're accessing it here in the ReceiveWage method, which is executing on our object. So that is the taxRate that we defined. If I hit F5, now I'm executing for George, we still see that the taxRate is 0.15, so it is

the taxRate on the class level that I'm using in this case. And it also means that if I change it, I also change it for everyone. If I now go back here and I want to change the taxRate, how do I now do that? Now, remember the taxRate is not defined on Bethany nor George, so not on the object, but on the class. So I cannot go and say bethany.taxRate. As you can see, it's unavailable. We could do bethany.firstName, bethany.lastName, and so on, but we can't do that with taxRate because it's defined as static. It's on the class level, so that means that I can actually use the class name.taxRate. That gives me access to the taxRate which is then shared across all objects, and I can set that to 0.02. Taxes have lowered. How great is that? If I now do the same and I again let Bethany perform some work and let George also perform some work, we'll see that the new taxRate is used for all objects. Let's run it again with the breakpoint still set. So, first one, tax rate is 0.15. For George, also 0.15. Now we are here, the taxRate is set to 0.02. And if you now look at Employee, you see that the taxRate is 0.02 for Bethany, and it's also 0.02 for George. Now, next to static data, we can also work with static methods. Static methods can, however, only work with static data. In here, I cannot do anything with, for example, the firstName because there might not even be an Employee, so how could I work with the firstName of an Employee if there is no Employee? So in a static method on a class, I can only work with static data, so the taxRate. Now, notice that now I have made my method static. It also gives me the ability to go back here and call Employee.DisplayTaxRate. Again, using the class name.the name of the study method, I can invoke the method on the class.

Working with Null

So far in our application, we haven't encountered null values. The concept of null, however, is very important, and it's something you'll come across, well, on a daily basis when working with C#. Let's understand null and see how we can work with it. You've seen this graphical representation before, showing the stack and the heap, two places used by C# to store values. Let's bring our Employee class into the picture again. I'm going to read the following, Employee employee. I've now created an Employee variable, but notice I haven't instantiated anything yet. The variable has been created on the stack, but since Employee is a reference type, it doesn't contain a value. It is null at this point. It's not pointing to an object on the heap just yet. Only when we have an object can we actually start calling methods onto the object. Once we have instantiated the class, a new object is created on the heap, and the Employee reference is now pointing to the actual object on the heap. When the actual object

has now been created, we can't call anything onto it. Of course, there's no object doing from the method on. The code you see here on the slide will actually compile, but it will give you a runtime error. I'm again creating a variable of the EmployeeType, and then I call before work on to that variable. Since there is no object, we'll get an exception while running the app. C# will give an exception. You'll see a null reference exception. Indeed, our reference to the object is null. We can't invoke anything on an object that's simply not there. As said, by default, a compiler won't give you a compile time error, and things will only go wrong when running the application. It's also possible, once we don't need to object any longer, to set the reference to null. What this will do is effectively breaking the link between the reference and the actual object. And the result is the same as not initializing the object. You can't invoke any method on the Employee anymore, since it's null. The object itself is probably not reachable anymore. It is target for garbage collection, which we will discuss after the following demo. Now I've talked about null in the context of reference type, so our class, Employee. But we haven't touched on null in the context of value types. Can these actually be null? Well, that's a great question. Now, think again of the difference between value types and reference types. Value types contain their data directly. Reference types point to the object on the heap, and those can be null. But, in fact, also value types can be null. Let me explain, C# supports this notion of nullable value types. A nullable value type basically represents the actual underlying type, plus an additional null value. Say that we want to create a variable of type int, but it could be that we don't have a value, so setting it to null would be what we need in this case. To a nullable value type, the int nullable value type, this is possible. To that C# know that we actually want one of these, we need to suffix the type here, int, with a question mark. Now we can both set our variable to an integer value or null. Now, a regular int would always have a value, but we're not sure about that anymore. And for that reason, on a nullable value type, we also have a few helpers, and you can see one here. I can now check to the use of HasValue if b is null or actually does have a value.

Demo: Using Null

In this demo, I'm going to show you how it can work with null at runtime, as well as nullable types. Now, before we can actually do something with a regular object, we need to have that instance created, right? Remember that we have on one hand the variable which is stored on the stack, but then we have also the object which is stored on the heap, and it's on that object that we can invoke

methods that work with the object's data. Now, before the object is instantiated, we can't do anything with it. And because of this, we can actually get the exceptions because we try to invoke something on something that doesn't exist. Let me show you. Let's now go back to the Program, and let's create a new employee, and I'm going to call him or her mysteryEmployee. And I say I'm going to create a new employee, but I'm not going to do that. I'm going to create a variable of type employee, but I'm not going to point it to an object. I'm not going to new it up. I want to set it instead to null. I don't really have to do this explicitly, but I'm just being clear saying that this employee reference is not pointing to anything just yet. If I now do mysteryEmployee.DisplayEmployeeDetails, we might actually run into an issue because the employee doesn't exist and I still wanted to do something. Let's run the application and see what happens. Very soon, the application will blow up. We'll get a NullReferenceException saying that the object reference is not set to an instance of an object. We have a reference, but it's not pointing to anything just yet, so we can't work with it yet. This is a very common exception to have in applications because we might end up with a reference that doesn't point to anything. The object might have been deleted already. That is one aspect of working with null. In some cases, we might want to be explicit that something in our code can actually be null. Maybe we load a list of employees from a database and the hourly rate might be null in the database. If we want to represent that in code, we can actually work with nullable types. Now, if we look back at the employee, say that the hourlyRate should be null. Now, we have double here, and double represents a double value. It cannot represent null by default. If I try to say here = null, it's not going to work. Because it is a primitive type, it cannot be null. It is actually stored on the stack. It's a value type. It cannot point to something on the heap that is null. We can, however, make hourlyRate nullable, and then it can contain every value that double can contain, plus null. Let me show you. Let us on the constructor start by making the rate that I'm passing in nullable. Making a primitive type nullable, I can do by adding a question mark. Now, this can contain every value that I just said, every value in double, plus null. So if the employee is being created and the data is coming perhaps from a database and the database hourlyRate is not known, then I'm going to pass in null. Now, on this line here, we're getting an error because hourlyRate is a double, and this can represent a double plus null. So we also need to make hourlyRate nullable. So I'm going to also go here and add a question mark, and now this is back in sync. Now hourlyRate can in our code also contain a null value, basically not contain a real double value. Now more errors appeared, as you can see here. Let's go here and try to fix those. We're getting an exception in the

ReceiveWage method now? Why is that? Well, in here, I'm again expecting wageBeforeTax to be, well, a double, but I'm using our nullable double inside of this calculation here inside of this expression. This is not accepted because on the right-hand side, I may end up with null, and on the left-hand side, I can't handle null. Now, what I should actually be doing is I should use, in this case, because this is now nullable, I should use .Value, and this will take the value from this nullable hourlyRate. And I need to do that here as well. Of course, it can contain null, so either I need to check if there is, in fact, an hourlyRate filled in. I can do that using hasValue, or what I can also do, and that's what I'll do here, I can actually use a default value. I'm going to introduce something here that we haven't seen yet, another operator, which is the null coalescing operator. If rate is null, I want you to use what is on the right-hand side of this double question mark. And that can then be, for example, 10. Let's say that 10 is then the default value. So this line effectively says set hourlyRate = rate. If rate, however, is null, then set it to 10. And that way, I'm also sure that rate will also be filled in and actually will have a value in my calculation of the ReceiveWage. Let me show you one more thing in the program. I can now, for example, also say here that George, instead of getting a value for the hourlyRate, gets null. Could be that this is not known, and we set it to null, and then in code, we will use that default of 10 here.

Understanding Garbage Collection

Our discussion about null was actually a great bridge to something called garbage collection. What is this all about then? When working with objects, we have the variable which points to the actual object. In a real application, a lot of objects will be created constantly. C# will keep track of the references for you. Of course, references will become null at some point. Maybe we explicitly set it to null, or the variable went out of scope. At this point, the object itself will still exist in-memory, but the link is broken. Now I said something that should actually worry you. The object still exists. Indeed, it's still sitting there in-memory, and this will actually happen frequently. Links will be broken, and over time, more and more objects will be sitting there in-memory without any reference pointing to them. As said, worrisome, because memory is not infinite, and if your application keeps doing this, the memory will be full of zombie objects. To solve this, C# comes with garbage collection. I say C#, but it's actually a .NET feature, part of the CLI, the runtime that hosts your application. Now, what will the garbage collector do? Well, it's a process that runs automatically when .NET, the CLI, in fact, deems it to be

necessary. The garbage collector will run through the memory and it will look at objects that belong to our application. If no reference is pointing to the object, it is considered unreachable and it will be removed from memory automatically. So the garbage collector is really what the name gives away. It will be cleaning up unreachable objects, throwing them away and clearing memory.

Demo: Using Garbage Collection

Let's look at garbage collection in the next demo. So, it's a process that you typically don't see. I'm going to show you a few things in this area. Showing you garbage collection is not that easy. It's an internal process, and it tends to not run that often also when you're using the debugger. But I'm going to show you something in Visual Studio called the diagnostic tools that allows me to give you a bit of an insight in the memory used by our application and also garbage collection that runs every now and then. I'm going to create first a large number of employees, and I'm going to use for that, again, a list of employees. We've touched on lists already. We'll see them in more detail when we look at areas very soon. It's going to be used just to store a large number of employees. You see here that I'm creating one of those lists, and then I'm going to loop 10 million times and create an employee and add that to the list of employees. Surely this is going to take a lot of memory, and you'll see that happening in just a minute. Now, normally we'll see the memory increase and increase and increase, and when that list goes out of scope, when the employees in that list are no longer referenced, garbage collection will at some point come by and clean all these references up. Now, I can try doing that, but you won't see it happening automatically. And what we can do is try to clear the list, which will basically remove all the items from the list, and I'm going to also specify that the list is actually null, which should, in fact, clear the list of all employees and basically set it to null. And all the objects, all the employees, should, in fact, become orphans, and they should be the target for garbage collection. Now, we can run the application. I'm going to put a breakpoint here so that I can hook into it before it actually starts. And I'm also going to put a Console.ReadLine so that I can also put a breakpoint over here. Thank you. All right, let's now start the process. And we are going to look at the diagnostic tools here. If you don't see the diagnostic tools, you can go to Debug, Windows, and then Show Diagnostic Tools. They will give us an insight. It is CPU load, as well as the memory load. As soon as we continue, you'll start seeing the memory going up because all these objects are being created. Let's go. See, there we go. We are hitting 1 GB of memory, 2 GB. I think we're going to end up somewhere

around 4 GB. There we go, so a very steep increase. And I have reached the end of the line, and still the memory is not cleared. Although I cleared the list of employees, and therefore all the objects in the employee list are orphans, they haven't been cleared by garbage collection. That is also because I'm running the application in debug mode, and that tends to trigger garbage collection at different points. Now, all these yellow marks here are actually runs of the garbage collector, unforced, as you can see. .NET was getting a bit worried when it saw the memory increase that quickly, so it tried to do a garbage collect. However, there were no objects to delete because they were all in that list at that point. Now what I can do to show you garbage collection is trigger it manually. I can say here after clearing the list GC.Collect. If I now run things again, you should actually see a drop in memory use because now I'm forcing garbage collection at the very end. There we go again. So we see the memory increase, and boom, you see that there is a steep drop in memory. We forced garbage collection. It saw all these 10 million employees sitting there in that list. The list was cleared, so the employees were not referenced anymore, and therefore they could be garbage collected, and garbage collection cleared up all that used memory.

Demo: Using a Class Library

Next, I'm going to show you how we can use a class library from our application. The C# and .NET ecosystems are vast. They have been around for a long time, and that means that a lot of great minds have already written a lot of code that can be reused to accomplish certain tasks. The .NET base class library is itself a huge collection of classes that we can use, and we've done that basically right from the start of this course. Even when our application had just one line, Console.WriteLine, we were already using a class part of the .NET class library. Using a class will often require bringing in the namespace that the class lives in using a using statement. And we can, of course, also reference or use code written in some other project or library. Maybe your company has written a library with common calculations. Say, for example, that another team at Bethany's Pie Shop has created the WageCalculation class, and that logic we can reuse in our application. No need to invent the wheel, in that case. Far from it! We could, of course, copy that code into our application, but a much better way would be that we reference the code. The external code will typically be packaged in a library. We can reference that library in our application, and once that is done, that code becomes usable from our application. So instead of writing everything from scratch, we can tap into the vast amount of already

created libraries. How cool is that? Back to Visual Studio, and we are going to learn how we can use a class in our application that is part of an external library that we will reference in our application. In the final demo of this module, we are going to use logic perhaps written by a different team at Bethany's Pie Shop. They created a class library, so a library without a UI, so another console application even with just logic at DLL. That's, in fact, an assembly, it contains compiled code, and they created that assembly and they made it available to us. And you can see it here. You can also get access to it. It is in the Assets folder of the m9 folder, so the module 9 folder. That contains logic that we are going to be using in our application. I want to go and do it similar to what we already did with the NuGet packages, but now I'm going to use a library, so a DLL. How do we now use that logic, that class library, in our code? Let's start by copying the path here. I'm going to right-click on my project and say Add, Project Reference. We're going to create a reference to a local DLL, in this case, a local assembly. And now we need to click on Browse, and once again, on Browse, and then I need to navigate to the path that my DLL lives in. That is this DLL. Let me click on Add, add on OK, making sure that this DLL is checked, and this will add under Dependencies an extra note here under Assemblies, BethanysPieShopHRM.Logic. We can, in fact, double-click it, and that should open the object browser we already saw, and I chose the BethanysPieShopHRM.Logic namespace, which contains the WageCalculations class, and it contains a constructor, and the ComplexWageCalculation method. That's a method that is probably documented by the other team, and it requires two doubles and two integer values. Assume that this is the logic that we are going to be using in our application written by another team. So, let's go back to our employee and let us now do another CalculateWage method. We already saw that the class was called WageCalculations, so I'm going to bring in an ins of the WageCalculations. By default, Visual Studio won't find it, but since we have already referenced that assembly, it will probably find it, and we can just bring in a using statement, and we can instantiate an object of that Calculations class. Then we can create a double, which is going to be the result of on the wageCalculations object that we just created, invoking the complex wageCalculation. That requires a few arguments, the wage, the tax rate, the country ID, and the age. And assume that that thing does all the complex calculations for us. So we're going to pass in the wage, the tax rate, country ID, let's put it to 3, and let's set the age to 42. And finally, we can return the calculated value. I actually seem to have made a typo and I wanted to have calculatedValue. If I want to rename this, everywhere where I'm using this, I can do CTRL+R+R and then I can just change the value to

calculatedValue, hit Apply, and that will rename my variable everywhere where it's being used. So by making a typo, we actually learned something new. All right, with this in place, we can now go back to Program, and over here we can on Bethany method call the CalculateWage method, which then calls into the complex WageCalculation method. This is how we can work with an external class library.

Introducing Records

In the final part of this module, I want to touch briefly on records. Since C# 9, the language comes with a new reference type called records. It's a reference type, like a class, and in some cases, it can be used instead of using a class. You'll see in just a minute that it's very concise to create records. It's much more compact than a class to create, and in that sense, it is sometimes easier to just use a record instead of a class. However, while we could replace our classes, it is not a goal to do. Records are most commonly used when the class contains just data. Now, do note that they can contain functionality such as methods, too, but if that's the case, we'd typically still use a class. So when your class is simple and contains just data, it could be beneficial to replace it with a record instead. Now just using it for this would be pretty useless. However, records come with some built-in extra functionality, which can come in handy. Indeed, some code will be generated when using a record instead of a class. Now, by the way, behind the scenes a record is a class. If you look at the generated code, it is a class with some extra code that is created automatically. Here you can see a record being created, not surprisingly using the record keyword. This becomes a record class. The second line, so record class, is the same thing as just using record, and if you use this, your record will become a reference type. It is possible to create a record struct, and that would create just as with a regular struct, a value type. As said, when we create a record, code will be generated for us behind the scenes that gives us some extra functionality for free, which I'll show you in just a minute. Now, this is the most simple type of records. Let us add onto that. Records can be created using a primary constructor, which you see here. In this case, next to the record definition, we add the parameter list. Here it is just one. We have already seen this syntax earlier, as it's also possible to use this on classes since C# 12. Now, what happens with records, though, behind the scenes is different, but I won't go into the details too much here. What is created this way is a positional record, meaning that it will be required to pass in these parameters when instantiating an object based on this record. Instantiating a record is done in a normal way, just like what we do when instantiating a class. Here you can see that

I'm now creating an account, passing in one single argument to create our object. Since this is the constructor, it is required that we pass in this argument. Now, you may be thinking, thanks, but this seems pretty useless, and I can agree with you. At least so far, this doesn't seem to be adding all that much, right? Now, records have their purpose, though. They are typically recommended to be used when your types contain just data, as I already mentioned. And additionally, they have the added ability, thanks to the generated code, to block any changes to this data. If you're new to programming, it might not be easy to understand why you'd want to block changes to members. However, there are definitely cases where you'd want to have data wrapped in an object to be fixed, and so no other code can make changes to it. And then records have this built in. Classes don't. So, immutability is probably the first benefit of using records. Another interesting feature that we get on records, thanks to the automatically generated code, is value-based equality. I'll show you this one next. Finally, and this I hope you have already seen yourself, is the conciseness of records. Creating a record is much more compact than creating a class, thanks to the concise index. Granted, Assets in C# 12 classes now also support primary constructors, but still they are very clean and compact to create. Here's how this immutability plays out. In the first line, I'm instantiating my record, passing in a value for the parameter. This is using that positional record again. However, as said, when using positional records, we get the added benefit of blocking changes to the first name here after it has been set. We know that this uses properties, more specifically, init-only properties, but diving into that would take us too far. I will talk about properties in a lot of detail in a later module. Don't worry about that for now. Let me try to explain the other benefit, value-based equality, in a bit more detail. For that, I need to talk about equality in classes first, which you may not have grasped from what we have covered so far. Imagine I'm just working with an Employee class and I'm creating an object, emp1. A reference is created, and an object is created on the heap, and the reference will point to that object. Next, I set the FirstName to Bethany. And I'll now do the same thing. I create a second employee, and the FirstName for that second object is also set to Bethany. And we ask C# if the two objects are the same, we get back false. Why is that, you may ask? Well, when working with classes, an quality check will compare the references and will give back true if the two references are pointing to the same object on the heap. Since we created two objects, we'll get back false. Let's now do the same thing, but using positional records. So you can see that I'm passing in Bethany. The result is, again, two objects on the heap, and two separate references pointing to them. However, when working with records, as said, thanks to the

generated code, if the records have the same members, so your first name and the values are the same, then we'll get back true if we do an equality check. This can come in very handy to verify if two objects have the same values. To be clear, we could do this with classes, too, but we would need to write that good ourselves to achieve this. With records, this comes out of the box.

Demo: Using Records

Let us return one final time to Visual Studio for this module, and I will show you the record type. Earlier in this module, we created this Account class, and yes, that we created indeed as a class. While there's absolutely nothing wrong with doing that, this is a candidate to use a record for instead. As you can see, this class is now pretty simple. It contains just data, and that is this account number here. I'm using a field, this one here, and a property. We'll learn a lot more about properties very soon. They are a very important concept in C#. And I can simply replace the class keyword here with a record, and things will continue to work fine as before. Let's do a compile, and yes, things continue to build. Now, we could also use a positional record. In this case, I'm using this primary constructor syntax here. Notice now I am putting parameters, in this case, just one, behind the class definition, and that is what creates a positional record. If we now head back to the Program.cs class, we can instantiate an account as follows. From the outside, it looks exactly the same to create an instance of a record class. I'm just passing in the arguments here. However, since I have now created a positional record, the value of the account number can't be changed after it has been set. If I try this, you'll see that Visual Studio gives me a compile error, saying that the value can't be changed after it has been initialized. This is where the immutability of records kicks in, and it can be handed to guard that no one is making changes to the data of an object after it has been created. Now, I won't go any deeper into records, as it is not something you'll use right from the start. In a later course, in the C# part, you'll learn much more about them.

Summary

We've reached the end of yet another module, and we've extended our knowledge on classes quite a bit. We've looked at namespaces, and we've learned that the main goal is grouping classes to avoid naming conflicts. We've then also see in static members. Making something static means it's at the

class level instead of at the object level, and we can thus access it through the class as well. We've also looked quite a lot at null. We've understood that unless an actual object is being referenced, we can't invoke methods or access properties. Otherwise, we'll get runtime exceptions. And we've also seen the garbage collection process, which cleans up unused objects for memory. And we've also covered how to reference code written by someone else that is wrapped in an external library. In the next module, we'll learn about working with arrays, and I hope to see you there, too.

Using Arrays and Lists

Module Introduction

We've covered a lot of ground already in this course, and you are well on your way to becoming a C# developer. So far, we have always worked with single items. We created a variable of type int or employee. But in many occasions, we will need to work with a set of items. For that, C# comes with support for arrays and lists. Let us see how we can work with them next. Arrays are pretty easy to understand. You'll see that very soon. We'll start with understanding how we can create them and access items within the array. Once we know the basics, we'll learn that using arrays, we can do some other interesting actions, like copying the array, and this is all supported straight out of the box. Now, while arrays are simple and can be enough to fulfill your needs, they might be somewhat limiting. The solution can be found in the collection classes, and that will be the final topic of this module. We will limit ourselves to the most commonly used one, and that is the list

Understanding Arrays

And so without any delay, let's start understanding arrays in C#. Many applications that you'll build are in need to work with sets of data. While creating this course, I was actually struggling to come up with a real-life application that I've built myself that did not require the use of lists. If you think of an HRM application like the one we are building for Bethany's Pie Shop, surely we come at a point where we are in need to work with a set of data. It could perhaps be a requirement that we loop through a set of all employees and that we calculate the wage for each employee. We'll need a structure in C# to store

all the employees. Maybe while you're following this awesome course, you have some background music playing using your favorite music streaming service. You probably have created a playlist. Well, that, too, is a set, a list. In the future, you'll probably write a C# application that connects with the database, and that will read in a set of records. As you hopefully understand, the number of times that you'll come across a list of data, well, it's pretty huge. But couldn't we just declare a number of variables to store all this data? Well, while that would work, it would be rather limiting. In many cases, the amount of items that we have in the list is, in many cases, unknown upfront. So declaring variables won't cut it. We'll need a structure that supports storing a set of items. And to support this, C# has in general two ways of handling sets of data. The one we'll look at here is arrays. And next to arrays C# also has a rich support for collections. Arrays, as we'll soon see, shine mostly when we are working with a fixed number of items in the set. Collections, on the other hand, offer a more flexible approach where the number of items can grow and shrink based on what is needed in the application. Arrays are a bit simpler to get started with, so let's look at these first. Arrays in C# are a built-in structure in the language, which allows us to store multiple variables. Indeed, in one single array, we'll be able to store a number of variables. When working with an array, these variables all need to be of the same type, and this type needs to be specified when creating the array. Items within an array can be accessed using an index. The index is the number in the sequence the item has. In C#, everything is 0-based, meaning that the first element can be accessed using index 0. Now creating an array requires a bit of a special syntax, so let C# understand that we, in fact, want to declare an array. As mentioned upon creation of the array, we must specify the type. This is the type of items that this array can contain. In the first line, that's again just int, but then notice that the type is followed by a set of square brackets. What I'm doing here is declaring that I'm going to create an integer array variable. The name of that variable is here, allEmployeeIds. Now, to be clear, adding the square brackets make C# understand that we have the intent to create an array, not a single variable of type int. Next to using just a primitive type, like int, we can declare arrays also of just any type, even custom ones. Here in the second line, I'm declaring an array of type DateTime in yes, the exact same way as we did for an integer array. StartDates is the name of the variable. So far we have, in fact, done nothing more than creating the variable. The array doesn't exist yet. Hey, we've heard that before. Doesn't that sound like instantiating a class, a reference type? Indeed. Great remark. Arrays are reference types. They are stored on the heap. And yes, that happens no matter what type of elements we are storing

inside of the array. So even our integer array will be a reference type, and it will be stored on the heap. AllEmployeeIds here in this sample is just a variable that points to the array. Now let's look at the right of the assignment. There we see indeed that I'm again using the new keyword. And then we specify the type. Again, followed by a set of square brackets, and in between these, we need to specify the size of the array. Here we have defined 4, which means that this array will be created so that it can contain 4 integer variables. It's only when we use this new int here that the memory will be allocated. So after executing this line of code, what will happen is that all elements are created and a default value will be used for initialization for each of these elements. So in essence, here 4 integer variables are created, and they all receive their default value of 0. Now just to be clear, arrays that we create are reference types, no matter what the type of the array will be. We created an integer array, and that will be created on the heap. If we create an employee array, as we'll do later, the exact same thing will happen. Only when we are calling on to the new keyword are we, in fact, creating the array, and all elements are initialized with their default value. If the type of the integer is int, the default 0, if we define an employee array, all elements will be initialized with null. Another important aspect for arrays is that their size is set when the array is created, and it cannot be changed. So if more elements need to be added to the array, after what has been set at its creation, the only option that we have is to create a new array and copy over all the elements. The original array cannot be changed. If that would be a requirement in your application, collections, which we mentioned already early in this module, might be a better fit. In C#, arrays are 0-based, meaning that the index for the first element is 0, then 1, and so on. Although the size of the array is fixed, it is possible to specify the length only at runtime. This length could be, for example, a value that we asked the user, or it could be a calculated value. You can see a snippet where I am asking the user using the console again to enter a numeric value, and that is then set as the size of the array. Now we now know how to create the array, but we don't have any data in it yet. At this point, all elements are still their default. There are multiple ways to do that. We can, in fact, upon creation of the array, already pass in values for each of the elements. In this case, we are looking again at the integer array, and now notice I'm passing in initialization values, integer values, that is, in between curly braces. I'm using constant values here, but these two could be a value which is only available at runtime, so a calculated value. The number of items that we pass in the initializer path, though, must be the same as what we are indicating as the length, so 4. For that reason, the second line here will cause a compiler error. Since there, we are still saying the array

should have a length of 4, but we're only passing 3 values in the initializer. Now, we can, however, omit the size value if we pass in an initializer, as you can see in the third line here. Now, of course, we won't always know the elements we want to store in the array upon the creation of the array. Now, we can, of course, access the elements also after initial creation of the array. And this we also specify a value for the individual elements. Through an indexer, we can access individual elements. In the first line in the snippet here, I'm setting the value for the element at index 0. Now, we can also access the values of an element, and that's what I'm doing in the second line here. We can again access the values through the use of an indexer. Here in this line, I'm passing 1, so we're trying to get the value of the second element in the array. That is, in our case, an integer. The value can be assigned to another integer variable. Now, do pay attention. As I mentioned, arrays have a fixed size. If we try to get the value of an index that doesn't exist, we'll get an exception, and this will be a runtime exception. The compiler doesn't know about it. I will see later how we can handle this through exception handling in our code.

Demo: Creating Arrays

Now that we understand arrays, let's return to Visual Studio and bring in arrays into our application. Let's add an array, and we'll also see how we can work with the individual elements. We'll also see that we can use loops easily in combination with arrays. Let's create a very simple first array. When I type first a type of array I want to create, it's going to be an array of integer value, so we simply type int, but then we suffix that with a set of square rackets. Now, this is an indication for C# that we're actually creating an array. Then we write the name of the array. So, let's say that we want to create a sample array, and I'll put sampleArray1. So this is the name of the reference, the name of the array, really. Then on the right-hand side of the assignment operator, we can write the keyword new again. And since this is created on the heap, it actually requires the new keyword. Then we again type int, and between the square brackets this time, we need to specify the length. Say that we create an area that contains 5 elements, we simply write between these square brackets, 5. And now we created an array which has 5 elements, and that's it. We cannot change it to have 6 or 7 or 8 elements afterwards. If that's what we want, we really need to create a new array and copy over the elements. So now we have created a very simple array of 5 elements. Another way of creating an array is doing the following. So let's create sampleArray2, and let us now in this case do new int, followed by again

the set of square brackets, but now I can also pass in the initial values, and this needs to go between curly braces, so a set of curly braces. Now I can, in a comma-separated list, specify the initial values for the elements inside of my array. So if I do 1, 2, 3 and so on, and now I have an array that contains these elements. Now notice that at this time, I haven't specified the length. If I would do the following, so let me copy this, and let us make this sampleArray3, and if I do specify the length, set it up with 5 here, and I'd include a 6th element, I will get a red squiggly, as you can see here. The compiler has noticed now that I'm creating an array of 5 elements, but I'm trying to put in 6, and it is, of course, not allowed. So this won't compile. Now, although an array is fixed in terms of size, once it's created, that size, that length, can be dynamic. I can do the following. Here I'm now asking the user how many IDs do you want to register? And I'm going to read that in a variable called length. And then I create my array at runtime with that length being passed in. So at runtime, we are going to decide how many elements can go in there. Now, once we have the elements inside of the array, we can actually access them, of course. Otherwise, it would be a rather useless element. Say that I want to, for example, get the value of the ID in the first position. That would be employeeIds[0]. Remember, C# is 0-based, so arrays 0-based as well. That means that the first element has the index 0. It also means that I cannot do the following. If the length is the number of elements I want to put in, and the array is 0-based, I of course cannot access an element at position being the length because say that that length is 5, we don't have an element at position 5 because it would be the 6th element. We don't get any compile time errors on that, but we will get a runtime error on that. Let's run the application, and we'll see what happens. And I'm going to put a breakpoint before I do so. So here's our array. If you hover over the array, by the way, in debug mode, we can also see the different elements at the different positions. You indeed see that we have position 0, 1, 2, 3, 4, and an array of length 5. Now we're going to ask the user how many elements they want to register. Say 5. And now you see that I'm getting an exception here because I'm trying to access an index which is out of range. So I'm getting an error that says that we're trying to access an element that is outside of the bounds of the erroring. Let's put this line in comment because it's going to continue to give us errors. Of course, we still need to get values inside of the array. So what I'm going to do is I'm going to create a for loop here. That for loop is also going to be 0-based. As you can see, my counter, my iterator is 0, and I let the loop run until the length is reached, but not including. I write to the console to enter an employeeId, I read it from the command line, and then I put that value inside of my array by using again the index. Once we have the elements

in there, we can also display all the elements again to the console. And I'm doing that here also by reading out the employeeIds, passing in my iterator. This is a nice example, so again of looking at the scope of the variables. Notice that I have two times i declared here, but two times they fall within this for loop in terms of scope. All right, let's run this again. So my application is going to ask me how many items I want to register. I enter ID 1, ID 2, ID 3, 4, and 5, and then I display all my elements. And do notice that I have here ID 1, 2, 3, 4, 5. That is because I used here i + 1 as the displayed value.

Demo: Working with an Array of Employee Objects

As we have seen, arrays can contain all types, going from primitive types to custom types. And let's see how we can use the array in combination with our previously created Employee type. In the previous demo, we used integers in our array, so we created an integer array that can contain only integer values. But we can also create an array that will contain elements of a custom type, say, Employees, if I have created a few employees using the constructor like we've done before. If I want to create an array out of this, well, I can do that in pretty much the same way, I can create an array like we've done before using the square brackets, we call that employees, and that will be a new array of employees. I've created seven employees. So I'll create an array of seven employees. There we go. Now I can add to that employee array in the first position, so at index 0, I can put bethany. And then we in the second position can put george, and so on and so forth. Now I've filled up my array. Again, like we did before, we placed all of our elements inside of the array. And now I can do, again, for example, a loop and call on each of these elements a certain method that we have defined on the Employee type. Why don't we try that out? So far, we've used the for loop, but let us use another loop, namely the foreach. We haven't seen that one yet. The foreach keyword allows me to create a loop that loops over all the elements in a collection, like the Employee array here. Behind the foreach keyword, I now use a different syntax. I don't have to create an iterator myself. I don't need to increase the count. It will just loop over all the elements in the array, in this case. I then specify that for each Employee in employees, e is here a local variable, and basically every time that we loop, e will point to a new employee in the array. So for each employee that I encounter in my array, I will, for example, display the details. Since that is an employee, we can call the DisplayEmployeeDetails on that. Now, while we're at it, and while I was learning some new things, let's continue doing that. Let me create a random value to randomly generate a number of hours. I'm creating a local variable,

numberOfHoursWorked, within the scope of the foreach loop, and I'm going to set that to a random value. Now, to create a random value, .NET has the Random class built in. It is a built-in type that I can use to create a random value. So I need to create actually a new instance of random, and then call Next, the next method on that, and that expects an integer value that would be the maximum random value I want to generate. In other words, we can, for example, specify here that we want a random value between 0 and 25 to be generated for us. This will give me back an integer, Random numberOfHoursWorked. Now I can call on my employee the PerformWork and pass in that random number of hours worked. And then on that employee, I can also call the ReceiveWage. There we go. If we now let this execute, we'll see that we have all the employees, and we'll loop over the array of employees, and for each we'll execute these methods. Let me comment out the code that we wrote in the previous demo. There we go. And let's run the application now. And here I have the output for each of the employees. I'm showing the details, I have a numberOfHoursWorked, and also a ReceiveWage executed for each of these.

Demo: Working with Arrays

The C# arrays also offer functionalities that we can use to perform more advanced interactions with arrays. Let's take a look at some of these. Once we have created an array, we have, in fact, created an instance of the built-in array type. And on arrays, we can then use built-in methods again. I'm sure you have an idea what the CopyTo method will do. Do remember that arrays are reference types, and thus are created on the heap. Creating a new reference to the same array in-memory doesn't create a new array. No. Instead, we'll end up with two references to the same array. If we really want to create a copy of the array, so the actual data in the array, then we can use the CopyTo method. Now, I'll show you how this works in the demo. The Sort method is another built-in method that we get for free on arrays. Now, Sort will work pretty much out of the box. If you're, for example, working with an integer array, C# knows how to sort integer values. They are sorted by default in an ascending order. But if you, for example, have created an array of employee instances, you'll need to include code that lets C# know how to perform this sorting. The reverse method will, as the name implies, reverse the order of the elements inside the array. Finally, we have a length, and that is a property, and we haven't talked about properties. That'll be for the next module. For now, just know that using length, we can get the number of elements in the array. Let's return to Visual Studio and learn more about these

built-in functionalities of arrays. Each array that we create is, in fact, an instance of the array class, and that comes with a lot of functionality built in. So as soon as we create our own arrays, we can use that functionality. Now, for simplicity's sake, I'm going to use again the array that we created earlier in this module, so the one with IDs, and that allowed me to enter IDs. We've already written this code together earlier. Now, with this array, we can start playing a little bit. We have already written the array to the console. Now, I can use a method, the Array.Sort method, and pass in my employeeIds array. And that's what you see here. What now happens is automatically my employeeIds array, so containing integer values, will be sorted in an ascending order. This works because .NET knows how to sort integers. If I want to sort, for example, employees, then I will need to write custom sorting code that knows which value to sort by. For example, alphabetically on the last name. But let us focus on what we are doing here. I'm going to sort my employeeIds in an ascending order, and I'm going to write that again to the console. So I can simply copy this code here and paste it in here, and I will now output my sorted employeeIds. Let us write to the console where we are showing the sorted values. So let us enter five IDs. I'm going to enter them in a non-sorted way. So first, 99, then 1, then 55, then 84, and then 15. The first list shows the arrays in the order that I entered them. The second one shows them in a sorted way. So this is built-in. We don't have to do anything for that if you want to sort integer values in our array. Another functionality that is built-in is copying an array to another array. If you want to do that, we'll have the CopyTo method, but we also need to provide the array, the target area, let's say, where we want to copy to. So I'm going to create a new array, employeeIdsCopy, with the same length, and I'm going to ask to do the copy. Now, this is not a static method on array. I now need to take my original array, employeeIds, and I need to ask to copy that to employeeIdsCopy. The second parameter that appears here is basically saying at which element I want to start. And I want to copy the entire array, so we start at position 0. At this point, I now have two arrays. There are two different objects on the heap, and I have two arrays on the heap. There are two different copies. They are not related to each other anymore. That means that I can, for example, now also use Array.Reverse. I want to, for example, reverse the original employeeIds. That's another built-in method we get on arrays, and it will simply reverse the order of the elements within my array. And to show you that there are really two different copies, let us now output the original employeeIds, as well as the copy. So here I'm showing the reversed original array, and this one is going to output the copy of the array. So let's again enter some values. So the sorted array appears. That does its work on the original array. Then

we reverse the original array, and the copy still contains the elements in the original order, so that was the sorted order.

Working with Collections

Now that we know the basic way of working with lists of data in C#, arrays, let's learn more about collections. In my opinion, you'll use these most of the time in your C# applications. So we've now seen arrays, and we've seen that they can contain lists of pretty much all types of data, like integers or employees. But I hope that you agree that they do have their limitations. The biggest limitation is that we need to specify the size upon creation. They're not flexible. And when we need more space, we basically need to copy everything over manually. Also, the way to access elements and add new elements is somewhat limited. But these shortcomings aren't there anymore where we can use collections. They will often prove to be a better solution. Collections are again a .NET feature. There are classes built into .NET. The list is the most commonly used one, and it's the one we will use here, too. Here is the first instance of the List class, so one of the built-in collection classes. For a large part, this looks familiar already. We have the List class, and using the constructor, we are creating a new one. But hey, there's something special going on here. Behind List, now see two angle brackets, and in between there's a type, int. Now, what's that all about then? Well, I'm glad you noticed. The collection classes and thus also the list are what is known as generic classes. The in between angle brackets is a type parameter, which boils down to the fact that we are specifying that we are going to create a list of integer values. Indeed, upon creation of the list, we are going to say it can contain integers, only integers, in fact. If you would try to add something else, say, an employee, we'll get a compile time error. Generic classes help us with type safety. Upon compile time, the C# compiler will check what we are adding. That means that also when we are reading out values from the list, we know for sure it will be an integer value that we're getting back in this case. One other important thing to note here. I haven't specified any size. Indeed, the list will size dynamically. When we add items, the list will have all functionality built in to ensure it can grow. So, not our problem anymore. Once our list is defined, we can use the built-in methods on the List class to add or remove items. Here, I'm doing exactly that. I'm adding a few integer employeeIds by using the Add method, and I can also remove an item again using the Remove method. Now, although we don't need to worry about the size of our collection of items, we can ask how many items there are currently in the list. For that, there is the

Count property, which is built in. I do want to emphasize once again the type-safety aspect here. With Lists, and, in fact, with other generic collection classes we're getting, we'll be notified about any errors we're making against the type-safety. Here you see again are a list of employeeIds, which is defined with the integer type parameter, so the one between the angle brackets. Because this int is there, the compiler will ensure that we're getting compile time errors if we try to add, for example, say, an employee. As said, when we later read out the items from the list, we can be sure that the value we're getting back is, in fact, an integer.

Demo: Using the List<T>

In the next final demo of this module, I want to show you how we can work with the generic list. And let me in this demo show you a generic list, one of the built-in collection classes. The list, as you can see here in IntelliSense, gets these angled brackets behind it. It is the generic List of T, meaning that it is going to work with different types. It can be used to contain a list of integers, can contain a list of employees, it can contain whatever we want, but we need to specify it as a type parameter. I'm going to work here with a list of integer values. And because of the characteristics of a generic type, this will now be typed in integer. In other words, it can only contain integer values. That is type-safety playing again. Now in my list, I can only place integer values, and at compile time, this will be validated. So let me create a list of integers like I was already doing. I'm going to call that employeeIds, and that is going to be a new List of int. So this is now creating a new collection, new list, of type integers. And I need to new this up using a constructor. That's why I'm using a new keyword here. And we also need a set of brackets at the end. And the big advantage of using a collection class is that we don't need to specify the initial length like we did with an array. I don't need to specify a length at all. Behind the scenes, the list is going to see when it's full. It's going to copy the items over to a new location, but we are not bonded with that. We're not limited in the number of items that we can add. This is all handled by the list itself. The list also comes with a lot of methods built in that allow me to work with the list. For example, adding items, removing items, and so on. So there's a very extensive set of methods, a very big API that is built around generic lists. If I want to bring in my IDs like we did before, we can do employeeIds., and then you can see the list of methods that is supported here. And we'll take for now the Add method, and I'll add my first ID, 55, there we go. Now I've added a first integer to my list. I can add, like I said, as many as I want. No way do I need to watch out that my list can be full; it will handle

that behind the scenes. And the type-safety that we're getting because of generic lists is also very useful. Now, I cannot make the mistake by adding to employeeIds a string, for example. Let's try adding a string, test, and after a few seconds here, we'll get a red squiggly. If we hover over that, we'll get an exception saying that we cannot convert string into an integer. Of course not, because there's no implicit conversion that exists here. So the compiler will already flag that we're making errors. We can only add integers to this list. And that will help us quite a lot because we cannot by mistake add an object of the incorrect type to my list. Now, as said, the list contains a lot of functionality already built in. For example, I can do a Boolean check to see if my employeeIds list contains a certain value. Say that I want to check if the value 78 is there. And then I can do that using the contains method. And if that's the case, we can write something to the console. If you want to check how many items there are in there, we can do that using the employeeIds.Count. The Count is a property. There's no brackets. We'll talk about properties in the next module. But this will get me back the number of elements that are in my list. If I, for example, would need my employeeIds in the form of an array, we can use the ToArray method. The ToArray method will convert this into an array of integer values. And if I want to just clear, remove all the elements again, well, then we can on the employeeIds do a clear, and that will remove all the elements and basically put it back at 0 elements. Now, to do as before, and just ask the user how many IDs they want to register. And I'm going to use exactly the same code as we did before, but now I can just say to employeeIds use the Add method to add the ID that the user has entered here. There we go. It is as simple as that. And I can keep on doing that without running out of place. And, of course, generic list can also be used with other types. We just need to specify the type upon creation of the list. Say I want to create a list using employees. I can do that. Let's bring our list of employees in again. There we go. If I now want to create a list of employees, I can do that. List of Employee. See now, I've replaced int with employees. Now this list can only contain employee instances. Let's call that employees, and it's going to be a new list of employees. In that list, I can now say, for example, Add, well, let's add george. At this point, george is at position 0. Now I can also use employees.Insert, and I can then use another element and an index to position that element at a specified index, and all the other elements will move up an index. So say that I am now going to insert at position 0 bethany. Well, then george will move up a position. He will go to position 1. And if we continue doing that, we can add, for example, mary, and she will then go in the third position. If you now want to verify that the Bethany is indeed at position 0, and then George, and then Mary, we can

again use a foreach loop. Let's again type foreach. By the way, a small shortcut here. If you type foreach and you then do Tab, you will get some code generated for you. It is using by default implicit typing, so var, but we can, for example, also simply say that I want to see every element as an employee, but var would have the same effect. And for each employee, I'm going to again execute some codes. And let's keep it simple for now. Let us just for that employee call the DisplayEmployeeDetails method. There we go. If you haven't made any mistakes, we'll see the details of Bethany, then George, then Mary, and so on. So let's now run the application. That will do all this code that we've written here. We will check if 78 is found, then we'll need to enter the IDs, and then we'll create the list of employees. Let's try that out. So 78 is found. That was normal because I did enter 78 in the original employeeIds. Then I collect the list, so now I'm going to add my own custom employeeIds. So let's enter the 5 employeeIds again. And then we are going to show the employee details, indeed, showing first Bethany because we inserted Bethany at petition 0, then George, then Mary, and so on. This is how we can work with lists. In general, I'll prefer using lists over arrays because of their flexibility and their type-safety. And that's what we'll do also in the next modules.

Summary

We now have seen how arrays are an option to work with sets of data in our C# applications. Arrays offers a lightweight and simple approach to do so. They are a reference type, and thus require the use of the new keyword to get instantiated. We then also saw the use of the collection classes and we see that they are more flexible to work with. They will size dynamically. In the next module, we will learn about OO principles, including inheritance and polymorphism. Again, very crucial content in your quest to become a C# developer.

Understanding the Fundamentals of Object-orientation

Module Introduction

Hi, and welcome once again to another super exciting module in the C# Fundamentals course. We have been working with classes and objects for a few modules now. Since C# is an object-oriented language, working with object is probably the most foundational concept in the language. In this module, we will dive deeper in what object-orientation really means, and how C# supports this. We've mentioned this magical object-oriented term a few times already, not only in the introduction of this module, but also earlier, but I haven't explained it in full. So, time to do that. And I'll start this module with exactly that. Once we have understood the concepts, we will apply them in our application. And the first concept is encapsulation, then we'll include inheritance, we'll bring in polymorphism, and finally, I will also introduce interfaces in this module. So again, much interesting stuff. Let's get started straightaway!

Understanding Object-oriented Programming Principles

So let us dig into what object-oriented programming is, and understand how C# supports these concepts. Object-oriented programming, OO for short, is a programming paradigm where the focus lies on objects and classes instead of functions and logic. It is one of the most popular paradigms used by many developers in many languages, and for many, it is the de facto way to create applications. Object-oriented development lends itself well to larger applications. And we have a business model we need to describe in our code, and that will be mapped to classes and objects that are interacting. OO will also lead to code that is easier to update and maintain in the longer run. And OO will also lead to a higher level of reusability. That'll become clear very soon. C# is an object-oriented language and supports all the aspects, or pillars, of OO. We have in the previous modules learned already about classes and objects in C#. We thought of how we could model the problem we are trying to solve into classes and objects, and thus, we have already started applying OO principles. Classes represent models or entities we have, like cars, animals, employees, or more abstract models, such as a transaction. And to create object-oriented applications, C# comes with building blocks we've already seen. We have classes, so custom types, which are the blueprint for objects. Objects are instances of classes with specific data. Using methods, we can define the behavior, and they will work on the data of the object. That data will be wrapped in properties. We haven't talked about these, but we will do that very soon. For now, remember that these contain the

data of the objects, like the value of the name for an employee. It is clear that C# is an object-oriented language, as these mechanisms are in place to allow us to work with classes and objects easily. Writing our code in an object-oriented way means that we will follow the four principles of OO. Note that these are not C#-specific, but they are broadly accepted as the way to write OO code and are thus also applied in other object-oriented languages, such as Java. These four principles, often referred to as the four pillars of OO, are encapsulation, abstraction, inheritance, and polymorphism. I want to keep the theory part as short as possible, but I do need to give you a little bit of introduction on each of these, and after that, we'll go and apply it in our C# application. And so we'll start with encapsulation. Encapsulation means that we are going to write our code so that the information is contained within the object, and only certain information will be exposed by the object. What this means is that there will be part of the code that is internal to the object, and that cannot be accessed by the consumer of the object. The internal implementation is hidden inside of the class, and the internal data is hidden inside of the object. Only what we as the designer of the code choose to be accessible and public is usable from the outside. Think again of the example of a car. When modeling a car in code, we don't expose how the car's engine would work. That is internal to the car. However, things like the blinkers or the headlights of the car, they are public. Consumers that interact with the car can use this information. So using encapsulation, we will effectively as the designer of the class be capable of guarding that no one messes with the state of the object. I wouldn't want anyone to interact with the engine of my car, that is for sure. We've seen already that in C#, we have access modifiers like public and private. Using these, we are able to indicate which members are accessible for consumers of the cost, so the public ones, and which aren't, so the private ones. Private members, as we'll soon see when we start using this, are accessible only from within the class itself. The second pillar is abstraction. If you think of abstracting something away, it means, in real life, that is, taking the hard parts out and making it simpler to use. But the OO pillar means pretty much the same. And it is, in fact, an extension of what we saw with encapsulation. As creator of the car, we're going to create just a simple interface to interact with the object. The object might contain complex functionalities, but we are going to create a simplified interface to interact with that complexity. To talk again about the car analogy, the engine of a car is complex, I guess. When we want to move the car forward, we don't need to know how much gas needs to be injected into the engine. Instead, we have a simple interface, the pedal, and that's what we interact with. That gas pedal is an abstraction that allows users of the

car to interact with it in a simple way. When writing object-oriented code, we want to write methods on classes that expose complex functionalities through a simple interface. That's what abstraction is all about. Inheritance between classes means that we'll create several classes that can reuse functionality from others. That relation will be parent to child, and so we'll write common functionality on a parent class, and inheriting classes, so child classes, can reuse that functionality. Using common logic that's defined on the parent, that is, will dramatically reduce the development time since we only have to write certain code once, and other classes can just reuse it. If we need to make changes to that shared code, again, we'll save ourselves a lot of development time since we only need to make the change in one place. The fourth and final pillar of OO is polymorphism. When creating inheritance, as we just said, child classes can reuse functionality, so behavior defined on the parent. However, it could be that an inheriting type, so a child, wants to give a new meaning to a method, a new behavior. That's what polymorphism is about. Multiple forms can be given to a method; hence, polymorph behavior. The child can still be used as its parent, but even when doing so, the correct, more specific method or behavior will still be picked. Now, I totally understand that these four pillars can be a bit abstract to understand. It is more theory at this point. So let us now go back to coding and understand them in detail. I promise all will become clear during the rest of this module.

Adding Encapsulation

So, let's start with applying encapsulation in our C# application. Here is again the current state of our Employee class. We have a few fields, like firstName and age, which we have defined as public. So public means that from everywhere, this value can be changed. Everywhere means from within this class, but also from outside of this class. And because these fields are public, we can, once we have created an object, change the values of the fields, as we've done before from pretty much everywhere. So definitely from outside of the class, as you can see here. I've created an object of type Employee somewhere in the application, and I'm changing, I'm setting the first name. But while this of course works, this is not a great design choice. Having publicly accessible fields on our class means that everyone can change this data from everywhere. It is basically you sitting in your car going 70 MPH on the highway and then changing the amount of fuel injected into the engine. I am not an expert on cars, but my gut feeling says that this might not be a good idea to do. Your car, represented by the Car class, should at least check its value before it's allowing it to be changed. What we can do is making

the fields private. There, problem solved. Adding private on a member means that it's not accessible from the outside anymore, and so only from within the class can now the value be changed. The data is now hidden to the outside and it can't be changed. Now, while it is safer, I guess, it also makes our class, well, pretty unusable. And we could add methods that then can be public, and that can be used to alter the private data. In there, we could even add logic that validates the data before it's actually being stored in the fields. Adding a method, or even two, in fact, as you just saw, works, but it will become very verbose if we need to add that for all fields, definitely if the class gets bigger and we have a lot of fields to wrap this way. C# comes with a solution in the form of properties, which allows us to enforce encapsulation. Properties are members of a class, just like methods that sit around the private data of a class. They form what is known as accessors. They typically expose the data that we had before, but they can include logic just like a method, to perhaps validate if the passed in data is valid. But for getting access to the actual data, as well as for setting the value, they can include logic. Through properties, a class can expose data, but it can also include logic that is hidden and internally work to validate or change the actual value. They have a special syntax, as you can see here. I still have my private age field, but I also have the age property. Its type is set to int, and it contains a get and a set accessor. In the get, which is followed by a set of curly braces, we will typically include logic to return the actual value. So here, age. In the set, we can include logic to set the value, and we get access to the passed in value through a new keyword, value. Here we can include logic that will validate if the passed in value is valid for what we're expecting it to be. Properties are very common and are, in fact, used much more than public data. They wrap fields and give us the ability to still expose the data we want to expose. While doing so, the consumers of our class won't know what the logic is that sits around our fields. That is hidden, and thus we're applying encapsulation. By default, properties will contain a get and a set, but it is not required to have both all the time, and I'll show you this in the demos. Now we've created the property. How do we now use it? Well, we can use the dot operator again. Let me show you that. First, I'm setting a value here. I'm now using FirstName, so that's the property, and I set it equal to Bethany. Notice that FirstName is capitalizing each word. That is camelCasing, and it's a convention used to name your properties. By simply setting the property to a value like we're doing here, we'll invoke on that object the setter part of the property. If you want to get access to the value from the outside, we'll need to use the getter. Again, we don't explicitly invoke the getter. We call on our object.FirstName, and that will internally be wired to the getter logic. So by

using properties and thus controlling access to the data of our class, we are enforcing encapsulation. And we only provide properties for the fields you want to expose. All the other data stays private within the class.

Demo: Adding Encapsulation

Let us now extend our Employee class with properties, and we'll update the existing logic to use properties instead of the fields. This way we can protect the data in our Employee class. So let us now look at encapsulation in our C# application. I'm here in a Program where I'm using again my employees. I have Bethany, I have George, and I'm letting them do some work, receive wage. We've seen this before, right? You can, by the way, get this snippet again from the snippets that come with the downloads of the course. Now, let us go to Employee. On our class Employee, there's still something that is really bothering me. The data on this class is public. That means that from anywhere I can change it. If we go back to Program, I can change all the values. I can say bethany.hourlyRate = -10. Whoops! That's painful. Now she's going to have to pay when she's working. That's not right. That is because this class, this Employee class, is exposing all its data publicly. From everywhere we can change it. Even after we've set the values in the constructor. We should, in fact, on our Employee class, protect our data. We should be able to on our Employee protect the data of the class. We should maybe build in some validation logic so that a mistake, like setting the hourly rate to a negative value, can't happen, that we can't do if the data is publicly available. We'll need to wrap the data, and that we will do using properties. So let us start by removing this negative hourly rate and let's go back to our Employee class. So as said, I'm going to wrap my data, and I'm going to do that using a property. Let us create a first property for the firstName. So I'm going to wrap this firstName data field. The property is going to be public because that is going to be my access point for data on my employee type. Then the type is typically the same, I'm going to wrap the firstName field, so property will also be of type string. Then I typically also use the same name. So if I'm wrapping firstName, I'm going to create a property FirstName, but then in PascalCase. Then you don't add brackets, but you do need to include the body using curly braces. Now, inside the property, we again use a specific syntax. We can have a getter and a setter in here using the get and set keywords, and each of these will typically contain also a body. In the get, we will write a code that will return the value of the field. So, we will typically return here in this case, FirstName. In a setter, we are going to write the logic to

set the value on FirstName. If no validation or other code needs to run, we can simply specify that FirstName is equal to yet another new keyword, value. In that case, when we assign a value to the FirstName property, that will come in via this value, and that will then be assigned to FirstName. Now, this doesn't do anything yet. We also, as said, need to make the data private. So I'm going to make this firstName into a private field. And when we go back to the Program class, we can't use the first name field anymore because that was set private. We can, however, use the property. When I now specify the FirstName to be John, we'll actually go to the setter of our property. When I want to get the FirstName, I'm going to go through the getter. That will get me back the FirstName of my object. So now I'm always going to interact with my object using these properties when I want to access the data. And they can contain logic. Now, I'll need to do this with my other fields as well. Now, I don't need to create public properties for all data fields. It's very well possible that our Employee class also contains private data that is not even for public consumption, so it's internal to the class, and no one from the outside should see it. Then we don't even need to create a property, but see properties as a way to access the data that is encapsulated inside of my class. Hence, we talk about encapsulation. I'm going to avoid you from watching me do all the typing. I'm going to change my class so it has all private data and public properties. All right, here we are again. My data is now private, as you can see, and I have now includes properties for all these data fields because they should all be accessible from the outside, at least for now. Take a look at this hourly rate. It's a bit special. I mentioned before we shouldn't make that negative, so I have included some validation logic on the hourly rate that will always check that it's greater than 0. If not, I set it to 0. At this point, all my properties are public, but the numberOfHoursWorked. That's the value that I use inside of my Employee in the PerformWork method, for example, to calculate the number of hours worked to keep track of that, let's say. We shouldn't let just anyone change that value. And it is now possible through the numberOfHoursWorked property. Again, not an ideal situation. So I'll still let the get return the value. That's okay. Everyone can see the value. But maybe from the outside, I don't want them to change it. In that case, what I can do is either remove the set, but that will give me some other issues. I won't go into that. I'll make it into a private set for now. A private set means that from within my Employee type, I can change the number of hours worked through this property, but not from the outside. So in other words, the setter is hidden for outside consumers. And for the wage, I'll do the same. I'll also include here a private set. So if we want to see this in action, let's return to the Program class. I can now go to the wage, but I cannot set

it. As you can see, we'll get a red squiggly saying the Employee.Wage cannot be used because its setter is inaccessible. I can, however, get the value. That is still possible because, in that case, I'm using the getter. This should already give you an idea on how properties can be used to give you access, selective access, let's say, to data. Now, since we are using these properties to work with the data, I should, in fact, also refactor my Employee class again so that I'm always using the properties themselves. For example, here in the constructor. Now, I'm directly still using firstName. Now, firstName is not really an issue, but hourly rate, while we had that validation logic in there, maybe we also want to trigger that validation logic when we are setting a value to the constructor. So it is best practice to always inside of the type, also use the properties and always go through them to use the validation logic and other logic in there as well. So I'm going to change my constructor so that it also uses the properties. So this should become FirstName, this should become LastName, and so on and so forth. Now, one other thing that has been bothering me is these names of these parameters. Let me also make them a bit more explicit in what they are doing. It's always best to be as explicit as possible in your code to make it more readable. And of course, I need to make the change here as well, so that we use the new parameter names. There we go. And let's finally also update this other constructor. So there we go. Now my constructors use better names for their parameters. So as said throughout my codes, I've changed the constructor. I also want to update so it uses the properties all over the place. And this was that constant that I haven't changed. It should also be the NumberOfHoursWorked to the property. That's the parameter. I don't need to update that. I need to update FirstName here. I need to update LastName here. This is the parameter again, so I don't need to update that. I'm going to continue doing that and show you the final result here. So as you can see now throughout my entire class, I'm using my properties. Here we have a NumberOfHoursWorked. In here, we have a NumberOfHoursWorked as well. ReceiveWage is using Employee type now, so the property. These are local variables, and we need to have updated wage. CalculateWage uses the Wage property as well. ConvertToJson didn't use anything. And then in the DisplayEmployeeDetails, also I'm using now the properties. So now I think we're good with using the properties all over my Employee type. Let's make sure that everything still builds. There we go. Success. So our Employee class is now in a much better shape. It is encapsulating its data.

Bringing in Inheritance

Let's now see how we can add inheritance next in our application. Let's return to the business case where we have worked with employees. Let's for a minute think again of real life, not in C# code yet. An employee. It's a pretty broad term, right? When we look at Bethany's Pie Shop, all people working for the company, they are all employees and have a number of properties, like the name, but employees will have different roles within the company. Although they are all still employees, some will be store manager, and they will probably have a few specific functions. A few will be manager. There will also be sales people. And to figure out new pie flavors, I have a few researchers working for us as well. Because of their different role, they'll have different tasks, and perhaps as a company, we also will be storing different pieces of data for them. But they're all employees and they all have a name. And yes, they probably also share quite a bit of other functionality. They'll work for us, they'll need to be paid, and so on. Do you hear where I'm going with this? What I'm introducing here is inheritance. All the different roles, the manager, the salesperson, the researcher, and so on at Bethany's Pie Shop, they're all employees in the end, and we can treat them as such. Because of this, they will share a lot of their functions as well. When we make the move again to C#, inheritance is a very important concept when applying object orientation. What we will do is creating a class typically, and that will contain the base functionality. That will be in our case the Employee class. Different other classes will inherit from this base class. And because of this, they will share the functionality with the Employee class. Indeed, managers are also employees, and many functions are the same, but probably not all. A manager needs to make decisions, needs to attend management meetings, and the like. That is specific, and that will be part only of the manager's functionality. What we're getting this way is a tree, an inheritance tree, and we will also model that in our application. Inheritance works with parent and derived classes. The parent or base class is the one that contains the shared functionality, whereas the derived or child classes will add specific functionality. You'll hear the term base class and parent class being used intermittently, as they're really the same concept. In our class, as said, the employee is the parent, and the manager and the researcher and the salesperson will be modeled as derived or child classes. By default, when they inherit from the base class, they will share functionality defined on the parent. One of the advantages that comes out of inheritance is that we will be able to reuse code. Instead of giving all inherited classes the same functionality, we now just add it on the base class and reuse it from there. That also means that if we need to make a change to common functionality, it only has to be done in one place. Our code, therefore, becomes easier to maintain. Another big plus. Now,

I've so far mentioned the relation between employee and manager, so parent and derived cost, but it can actually be multiple levels deep. We could, for example, derive from salesperson yet another class, like junior salesperson, which is again, more specific.

Creating a Base and Derived Class

Now that we understand the concept of inheritance, we're really going to apply this to our code, and I'm going to follow the example we just laid out. I'm going to create a base class and a derived type in C#. And this way, we will understand how inheritance works in C#. Being an object-oriented language, C# fully supports inheritance, since it's, as said, one of the cornerstones of object orientation. Here you can see how we can let C# know that a class needs to inherit from another class. The top we have the BaseClass. As you can see, at least on the class declaration, there's nothing special going on there. Then the second one, the DerivedClass. It's a bit different. It needs to point to its direct parent. And this we can do by adding behind the class name a colon, and then the name of the BaseClass. So the colon and then the BaseClass name are the way to create inheritance in C#. Doing this has quite a few consequences, as you'll come to understand. When we apply this on our business case, Employee will be the base class, and so we don't need to make any changes at this point. Then we'll bring in another class, Manager, and that will inherit from Employee. As you can see, it has a colon and then Employee as its base type. Manager is a Derived Cass, Employee is the BaseClass. Now, as said, introducing a BaseClass has a number of consequences, which is pretty much why we would add inheritance in the first place. The Employee class will contain functionality, and all inherited types will gain access to because of inheriting from it. This means that the name field and the PerformWork method here defined on Employee will become available for Manager, as if they were part of Manager directly. That's what you see here. Manager itself contains a method called DisplayManagerData, which will now will just display the name of the Manager. But notice that Manager doesn't contain the name field. No, that is defined on the Employee class. But you can use it on Manager because we inherit from Employee. Members defined on the BaseClass become accessible for the DerivedClass. Now, that last sentence does require some extra explanation. The DerivedClass can access members of the BaseClass. This is where access modifiers start playing an important role. Remember, the three most important access modifiers, public, private, and we've already looked at these briefly, and there's a third one, protected. When using inheritance, the

DerivedClass can access the public members of the parent. However, members declared as private on the BaseClass are not accessible; they are internal to the class and can't even be accessed by inheriting types. So if you want a DerivedClass, never to access a certain field, property, or method, whatever, just make it private in the BaeClass. And then protected is somewhat in the middle. When using inheritance, protected members of the BaseClass are accessible for inheriting types. So, for a derived type, protected and public mean the same. A derived type can access protected members. However, protected members cannot be accessed from anywhere else. So for non-inheriting types, protected behaves as private. Let's see them in action before we head to Visual Studio. Notice what I have here. I've made the name private on employee now. Private means mine and mine only. No one can see or use this member. Our trusted Manager class, although it inherits from Employee, I won't be able to access the name field any longer. You'll get a compile error saying that indeed name is private and it cannot be accessed by the derived type. However, making it protected again does allow the Manager class, a DerivedClass, that is, to access it. As said, protected, behaves like public for inheriting types. For anything else, it behaves like private.

Demo: Creating a Base and Derived Class

All right, we've been talking a while about inheritance now. Time to see it in action. Let's head back to Visual Studio and see how we can turn our employee into a base class. Then we'll create an inheritor type, and finally, we'll learn about the access modifiers and what influence they have. Now so far, we have only one class, the Employee, really, which is going to be used to work with our employees. Now, we use the EmployeeType enumeration to basically distinguish between the type of employees. But instead of using the EmployeeType to distinguish between types of employees, we're going to use different classes, and then we can give different behaviors to these different types. So I'm going to start by removing the EmployeeType first in my Employee class, and then we'll bring in multiple other classes that then contain other behavior and maybe also other data. Let's do that. So I'm going to remove the EmployeeType. There we go. That will give us some errors. So let's see where we get some errors. That's over here, of course, the property. We don't need that anymore. I think we'll also have an error over here in the constructor, there we go. And since we don't need the EmployeeType anymore, we're also going to remove it from the constructor. So give me an error over here as well, so let's remove that as well. And we also have some errors in the ReceiveWage methods because we

didn't explicitly check for the EmployeeTypes. I'm going to remove that here and to make this code, well, a bit more generic again, so that we have to wageBeforeTax with the NumberOfHoursWorked times the HourlyRate.Value, taxAmount, and then the CalculatedWage. Finally, I also need to make a fix here in the DisplayEmployeeDetails. That seems to be okay. Now, we also need to go back to Program, and also over here remove the EmployeeType argument. Let's do a build to make sure that everything now works. All right, we're back to a situation without the EmployeeType. Now, as said, I'm going to be more specific. I'm going to introduce a new class for the manager. For example, first, that is then going to inherit from employee. So let's go there. Let's go to the HR folder again, and let's bring in a new class. We can go this time to a New Item, select the Class template, and the type that I'm going to create, the class that I'm going to create is Manager. Again, a class is being created. The first thing that I'll want is since every manager is an employee, I'm going to let the manager inherit from employee. And I do that by typing first a colon and then the class I'm going to inherit from. And that is, in this case, Employee. By doing this, the functionality defined on Employee is now also available on the Manager class. So this is inheritance already applied. My managers are now employees. Now we are getting a red squiggly here. Why is that? Well, it says here that we need to generate a constructor, and we're getting, in fact, two options. I think this is the one I want that's first generated. I've now created a constructor on manager because when I'm creating a manager, I'm also going to create an employee. So what I'm doing here is I'm creating a constructor like we've done before, accepting a few parameters, firstName, lastName, and so on. But then I'm also going to call the constructor on the Employee class. I'm using for that the base keyword. So what I'm now creating, when I'm constructing a manager, I'm also going to create a behind sitting employee, let's say, and I'm going to pass these parameters here. This base keyword is very similar to what we had in employee with the this keyword. In here, we called from this constructor, this constructor. Now I'm calling from this constructor on Manager the Employee constructor over here so that when a manager is constructed, also an employee will be constructed. And so now in here, I have access to the things which are already in Employee. For example, I can access the FirstName. There you go. That is the property FirstName on Employee. I can change that here. I also have access to the PerformWork. That is the method defined on Employee. And you see that IntelliSense is already suggesting that here. And let's come back to that later because we also need to talk a bit about what exactly is accessible from the manager. But let us first go to the Program and construct now a manager. Let's remove George here, and let us now

bring in a Manager. And let's call that Manager mary. It's going to be a local variable mary of type Manager. So I'm going to create my new manager here, and I'm going to need to specify a couple of arguments as defined in the constructive parameter set. So firstName, lastName, email, birth date, and hourly rate. There we go. We've now instantiated Mary the manager. And now let us work with Mary. Let us first on Mary call the DisplayEmployeeDetails. That DisplayEmployeeDetails if I do an F12 or a right-click on that to go to the definition, which also says here F12, we go to the DisplayEmployeeDetails on Employee. So that method is defined on the base type Employee, but I can use it on the Manager, the derived type. The same goes for letting Mary perform some work and also let her receive a wage. So let us make sure that everything works. Let us run this and also see the output of that. So everything compiles fine. We see that Bethany is being displayed and that Mary is also being displayed. Mary performs some work and also gets a wage for that. So we now have a manager and an employee. We have different types that we can use in our application without having to use the enumeration. You may be thinking, what exactly is the purpose? Because my manager, although I can distinguish between managers and employees, can do the exact same thing. Well, at this point it is true, but I can add extra functionalities on the manager which a manager only can do and an employee cannot. Now what does the manager typically do? Well, they go to meetings, right? So let us create another method, AttendManagementMeeting. That is now extra functionality that only a manager can do. This I couldn't solve with enumerations because enumerations were just a numeric value that they could keep track of on the EmployeeType. This is now functionality and is only accessible for the manager and no one else. Now, management meetings tend to take long. So I'm going to say that the NumberOfHoursWorked is going to be + 10. I'm getting a red squiggly here. What is wrong here? Well, it says here when we hover over this that the set accessor is inaccessible. Hmm. That's weird. Let's go and take a look here. Ha, there we go! Here we see on the public property that we have created a private set in a previous demo. Indeed, I didn't want NumberOfHoursWorked to be used from the outside, or let's say changed from the outside, but instead, I want it to be only accessible from within Employee. But now I've created the class Manager that inherits from Employee. So we'll need to make this accessible from within Employee. So what I'm going to use here is a different access modifier, and that's protected. A protected set will allow access to NumberOfHoursWorked from a class that inherits from Employee. Ta-da! And gone is the red squiggly. Things are now working fine. NumberOfHoursWorked is now usable from Employee and its inheritor.

So, Manager inherits from Employee, so we can change it from here. And we'll also add some output so that we see that the AttendManagementMeeting is being used. Now, I can go back to my Program and ask Mary to attend a management meeting. Can I also ask Bethany to attend a management meeting? I don't think so. It doesn't exist. Indeed, this is only accessible on the inherited type, so Manager. Now, I'm going to bring in some more types corresponding to some of the enumeration types that we had before. First, let me bring in the StoreManager. The StoreManager is a very simple class that is very similar at this point to what we had initially with the Manager. I'll also bring in another type that also inherits from Employee, and that is the Developer. Someone has to create those applications for Bethany's Pie Shop, so we will create a type, Developer. Again, of course, it inherits from Employee. And so we'll extend this type also with an extra property currentProject, which I just put it as a string, so I use a property to wrap the private field and I also create a constructor. So here you see also that we can on the type Developer add extra data, so currentProject. I'll also bring in the Researcher, and I want to use the Researcher to show you that it doesn't have to be just one level deep. We'll create another class, the JuniorResearcher, which then inherits in turn from the Researcher. I'm also going to bring in the JuniorResearcher, which as said, doesn't directly inherit from Employee, but instead inherits from Researcher. That's also possible. And of course, this will also require the use of a constructor. So now we have a full class hierarchy. We have Employee, we have Manager that inherits from that, Researcher that inherits from Employee, and we have JuniorResearcher that inherits from Researcher. While we're here, I'll also bring in some extra functionality on Researcher. I've pasted in a bit of code. As you can see here, I have the number of pie tastes invented. That is extra data that I use on the researcher, and I set it initially to 0. There is a method that's also added, so extra functionality only available for researchers and also for junior researchers, ResearchNewPieTastes. And that, as you can see, will increase the NumberOfHoursWorked with the number of researchHours that is passed in. And then we'll leave it to chance if they find something. Again, we use a random, between 0 and 100, and if that value is higher than 50, then we increase the number of pie tastes invented, and that was this value here. And let's go back to our Program, and let us now create a JuniorResearcher. I'm going to bring in the JuniorResearcher, bobJunior, and I'm going to let him do some research on pies. Hey, Bob, can you do some research on pies? That is functionality inherited from the researcher, and I'm going to let him work for 5 hours. I'm going to let him do that twice. Maybe we have a chance to find a new taste. Let's

try and see what the result is. And we're not in luck. Bob Spencer is still working on a new pie taste. So this should make clear the use of inheritance. We can add extra functionality on the inheriting types while still using functionality defined on the base type.

The Is-A Relation

What inheritance in C# is bringing in is what sometimes is referred to as the Is-A relation. Indeed, even in the spoken word, you would say a manager is an employee. The Is-A relation is exactly what inheritance is bringing us, also in the behavior we're getting in our C# code. Let me define a manager instance where the Manager class derives from Employee, all public and protected members defined in the base Employee class become available for manager. If PerformWork is public or protected on the Employee class, because manager is an employee, it becomes available for the manager. The Manager class can, like you have seen in the demo, of course, also add new functionality, which is then only available for manager instances, not for employee instances. And of course, multiple types can also inherit from the base Employee class. If the Researcher class also inherits from the base Employee class, every researcher, too, is an employee, and therefore researcher instances will get access to the PerformWork method as well.

Demo: The Is-A Relation

Let us return to Visual Studio and work with multiple classes that will inherit from the Employee base class. Because they inherit, we can use the Is-A relation, and we will get access to the base functionality. Now the Is-A relation is very important. It means that derived types can be addressed through their base type. They are just more specific. In fact, every manager, I think I've mentioned it already a few times, is an employee. Hence, we can, on this line here, replace Manager, the reference, that is, by Employee. Now, Mary, which is now an Employee reference, still points to a Manager. Now for Bethany, nothing has changed, as you can see here. All the methods still work as before. But for Mary, we now have an issue. As you can see, we're getting a red squiggly here under AttendManagementMeeting. That function is not available on Mary. Mary is now known as an employee reference, although it does point to a manager behind the scenes. But the reference is of type Employee and that doesn't know about AttendManagementMeeting. However, the

DisplayEmployeeDetails, PerformWork, and so on, they still work. They were defined on the base type. Also, with JuniorResearcher, Bob, we can for example, replace this with the reference of the base type, Researcher. Now, bobJunior is still a JuniorResearcher, but every JuniorResearcher is a Researcher, and the ResearchNewPieTastes was defined on the Researcher type. Hence, that method can still be called without any issue on bobJunior, which is now used to a Researcher reference.

Demo: Understanding Composition

Next to the Is-A relation, there's another possibility the Has-A relation. Take, for example, a trusted Employee class again. You've already had properties like firstName and lastName on there, but say that you want the Employee also to contain an address, and that address will represent through another type, another class conveniently named Address. In that case, Employee contains Has-A address. Address is part of the employee, and in this case, we talk about composition. The address itself is accessible through the employee. Let's return again to Visual Studio and extend our application with a new class, Address. And of course, we'll add to our Employee class a property of type Address. Now we've seen the Is-A relation in the previous demo. Now let me show you composition which is the Has-A relation. I'm going to create a new type, the address type, and will then let an employee also contain an address, and that will be composition. But let's first create the address. The address will be a new type that I'm creating, a new class that I'm going to create, and maybe we can share that between employees, customers, and so on. It contains general address information. Let's create a new class for that. Add, New Class, and that will be my Address class, and it will be a very simple class. In fact, it will just contain some data and a constructor. Let me paste in the code. You can find it in the snippets. So here is the address. It contains a street, a house number, zip code, and city. As private strings, I have a constructor, which just sets these values, and I then wrap each of these private fields with a public property to get access to this data. This is just going to contain some data. I don't even have any functionality on that. Now, I'm going to go back to my employee, and I'm going to let my employee contain an address. That will be the Has-A relation, as mentioned before. So I'm going to go here, and I'm going to let my employee also have an Address field. As you now see, I'm now using another custom class within this Employee custom class. And I'm going to call that private field address, lowercase. And as before, we can also create a property to

access the address information. So let's go here and bring in the address property. That will just return the private address field. And we can also set it by passing in a new address. Now, my employee contains the address. I have the property to exit, but, in fact, I can also introduce maybe a new constructor that also comes with address information. Let me add a new constructor here. Here we can see that I have now added another overloaded constructor, which now gets in a street, the house number, zip, and city. But it's not using it yet. Maybe what we can do here is instantiate the address. That would then be the property, and I can assign that to be a new address. Indeed, my Employee class will create also its address, and it's going to use for that the constructor of Address. And that accepted the street, the house number, the zip, and the city. There we go. Now the wrapped Address instance now is also constructed from the Employee constructor. We can now use this constructor also from the program again. Let's go to Program. Here I am now creating our new employee, Jake. The Jake object is being created using that extra overload of constructor, accepting the street, the house number, the zip, and the city. We can now also ask for the street name of Jake. How do you think we would do that? Well, we could go to the jake object. That has a nested address, that is the property address, so this one, and that in turn contains the street. So this way I can access the street in the nested object address inside of the object jake.

Using Polymorphism

Now that you have a good understanding of inheritance in C#, let's look at polymorphism, another important pillar in object-oriented programming. We've now created a few classes and applied inheritance. I think it's clear that using inheritance, we can reuse code. Functionality declared on the base type, like you see here with the PerformWork method is accessible on the inheriting types, and that allows us to move common functionality to the base class. Thanks to inheritance, we can now create a Manager class and called the PerformWork method defined on the base EmployeeType on it. And similarly, we can also create a Researcher and call PerformWork on objects of that class. Although that is great, the invoked method will always be the same for all inheriting types. You may be thinking, was that not the goal of inheritance? Why yes, of course. But what if we want to have a modified version of PerformWork for perhaps a few of the inheriting types? This is where polymorphism comes into play. Polymorphism will allow us to provide a different implementation by an inheriting class for a given method. The name polymorphism indeed says poly, which is multiple, and

morph, which is form. You can give a different form to a method in the inheritance tree. Polymorphism is tightly linked with inheritance. Instead of just using the same base class method, we can, in an inheriting class, define a new implementation. The base class should actually indicate that an inheriting type can give its own implementation. To use it, there are two new keywords we'll start using in this context, virtual and override. And let's see them in action. Here we have our sample again. Since the manager might need to do specific tasks, the implementation of PerformWork might be different than the generalized one we had before on Employee. What we should do now is recognize this in the Employee declaration. We should say, well, here's an implementation that all inheriting types can use, but you can provide a different one in the inheriting types. This we do by marking the PerformWork method on Employee virtual. This is a signal for C# saying that types can provide their own implementation, and that should be used if available. On the right snippet, you see the Manager again, which now needs a different form of PerformWork. And to create this, we'll use the override keyword again, stating explicitly that this version for the Manager type defines a new version of the method. Whenever we now use a Manager anywhere in our application, this version will be used. Here you can see this again in a diagram. We have Employee, which defines a virtual PerformWork. We have two inheriting types, Manager and Researcher. Manager defines an override, so it has its own version of PerformWork, while Researcher does not. Let us now see through polymorphic behavior which version of the method is called when we have a list of instances. Say that we have an Employee instance. In that case, the virtual PerformWork on the Employee class will be called. Next, we have a Manager. C# will see that the Manager class had its own version, so it will use that. When we are using our Manager instance, we don't have to indicate this; C# will always take the most specific version. If we call PerformWork on a Researcher, can you think which one will be used? Indeed, the one on Employee, since that is simply inherited. What polymorphism will then also bring us is the ability to reference an instance of the derived type through a base type reference. Now let me explain. Take a look at the first line of code you see here on the slide. I'm creating a new Manager object, since that's the part on the right, responsible to create a new object. But then on the left, I create a variable of type Employee. I'm using a base type reference here to point to an object of a more derived type. And that is possible thanks to polymorphism as well. On the second line, I'm doing the same for another type, Researcher, which we know also inherits from Employee. We can now on the Employee reference invoke members defined on the Employee class, such as PerformWork. But

again through polymorphism, the most specific version will be called. So if the Manager class has an override of PerformWork defined, that one will be called even though we access it through a base type reference. Now if AttendManagementMeeting is defined on the Manager class, we can't access it through a base type reference, since that is simply unknown to the Employee type. And you may be asking, why is this useful? Why would I treat my more derived instances using a base type? Great question! If you have a group of managers, employees, researchers, and so on, we can, in fact, treat them all using a base type. So you put this set in an array, and we want to invoke the PerformWork method on all of these. We can simply loop over the array and call the PerformWork method on each of them. C# will always take the most specific implementation based on the type of the actual object. If the actual object or Employee reference points to is an Employee, the virtual base implementation is used. If you're pointing to an actual Manager, again, the Manager-specific implementation is used. This is very similar to what we had before, but now we use all objects using a base type reference.

Demo: Using Polymorphism

Let us return to our sample and bring in virtual and override. We'll then use a number of instances, and we'll see which version is being used. Great news at Bethany's Pie Shop! All employees are getting a bonus. How great is that? And since all employees are getting it, we can define a method, GiveBonus, on the Employee base class, right? While we can do that, it might be so that different types of employees, so managers, maybe researchers, I don't know, actually will get a different type of bonus, a different implementation for the GiveBonus methods. That is something that we haven't done. So far, we have always used the PerformWork and the DisplayEmployeeDetails on the base type without giving the inheriting types a new version of that method. That's what we are going to change now. What we're going to do now is we are going to define the GiveBonus on the base types, so on an employee, and then we're going to let inheriting types give a new implementation if necessary. Let's do that. Let's go back to Employee and let's bring in a GiveBonus method here. So here I now have created the GiveNonus method on Employee, so I can now go to my Program again, and on Mary, although she's a manager, I can give her a bonus. Right? Of course, because that is now defined on the base class. And that'll work. But as said, it might be so that managers get a different type of bonus. And so do researchers, perhaps. To do that, to let inheriting types create a new version of GiveBonus, give their own implementation to it, let's say, we need to go back to the GiveBonus on

the base type and actually make it virtual. Virtual is another new keyword in C# that allows me to have inheriting types override this functionality and implement a new version of this functionality. If no other functionality is given, well, then this GiveBonus will be used. Basically, virtual doesn't change the default behavior as long as we don't create a new implementation on an inheriting type. Let us now go to Manager. The manager doesn't have a version of GiveBonus at this point, but I'm going to add one. If I start typing override and then do a space, then you will get in Visual Studio a list of all functions that you can override. And GiveBonus is in there. I actually can now create a specific version of GiveBonus that is only applicable for instances of the manager type. So when we are calling GiveBonus in this case on a manager, this specific version will be used. So, based on the type we call GiveBonus on, a different implementation will be used. If we call it on manager, this version will be used. If we call it on an employee or another inheriting type that doesn't define a specific override, this version will be used. At runtime, C# will actually look at the object we're calling this on, and it will take the correct version. This is polymorphism, multiple versions, multiple forms of the same method can be created on inheriting types, and based on the type, the correct version will be picked by C#. So we're going to test this out. I'll put a breakpoint already here and give bonus on employees, so the virtual one, and I'll also do the same here on the manager one. Let's now go back to Program. I'm going to change my list of employees here a little bit so that Bethany is a store manager, but she's actually being referenced through an employee, a reference. Mary is a manager, Bob Junior is still a junior researcher, and then we have Kevin and Kat, which are both store managers. And instead of having this code here, we comment this out for now, and let us add all these employees, because everyone is an employee, to a list of employees. We've done that before. I'm going to bring in all my employees into this list. There we go. We now have a list of all our employees. I now want to loop over all my employees. I'm going to use another foreach for that. I'll now call the DisplayEmployeeDetails and GiveBonus on each of these employees. We have the breakpoint set, so let's run the application and see which version of GiveBonus will be picked. Here we have GiveBonus. The first name is Bethany. That was actually, if I'm not mistaken, a StoreManager now. Indeed, but StoreManager doesn't have a specific implementation for GiveBonus, hence, the one on employee, the virtual one, is used. I do an F5, and now notice I'm in the implementation of GiveBonus in the Manager class. Indeed, we are calling this from Mary. Now the give bonus, the specific one for manager is being picked by CRP. The different for is used. Now we go to Bob. Bob was a junior researcher, and he also

gets the default implementation for GiveBonus. So here you now can see that depending on the type, the correct version of GiveBonus is used. We have created a virtual default implementation on the base, and we've created on manager and only on manager, in this case, a specific over

Introducing Interfaces

In the last part of this module, we are going to learn about interfaces and see how they can be used to apply polymorphism in C#, too. A few modules back, we already saw this slide. It contains the different categories of types that can be created in .NET. I've already looked at enumerations, structs, and, of course, classes. In the final part of this module on object-orientation, I want to touch on the interface type. Okay, first, what are interfaces, and why are they so important in C#? Well, great question! Let me explain. Interfaces are contracts. Uh, okay, what do you mean? Well, think about a contract in real life. What does it do? When you sign a contract with someone, it will bind you and probably another party to follow a certain set of rules. I think we're all familiar with this concept. An interface is pretty much the same, but then in code. It will contain typically a set of declarations of related functionality. This could be just a list of methods, for example. A class can indicate it will implement that interface. In that case, all members defined in the contract for which no default implementation exists must be implemented as per contract. And we now want to use that class that implements the interface. We know for a fact that the methods are implemented. That's the contract. We can rely on the fact that they have received an implementation, and we can build our code around that. Here is a first interface. It looks similar to a class, but there are definitely some differences. To start with, it has the interface keyword, and then the name, which will typically start with an i. And the letter is a convention; it's not required. But you will see that most interfaces you will encounter follow this rule. Then inside the interface, we see methods, but look at them more closely. They have no implementation. It is just the method signature and then a semicolon. So this is the contract. Classes that now implement this interface follow this contract, and that means they will need to give an implementation for these methods. Now what do I mean exactly when I say implementing an interface? Well, let's see, implementing an interface is similar to inheriting from a base class. Again using a colon, you can let the class implement an interface. If you see the Manager class, and if you want to make sure that the class implements the contract, that is, IEmployee, our interface, implementing an interface or the contract, also ties you to what the contract states. Your class is signing the contract, and that means

that an implementation must be provided for every method without implementation in the interface. Otherwise, the C# compiler will complain. The method implementation is really just a regular method. Now, interfaces cannot be instantiated. You can't call a new on them. If you think about it, well, that's pretty normal. They're just contracts. There is no implementation in the methods. Although we cannot instantiate an interface type, the code you see here on the slide is perfectly valid. I'm using a variable, which has the IEmployee type, so the interface. We can use this to point to anything that implements the IEmployee interface. And to top it off, we can on that instance call methods, which are defined in the contract. That would be here the PerformWork method, for example. So indeed, through interfaces, we can also work in a polymorphic way. We can, using a variable of the interface type, point to any object that implements the interface.

Demo: Using Interfaces

In the final demo of this module, we will create a simple custom interface and implement it. I'll then also show you the interface in combination with polymorphism. In this final demo of this module, I want to give you a small introduction to using interfaces. An interface is really a contract. It will define a number of members, such as methods, and when a class signs up to implement the interface, we know for a fact that all the methods defined in the interface will have an implementation in that class, and that we can again use for polymorphic behavior, for example. I'm going to start by creating a very simple interface called IEmployee. Interfaces typically have a name that starts with an I. That is a convention; it's not really required. You can also leave it out, but it is a way to easily distinguish between classes and interfaces just looking at the name of the type. Creating an interface is done using the interface keyword. I'll just replace the class keyword here with interface. In here, I can now define, for example, a number of methods that I want to be sure that all my types that implement the interface will give an implementation for. So it's my first interface. As you can see, it contains a number of methods. Basically what I have here are the method signatures that I already have in employee. But they are a bit different. They don't contain an access modifier. As you can see here, they do contain a name, a return type, but they also do not contain an implementation. I just have a semicolon at the end. Like I said, this is the contract. I want to make sure that classes that implement this interface, that they will give an implementation. So I can now go to my Employee class, and I can implement the IEmployee interface. All the members, in this case, methods defined in this interface, have an

implementation inside of my Employee class. We, in fact, already had that. And what if I bring in on my IEmployee a new method? Say that I bring in void and GiveCompliment. All employees would love a compliment, right? So that's now in the contract defined that all the classes that implement this interface also need to be able to give a compliment. If we now go back to our Employee, we get an error, a red squiggly. Employee does not implement the GiveCompliment method. We need to do that because otherwise the contract is broken. If I click here on the light bulb, I can say Implement interface. This will, typically at the end, add a default implementation. There's no code in here that we can really use, for the GiveCompliment method. So we can now replace this with real code. Now our Employee class fully implements the interface. It gives an implementation to all methods defined in this case in our IEmployee interface. Now you may be still a bit unclear on why we created that interface. Why is that contract so important? By creating a contract and having that contract be implemented on classes, we know for a fact that all these classes do give an implementation for the methods defined in the interface. Therefore, I can just call those, and again to polymorphism, the correct version will be used. Let's return to the Program class, and let's try creating an IEmployee first. We'll get a red squiggly. Visual Studio is saying I cannot instantiate an interface. Indeed, if you think about it, IEmployee doesn't even have any implementations, so how would we use it directly? Well, we can't. We can actually use IEmployee, however, to point to classes that implement the interface IEmployee. So let's get rid of this, and what I could do is replace here Employee with IEmployee. Now I've created a reference of type IEmployee that points to StoreManager, which is an Employee, and that implements the IEmployee. I can do that also for the Manager, for the JuniorResearcher, StoreManager, and this StoreManager here as well. I'm now basically using an interface reference to point to classes that implement the interface directly or via the parent. And I know for a fact that all these objects implement the methods defined in the interface. So I can also replace here Employee with IEmployee. There we go. Now we have created a list of IEmployee references, and they all implement the DisplayEmployeeDetails, the GiveBonus, and also that newly created GiveCompliment. There we go. And that is the power of interfaces. There is a contract defined, and by implementing the contract, we know that the class gives an implementation for all these methods. Let's run things and make sure that everything still works now using the interface. As you can see, we've now addressed our employees through an IEmployee reference, and we've called the GiveBonus method on it, and the GiveCompliment method on it.

Summary

Whew! That was a long module, but it was a very important one, too. Object-orientation is a crucial topic on your way to become a C# developer. I have touched on the basics of object-orientation of C#, and we have seen how the different pillars of OO are available to use in C#. We have seen how properties can help with encapsulation in order to protect the data of our objects. We've seen inheritance and how it allows us to reuse code that we've written in a base class. Through polymorphism, we get the ability using virtual and override to create a specific implementation inside inheriting times. And we've just touched on interfaces, too. They define a contract, but can also be used in a polymorphic way. Now that we have a working set of classes, we can now add tests to validate the correct working of our code. We can use unit tests for that, and those are part of the next module.

Testing C# Code

Module Introduction

We have already written quite some code so far. The more code you write, the bigger the chance that we have errors in the code that we create. Testing our code is therefore important, no matter how large or small the application we are creating. And for that, several options exist. The different options to test our code will be the topic of this very module. We'll need to test our code while at different times when creating the application. While we're writing our code, we'll use the debugger to test the code that we've written and using the debugger efficiently will be the first topic we'll look at in this module. Secondly, I'll introduce unit testing, a way to have code test other code, small pieces of code, that is. With unit tests, we'll be able to verify that the logic works as intended and keeps doing so even after we've made changes to it.

Testing Your Application Using the Debugger

So let's start learning more about the debugger and how we can use it to test our running application. We've already seen and used the debugger a few times in the previous modules. Debugging allows

us, as the name implies, to find errors in our code while it is running in debug mode. So, the application is effectively executed. That means that the errors that you are looking for are runtime errors, which are harder to find compared to syntax errors, or compile time errors. The latter are found by the compiler even before we can run the application. Debug mode means that the application is running, let's say, in a monitored way, so that errors that happen in the running application are directed back to Visual Studio. And in Visual Studio, we'll see the lines that cause the errors that we'll get. And at that point, our application will go in break mode. We can also put what is known as breakpoints in the code. A breakpoint will break the execution of the application and allow us to inspect the state of the application. We can inspect the value of the variables in-memory and also step through the code. We can let the execution run line-by-line and see errors that happen in the running app. Writing a real application without the debugger would be really hard. I don't know how we'd solve all the bugs I write daily without it. Here you can see Visual Studio with a breakpoint that I've placed in the code. By simply clicking in the sidebar, a breakpoint is created. Once you then run the application and that line gets executed, the application will enter break mode. While we're in break mode, we can also, as said, step through the code line-by-line. And doing so gives us the ability to pinpoint the statements that cause the error or to inspect variable values, as said. Because this is such a common task to do, Visual Studio comes with a number of useful debugger commands. And for most of these, a keyboard shortcut is enabled by default. You can see these here. I'm sure that after a short while you will know these by heart. F5 is very common to just start debugging your application. In other words, running the application with the debugger attached. F11 and F10 are used to, when you've hit a breakpoint, go step-by-step. In fact, I should say line-by-line. The main difference is that Step into, so F11, is a command that will also step or go into methods that you're calling on a particular line. I will show this in the demo. By default, connected to Shift + F11 is used to Step out, let's say, go up, if you will, the executing method.

Demo: Using the Debugger

It's best to see the debugger tooling in action, so let's do that next. I'm going to show you in some more detail the debugger and how you can create breakpoints. We'll also look at the most important debugger windows that Visual Studio has. So let's now take a look at the debugging features we have in Visual Studio 2022, and most of the things I will show you, you can also do in Visual Studio Code,

but we are, of course, looking here and using Visual Studio 2022, so I'll use those debugging features here. We have here in the Program class the IEmployee bethany that is a StoreManager, and now we are going to ask the user to enter the number of hours worked. We've already done that a couple of times, but notice this time, I've now written it in a few lines of code. First, I've created a variable numberOfHours is 0, and I've captured the input in string input, and then I pass that into numberOfHours. Once that is complete, we'll ask on Bethany to display the employee details, and we'll also ask Bethany to PerformWork. That will do that 1 hour. And then we'll also call PerformWork, passing in that number of hours that we have here. And let us now run the application and see what happens. So how many hours do we need to register for Bethany? Well, let's say 10 hours. Whoops. We're getting an error. Visual Studio is going into break mode, as you can see here. On the line where an exception occurred, where something went wrong, we are now stopped. We are in break mode. This gives me the ability to see what went wrong, and I can also look around in the application. I can now also look at the values. For example, here, I can see that input was indeed 10 hours, and then this line caused the exception because then I actually tried to parse 10 hours. Ten hours is not an integer value, hence, the parsing failed. So thanks to this break mode, I was able to understand what went wrong in my application. Now, if we let the application continue, it will simply close because, of course, this is a nonrecoverable error. The rest of the application cannot continue. You will need also see that something went wrong. The error that occurred is shown here on the console. I will talk later about exceptions and exception handling, but this is what you get by default in Visual Studio. Now we ended up here in break mode because an error occurred, but we can also let Visual Studio go into break mode without an error occurring. If I put a break point, let's say on this line here, then Visual Studio, as soon as it hits that line, will also go in break mode and allow me to do pretty much the same things. Let's try that again and run the application now with the debugger attached once more. So let us now enter, well, a valid value. Let's enter 10. Again, we can see that the input is now 10, and now this line will also succeed. Now using the debug tool bar, I can step through the code, and that's what I want to show you next. Now, we've already done this a few times, but I want to make sure that you understand these options in full because it is such a common thing to do when writing C# code. Now, I can actually let this line go through, I can, in fact, do a Step Over, and now we will see that number of hours is now indeed 10. Now we are at the DisplayEmployeeDetails. Well, I am actually not really interested in DisplayEmployeeDetails, so I can again do F10 to Step Over. Now, I actually want to step

into PerformWork. I want to see how that executes and maybe fix errors while we're there. For that, I can do Step Into, which is typically linked to F11. And now, as you can see, the flow of execution is followed, and that will call into the PerformWork method. I can then again do a Step Into, and that will jump in the other PerformWork. That is the one that is invoked here. And now I can see that this is now invoked with that constant value that we used before. So this number of hours here is going to be 1. That's very useful. Now, while we can hover over these values, it also makes sense to see the variables that are currently in scope. And for that, we have a number of debug windows. There is the Autos window, which is down at the bottom of my screen. If you don't see it, it might be hidden. And then you need to go to Debug, Windows, and here you see all these type of windows You have Autos, Locals, and Watch windows. Those will help you a lot while debugging. Let us open the Autos window for now. The Autos window, as the name implies, automatically gives you an overview of the variables currently in scope around the current breakpoint. So we are here in PerformWork, and you already see that number of hours, so this one is showing here. The object is Bethany. That is available. Through this is also available. I can, in fact, expand here and see all the values of the fields of Bethany, of our Bethany object. If I do another F10, you see that now the number of hours worked is also shown here in the Autos window. Currently, it's still 0. If I now increase it with NumberOfHoursWorked, you see that it becomes 1, and it also becomes red. If I'm really interested in the value of the NumberOfHoursWorked, I can also add a watch on that. This will then appear in the Watch window, as you can see here, and then that variable, as soon as it comes into scope, is shown here with its value. So it's basically showing the same information as having the Autos window, but now you can basically focus on the values that you need to investigate because maybe they are causing an error in the application. Another window that's sometimes very useful during debug is the Output window. Typically, all errors that occur will also give some output here in the Output window. Make sure that the output from debug is shown, and then any errors that occur will typically also show up here, possibly with extra information. This should already give you enough handles to work effectively with the debugger in Visual Studio and solve any exceptions that you may be getting in your application.

Writing a Unit Test

Debugging can help us greatly in pinpointing bugs that we've made in our code. While debugging will help us greatly, it's of course better to avoid having bugs in the first place. Writing unit tests can be a powerful tool in this area. Let's understand what they are and how we can create them in our application. Our code is working fine. We don't touch it anymore, right? Well, that would be an option, but that's not how things work in the real world. Very often, we'll have to revisit code that we've written before. And when we need to do so, the entire circle of having to debug and fix errors that we may have introduced starts all over again. Since we'll have to make changes to existing code often, definitely in real life applications, that becomes a time-consuming task. Through the use of unit tests, we can write code that tests other code's behavior. Maybe then make a change to our code. We can execute a unit test again to see if the behavior has changed. We are harnessing our existing code. We'll see if its behavior has changed, and if so, we'll be alerted of this. Unit tests are very simple to create, and it's again a task you will often do when writing C# applications. I say writing because indeed, as said, a unit test consists out of code, code that will test on regular code. Say we have a method that calculates the wage of an employee, and that's a method we've created before. Wriiting a unit test for that method will involve calling the method passing, in a set of parameters, and validating the result returned by the method. If we alter our original method and break its behavior, the result returned to the test method will also have changed. And that's what a unit test will flag. In general, unit tests are small pieces of code. The smallest bit of code we can typically invoke in C# is a method. So mostly we'll write unit tests which will invoke our methods, and they will, as said, validate the value returned by invoking the method. Unit tests should test our own code. We should test our code as much as possible without any external influence of other code. Now, writing unit tests will take time. It is a set code that we'll need to produce next to the actual application logic. While it will cost time initially to create your tests, they will make up for it afterwards, that's for sure. If you made a change to your code that breaks the original functionality, in other words, you've introduced a bug, you'll see that very early, as soon as you run your test, that is. While you can omit writing unit tests, having them will make you confident about changes you make to existing, perhaps already tested code. If you run the tests and your tests don't flag errors, that means that the functionality hasn't changed or hasn't been broken. You'll therefore improve the quality of the software that you deliver to your customer. Oh, and one more benefit. Having unit tests can also be seen as code documentation. I've already said it a few times. Unit tests consist of code that we need to write, and they will test our original application code.

Unit tests will thus invoke a consumer application code, and therefore, they will typically be added in a separate project, a unit testing project. Visual Studio and also the CLI come with templates for unit testing projects in which we will then write our unit tests, including NUnit and xUnit. These are unit testing frameworks supported by C#, and their task is making the writing of unit tests easier, and we will use xUnit in the demo. This is also the first time that we come across a situation where we need to bring in a second project in our solution. The unit test code will live in a second project, and that code that project will need to call code in our original project. Our actual application code is by default only accessible from within the original project. If we want to get access to this code from another project, we need to bring in what is known as a reference from the consuming project, so here the unit testing project to our original project. A reference basically opens a connection from one project to the other and gives access to the code and its classes within that project. Using this approach, we can write unit tests and invoke methods in our application. That's exactly what we need to write our test. Once we have the project created, we can start writing unit tests. As said, a unit test will typically test a method. It will invoke the method. It will do so by passing some parameters, and once invoked, we can see if the result returned by the method is what we were expecting. In that very sentence, I've, in fact, set the essence of a unit test. First, we need to set up some data, that is, the arrangement of the test. Once we have all we need, we can invoke the method. That is what is known as the act phase. Once that is done, we can in the assert phase of the test see if the data returned by the methods is what we expected. A unit test will always follow this structure. Now, I'll see them in action in the demo in just a minute, but before we go there, I want to show you a typical unit test. A unit test is itself often small and consists typically out of a method, too. In that method, we will perform the arrange, act, and assertion. Here you can see I'm setting up the data needed for the test. Then I'm invoking what we want to test here. That is the method I want to invoke and thus test. Finally, we'll include one or more assertions. And in the assert, I will validate the returned result. We can do this because we know what we passed him in the arrange part. In the assert part, we can check if the result is equal to what we expected it to be using Assert.Equal. Other assertions, of course, also exist. Finally notice that on the class and on the method, there is what is known as an attribute. And we haven't covered this in this course. For now, just remember that these are used for the unit testing framework to be able to find the tests in the project. Once we have one or more tests written, we can use another tool in Visual Studio called Test Explorer. It can search for all tests in our code, and it will actually use the just

mentioned attribute to do so. Using Test Explorer, we can trigger our unit test. So without running the application, we can validate if our code works fine. If the test succeeds, they will get a green light. If not, we will see a red light.

Demo: Creating a Unit Test

In the final demo of this module, we will go and create our first unit test. We'll start with the creation of a unit test project, and then we'll write a unit test. We'll then trigger it using Test Explorer. We are going to write our first unit test now. Now, unit test is code that is going to test auto code. So in our case, we're going to write unit test for our Employee class for these two methods here, these two methods I want to test. All right, let's do this. And since they will contain code, we need to write a class somewhere. As I said, the unit tests will live in a separate project. I hope you remember that this is our project, and the project is part of the solution. The solution is the container for one or more projects that work together. The unit test project is therefore also going to be part of the solution. So I can right-click on this solution and say Add, New Project. That is going to be a unit testing , s in this visit here to add a new project to the solution, I can search for units, making sure that C# is still selected, and then I get a couple of options. Different unit testing frameworks exist and Visual Studio supports most of these. I typically tend to use the xUnit testing projects for the xUnit testing framework. So I'm going to select that. I'm also going to give this project a name. Typically I give the same name as the main project suffixed with .tests. That's a good name to indicate that this project will indeed contain tests. And of course, this is going to be .NET 8 as well. It is going to contain code, and it's going to be also .NET 8 code. At this point, the unit testing project has been created, and it's going to contain tests for code in our main project. So I'm going to have to write code that uses code that lives in my main project. In order to be able to access that, I'm going to have to create a reference. We've already used references. But this time I'm going to create a project reference from this project to the main project. I'm going to right-click here on Dependencies and say Add Project Reference. In that dialog under Projects, I'll select BethanysPieShopHRM. That's our main project in the solution. I'll click OK. And now if you look, you will see here on the project that the reference to the main project has been created. By the way, if you expand Packages here, you'll see, amongst others, that xUnit is in there. Indeed xUnit is, in fact, nothing more than a NuGet package that is added to this project, and it was added by the template by executing that your own Visual Studio. Now, what the template also did was

creating this class already UnitTest1. This will contain our tests. This will contain code that tests code in the main project. I'm going to write a few unit tests for the Employee class. So, a good idea is then to also name this EmployeeTests, as it gives away that this will contain tests for my Employee class. All right, let's go to our Employee type for now. And as said, unit tests will test small bits of good, small amounts of code. Typically, the smallest bit of code that we can invoke is a method. We can invoke a method from pretty much everywhere when we create in this case, an Employee instance, so I'm going to test here PerformWork. That's the method I want to test first. Now what does PerformWork do? Well, it accepts a number of hours, that's a parameter, and it will then increase the number of hours worked stored on my object with the number of hours passed in. In other words, if I want to write a test for this method, what I need to check is that this calculation actually works fine, that the NumberOfHoursWorked after executing this method is indeed the sum of the old numberOfHoursWorked, plus the passed in one. That's what I'm going to test here. So let's write our first unit test together. This is a test method, and the unit test is, in fact, nothing more than a method. I need to give it the name that describes very well what this test will do. Well, under Employee, I'm going to call the PerformWork method, and I'm going to test that it adds correctly the NumberOfHours. So what I'm doing here is I'm giving this method a name that describes what this test, what this method will actually do, and it will test PerformWork, and it will check that the NumberOfHours adding works fine. Now, one thing that we haven't looked at is this Fact between square brackets. Now this is what is known as an attribute, and this is needed for the unit testing framework, so for xUnit to pick up on this test, so it knows that a test is coming. Now, in my unit test, as mentioned, I'm going to have to follow the triple A, or the AAA approach. First, I'm going to arrange, then I'm going to act, and then I'm going to assert. What am I going to arrange here? Well, I'm actually going to work with employees, so that is definitely a piece of the setup, that is a piece of the arrangement I need to do. So let us start with creating a new employee. I'm getting a red squiggly. And also, as you can see here, it doesn't say bring in just a using statement. Why is that? Well, something must be wrong on an Employee class. Let's take a look. While the Employee class is set to internal, classes which are internal are only available within the assembly, so within this project, if I need to test this, I need to make this public, so an external project can also work with this. So I need to make this a public class. When I go back to my unit testing project, I go to my little light bulb here. I can now bring in a using statement, and that will make employee known in my unit test. So I'm going to write here Employee employee is a new

Employee, passing in some data here. Also, DateTime isn't known here. Let's use the light bulb again to bring in system. There we go. So now we have our Employee. Let's go back to the Employee for just a second. Now, first, I do notice here a red squiggly. Address is complaining. Why is that? Well, it is because this class is also internal, and it's now used from a public class that also at least needs to be public. So I need to change that. And that's not what I wanted to talk about. I actually wanted to talk a bit more about PerformWork. Now, PerformWork also requires NumberOfHours, and that's a parameter that we need to pass in. And that is still part of the setup, part of the arrange. I'm going to set the numberOfHours to 3. I'm just creating a local variable numberOfHours, and I set it to 3. Now, this is hard-coded. I'll use that later on in my assert to see if the result is okay. Now I'm going to act. Now I'm going to invoke the code I really want to test. Now, what do I want to test? Th employee.PerformWork, passing in the numberOfHours. Now, initially, numberOfHours for an employee was guaranteed to be 0. Because it's a new employee, we expect that to have no worked hours yet. Now I PerformWork, and that takes in this 3. Now in the assert, I'm then going to check if the code behaves correctly so that the total numberOfHours worked for the employee should be equal to what we have passed in here, numberOfHours. That I can do indeed using an Assert. Using Assert.Equal, I can test that the numberOfHours here is equal to the employee.NumberOfHoursWorked. That is the value that is going to be influenced over here on the employee. So if that is equal to 3, after invoking this method, well, then my test will get a green light. Let me add another unit test and take you through it. And take a look at the name. I've called this method PerformWork_Adds_DefaultNumberOfHours._IfNoValueSpecified. In other words, I'm just going to call the PerformWork method on an employee, and then in my assert, I'm going to check that the numberOfHoursWorked for that employee is equal to 1. One, of course, coming from that constant value defined on employee. So I'm going to test that the numberOfHoursWorked on the employee is after a single call to PerformSork set to 1. Those are my two unit tests. Now, we need to run these tests. Now, for that we can go to Test, and then we can click here on Run All Tests.That will open the Test Explorer. The code is compiled, and we indeed get a green light. We have two unit tests that passed successfully. If we now make a change, let's say that we go to employee and we make a mistake here. We increase the numberOfHoursWorked with 1. We've made a change to the functionality, and we probably have broken something. Our unit test can now tell us that. Unit test can now be seen as a signal to quickly let us know that we've made a change that changes the behavior of

this code. If we go again to Test, Run All Tests, the test will run again, and notice what happened now. We're getting red lights. This is now a signal for me that I've broken the code, that I've changed the functionality, and the code isn't behaving as it did before anymore. I need to go back and inspect what changes I've made. This is how unit test will alert you that you've made breaking changes to the behavior of your code.

Summary

Another module done! Great job! In this module, we have learned a lot about the debugger and about setting breakpoints. It's an invaluable tool that allows you to test our code line by-line, but also inspect values at runtime. Next, you have learned about unit tests, and we've learned that these can create a harness around our code so that when we make changes to the code, we can validate if we haven't introduced any new bugs. In the next module, we're going to learn how our application can work with file.

Reading from and Writing to Files

Module Introduction

You are now already well-versed in object-oriented development in C#, and we've applied it to our application. And so far our staff management application stores the employees in-memory, which means that every time that we close the application, all the data is gone. The application isn't persisting the data anywhere. In this module, we'll change that, as you will learn how to read from and write to files in C#. This will allow us to store employee data in a physical file stored on this that can then in a subsequent session be read into memory again. In this module, we have just two topics. To work with files, we need to learn about a few classes built into .NET that allow us to get access to files within a directory. Once we have access to the file on this, we'll learn in the second topic how we are going to write the data to the file and read it into memory as well. In the demos, we'll do as said, we'll

store the employee information in a file and read that file again so that the information entered by the users of our application isn't lost.

Demo: Setting Up the Application Structure

Now, before we dive into this module and learn about files, I want to go to a demo. During these last modules of the course, we are going to be using our knowledge we have gained so far to work on a more complete, fully working application, and I want to show you the structure of that here already. We'll then add support for working with files in this application in the next parts. I've gone ahead and I've used the knowledge that we've gained in the previous modules to create a real application, and you see it here. Let's first try out the application, and I'll show you the code that I've already prepared. You can also find this code again in the start folder for this module. But no worries, I'll take you through it step-by-step. But first, let's take a look at the running application. As you can see, I've now used even some columns in the console. We have a title, then we have a TODO, and it's something that we'll fix later in this module, it has to do with file access, so that's the goal of this module. Then I'm also saying that currently I have no employees loaded in the application. And then I'm getting this menu here. I can choose one of the actions. I can choose between 1, 2, 3, 4, and 9 to perform an action within the application. I have already gone ahead and created the code for 1 and 2; 3 and 4 we'll do later in this module. So what can I do with the application while the application allows me to register employees? So I can create an employee first, I can select here 1, hit Enter, and then I'm getting this option to create an employee. In the previous modules, we have created the employee hierarchy, we have the base employee, but we also have a manager, a store manager, researcher, and junior researcher. Those are the types of employees I can create. So also here, I'm offering the option to select one of the types that I want to create in my application. So let's say that I'm first going to create an employee. So I'll again select 1. I'll enter the first name, the last name, the email, the birth date, and the hourly rate. And now the application also says that I now have one employee loaded in memory. I can add multiple employees. Let me do that quickly. And if I now select 2 here, I will get an overview of all the employees that I currently have in my application. Three or 4 are not implemented yet. So as you see, we're getting a TODO. And the same goes for 4. If I make an invalid selection, so say 8, then I'm also getting this invalid selection message. And finally, I can also quit the application by selecting 9. Now, let me take you through the code that I've already prepared for this. There is really

nothing new. This is all stuff that we've already seen. But of course, I'm going to take you through it step-by-step, so you're following along. We have two files again that are running the application. That is the program. That's, of course, again, used to start up the application. In the program, I'm keeping track of a list of employees. Because I don't know upfront how many employees I'm going to be entering as a user, it's best that I select a list. If I would use an array, I would need to know upfront how many employees we could have possibly in the application. And I don't know that, so I think a list is a better fit here. Then I'm writing the header to the console, and I'm doing that in green. And after that, I set the foreground color back to white. Then I'm calling Utilities.CheckForExistingEmployeeFile. Indeed. I've brought the Utilities class back. As you can see here, it's the same class as we've used before in which I have a couple of static methods. One of them is the CheckForExistingEmployeeFile, which currently has no implementation yet. It just prints out some stuff to the console that has to do with file access, so we'll look at that in this module. Then I've created a launch do...while loop, as you can see here. I've selected here the do...while, since I want to at least print my menu once, then a do...while was a great fit. I'm showing first how many employees we have. That's what I'm doing on this line here. Then I'm showing the action menu again with some plain Console.WriteLines. I'm capturing the user selection in this userSelection string, which I then use in this switch statement. The switch statement is responsible to know where to go next. In the case of 1, we're going to go to the Utilities.RegisterEmployee, which I'll show you in just a second. You have to view all the employees, save, and load employees. Also notice case 9. I have no code in there, I'm just breaking out of the switch. And if the user has made an invalid selection, well, then we go into this default block here which will simply write to the console that the selection was invalid. And I'm doing that until the userSelection was not equal to 9. So as soon as the user enters 9, then we exit the do...while, and we go to the "Thanks for using the application" Console.WriteLine statement. And for now, I also want to show you the RegisterEmployees. Notice that I'm passing to the RegisterEmployees, as well as to the other methods the Employees variable here. That is the list of employees that I defined over here. I'm going to show you the RegisterEmployee method we just mentioned, except a List of Employees. So again, there's nothing new in this method. So first, I'm creating the submenu where the user can select which type of employee they want to register. This time, I've used the escape characters \n to create a break, so that I can just do one WriteLine that shows all the menu options in one go. It's a bit cleaner, maybe. Then I'm accepting the userSelection over here. If the user has entered a value not equal to 1,

2, 3, 4, or 5, then I'm writing an invalid selection to the console and then I return already here. That's an easy trick to basically break out of this method earlier. Because the user has made an invalid selection, there's nothing else I can do, so I don't need to execute the rest of this method. So I simply put a return over here already. Now let's hope that the user makes a correct selection, and then I will ask for the other data. That's what I'm doing here. I'm going to ask a first name, last name, email, birth date, and hourly rate. And we are going to expect here that the user is entering correct data. Now, I have all the data to create the employee, but I'm asking the user which type of employee they want to create. So what I'm going to do is I'm going to create an employee reference. Remember, the employee was the base of manager, store manager, researcher, and junior researcher. So I can easily create an Employee variable. That's what I'm doing here. But based on the type of employee that the user wants to create, I'm going to either let it be an employee, a manager, a store manager, a researcher, or a junior researcher. Based on the selection, I'm calling a different constructor, passing in the data that we have received. Then I add that employee, whatever type it is, to the list of employees. It's a generic list of employees. It will only accept employees, but all the other types, manager, store manager, and so on, they are also employees. Remember, there is a relation, every store manager is an employee. And to view all employees, I also want to show you quickly. That, again, expects the list of employees to be passed in, and then it's going to loop over all the employees and it's going to call the DisplayEmployeeDetails for each of these. But again, relying on inheritance here, all the employees, no matter what type they are, have the DisplayEmployeeDetails method defined because it was defined on the base Employee type. Now, in the next demos, we are going to extend the application so that we can save the employees to a file and so load them also again in the next run of the application from a file that we've written them to.

Working with Files from C#

All right, as said, let us start with understanding how we can access files from our C# code. Many applications will have a need to work with files, and our application can benefit from it too. If you're creating a word processing application, then you'll need to store the entered text to a file. If you're creating an application that allows a user to create or edit an image, that image should probably be stored in a file as well. And of course, also, custom line-of-business applications like ours can work with files to store information entered by the user, which can then later on be read out. Applications

like ours will either store this information in a file or in a database. In this course, we'll look at how we can store it in a file as said. In C# applications, we can work with the classes of the System.IO namespace, part of the .NET library to work with files, paths, and directories. The Directory class has methods available to allow us to create directories or check if a directory already exists. We'll see what these methods are in just a second. Using the File class, we can create a file, copy a file, move a file, and so on. The Path class, also part of System.IO, can be used to work with strings that contain information of the path for a file or a directory. Next to these classes, an alternative set exists that you'll also often see being used, and those are the FileInfo and the DirectoryInfo. I will stick to the File and Directory classes in this course. As said, the Directory class allows us to work with directories, so folders on the local system. Say that in our application, we want to store the employee information in a certain file. We'll probably want to store it in a specific directory. We can, from our application, create that directory using the CreateDirectory method. That method accepts a string that contains the path of the directory we want to create. Now CreateDirectory is a static method. Do you remember what being a static method means? It means we can call it directly on the class name, so we can use Directory.CreateDirectory. We can, before creating the directory, check if it already exists using the Exists method, passing in, again, the path. And deleting a directory can also be done using the Delete method. I think that's a good name for it. The File class also has quite a few methods onboard that we can use to do file interaction from our C# code. Using the Move method, we can move a file, the source, to another location, the destination. Copy is similar, which won't touch the source file. It'll just copy it. Just like with Directory, we have an Exists method that we can use to check if a file exists before creating a new one. And Delete, well, yes, that does what we expect it to do. The File class also has methods to work with file contents. The WriteAllText, for example, is one of these. We can use it with a path that points to a text file. This will open the file, read all the text, and then close the file again. In a very similar way, the WriteAllText method can be used to write a string to a file. The text passed in as parameter is the text that will be written to the file. We'll see both of them in action later in the module. Here you can see a small snippet that uses the File class. First I have, in a string, the path of the file I want to work with. Then, using the File.Exists method, we can see if a file exists. That is a static method as said, and it will return a Boolean value. Only if the file exists will we call the Delete method on it, again, passing in the string path to the file.

Demo: Working with the File and Directory Classes

Let us returned to our application and see how we use the directory and file classes in there. In my program, I have made the call to Utilities.CheckForExistingEmployeeFile. I want to at start up of the application check if the file is already there and give the user notice about the file with employees being filed or not. So let us write a code for that method. So I'm going to go to this method here in the Utilities class. I had already written a TODO, so a bit of placeholder code, so let's remove that. And now I'm going to check if the file already exists. I've created on the Utilities class two fields of type string, one that contains the directory name, and one that contains the fileName. I have in this case indeed made those fixed. Notice that I'm using the verbatim notation so that escape sequences are ignored in the directory path. So we're going to check if the file already exists first. The complete path of the file is the concatenation of the directory name and the fileName. So I'm going to use the string interpolation syntax again to concatenate these two. That is the complete path. Now, I'm going to create a Boolean, and that is going to get the result of using the File class part of System.IO. So I'm going to use File.Exists, and I'm going to pass in that path, and it is going to return me a Boolean value. This is a static method on the File class that expects a path and will return a Boolean. It will just see if the file already exists. If that's the case, well, then I'm going to write something to the console. If the file cannot be found, well, then I'm going to already create the directory. So I'm going to use the Directory class. I'm going to use the Directory class, also part of System.IO, and I'm going to use the Exists call, passing in the directory path. So if the directory cannot be found, I'm using indeed the not operator here, so the negation, then I'm going to create the directory already. I'm going to use the Directory classes again, passing in that directory string again. Let us, in this case, also write something to the console that we've created in the directory. I'll do that in a specific color, and I'll reset the color of the console. If you take a look at this path now, so D:\data\BethanysPieShopHRM, doesn't exist yet on my machine. So let's run the application because in the program, we're calling this code, so this should now automatically check if the file exists. And if not, it will create the directory. Let's go ahead and try this out. Now, we do see the message here that it says directory is already for saving files. Indeed, we have a data directory, BethanysPieShopHRM, but there's nothing in there yet. That's normal because we haven't created the file that we'll do in the next demos.

Reading and Writing Text

Next, we'll learn how we can work with the contents of the file. This will allow us to write the employee information to a file and read it in again later. To work with the contents of a file, we have several options. We've already touched on the File class. That class has simple methods that allow us to read text and write text to a file. In a text file, our data will be stored as strings. In other words, a series of text characters. If you want to, for example, store our Employee instances in a file, you will convert this to a string in a fixed format. Each Employee entry will then need to be separated from another one using a known delimiter. That delimiter could be, for example, a semicolon or a hash sign. When we then read the file again, we know we should look for that delimiter to split the string back into the different Employee instances. We'll use this approach in our application. Next to this approach, we can use FileStreams to work with files. Using a FileStream, we can work with both text files, as well as binary files. A stream in .NET represents a flow of data from one location to another. A FileStream can thus be used to read a file into memory or to write data from memory to a file. Finally, we have the StreamReader and the StreamWriter classes, which can be used in combination with the FileStream to work more easily with reading and writing text. Now using streams is a bit more complex, so we'll stick to using the methods provided by the File class. Here, you can see a snippet to read text from my file using the methods provided by the File class. I'm using the ReadAllLines method in this case, passing in the path. Now, we haven't talked about this method yet, so what does it do? Well, it doesn't read all the contents of the file into a string. Now, instead, it will return as an array of strings, each containing a line of text of the file. After it is done reading, the file will be closed. And on this snippet here, we're writing all text to a file. The string that is passed in, so here, and employees, is the string that will be written to the file, and if the file already exists, it will be overwritten.

Demo: Reading and Writing Text

In the second and last demo of this module, we'll see how we can write text to a file and read it afterwards. So now we need to complete the application and write an implementation for SaveEmployees and LoadEmployees. Let's do saving first so that we can see that the file containing our employee information will correctly be saved. So I already had a placeholder here, so let's remove this code. And let us now write the code to save the employees to a file. Now, what I'm going to do is I'm going to loop through the list of employees and write each employee to a line of text. I'm going to convert each employee to its string representation. I'm going to create my own string representation

here. I'm going to write each employee to one line. I'm going to do things here manually, so I'm going to create a string manually for each employee, and we could also use the JSON notation that we've seen before and write those to a file as well. But let's do things manually here. So I'm again going to concatenate the directory and the fileName to create the part where I want to store the file. And since I'm going to be concatenating quite a lot of strings, I'm going to use the StringBuilder here. We already know that bringing in the StringBuilder requires us to add a using statement. To bring a support for the system, text namespace. There we go. I'm going to use the StringBuilder here because I'm going to be concatenating quite a lot of strings, and then the StringBuilder really shines. Visual Studio is already giving me a good suggestion. I'm going to be looping over the employees, and a foreach is a good suggestion here. Let's create that foreach loop. In the foreach loop, I'm going to convert each of these employees into a string, and I'm going to use for that the StringBuilder. Let me paste in the code because this is pretty simple stuff. So notice what I'm doing here. I'm using a couple of appends on my StringBuilder. Each of these appends has the same format, more or less. I'm using first the name of the property. So, firstName. In fact, I should say it's the field. Then a colon, and then the value, and then a semicolon. So I can later on split on the semicolon to see that the first part is the firstName and the lastName, the email, and so on. And then at the end, I do a new line so that each employee will be one line in the file. Now, we do have one issue. My list of employees could contain managers, employees, store managers, and so on. How do I know which type of employee I'm actually saving because I'm getting them all in as the base type? Well, for that, I'm going to add a simple method to convert the type into a string. Let me show you. I've created here a small method that expects an employee as a parameter, and based on the real time that employee is, I'm going to return a string. Notice I've made this method private. This is definitely internal to the Utilities class that is also hidden and is therefore also encapsulated fully in this Utilities class. I'm introducing something here that we haven't used, the is keyword. The is keyword can be used to check if an object is of a certain type. I'm going to in a very readable way ask C# if the employee is actually a Manager. If that is the case, then I return the string 2. If the employee is a StoreManager, I'm going to return 3, and so on. Now, every instance is, in fact, an employee, so we need to move the more specific types to the top because everyone is an employee. If I would put that line first, well, then I would always get back 1. Now, I'm asking first are they a manager, store manager, then junior researcher, because that is, again, more specific than researcher, and only then employee. This method I'm now going to call from my foreach

loop. I'm going to call the GetEmployeeType, passing in the employee, and I'm capturing that in a string here. I'm then also going to write that as part of the line I'm saving to my file, using in this case as the identifier type. Now, at the end of this foreach loop, I have all my employees nicely represented in one string inside of the StringBuilder. So I can now use the File class again and call the WriteAllText method. I'm going to pass to that the part to where I'm going to save, as well as the contents of the StringBuilder. I'm calling my StringBuilder ToString. If that all goes well, then we'll write to the console that you have successfully saved the employees and we'll reset the color again. All right, let's try this out. I'm going to run the application and see if our employees get saved to the file. So I'm going to quickly register a few employees. So I now have two employees in-memory, and let's now click on 3. And success! Our employees are saved to a file. Let's verify that. Here's our folder again. Indeed, there's a file in there. We can now simply open that with Notepad. And indeed, we see the string for each employee. Each employee is on a new line, that is. I have the first name, the last name, and so on. And also notice that type is at the end. First I created an employee, so that's type 1, and then I created the manager, and that's type 2. Close the file, and also close the application. And the last thing that we'll need to bring is now also the functionality to load the employees again from the file. So we'll go to the LoadEmployees method, remove the placeholder code, and then I'll paste in the code because it's quite a lot of passing I need to do. I'm going to take you through the code again. You can find this in the snippets. Because it is a lot of code, I'm not going to type it all. First, let's focus on the file check. I'm going to check first before I'm going to load the employees if the file effectively exists. If the file exists, well, then I'm going to clear the employees currently already loaded in the memory of the application. That's what I'm doing here. Then I'm going to use again the File class to read all the lines. All the lines are loaded into a string array, and that is the employeesAsString here. The ReadAllLines effectively returns a string array, so you will need array knowledge. Each employee will basically be a string in this string array. Then I've written some passing code. How am I now going to read this file? Well, I'm actually going to split the line based on the delimiter. The semicolon is the delimiter, and then I will have all the parts, so firstName:Bethany, lastName:Smith in-memory, and then I need to take this part, the actual value. So once I have the different parts, I will substring them so I can take only the part I need to recreate the employees. That is the code you see here. And it might look a bit complex, but it's absolutely not complex. First, I'm taking the employee, which is this part here, and I'm splitting that, I'm calling the Split method, it's a method that we haven't looked at,

but I'm basically going to split the string into separate smaller strings based on the delimiter. That's going to return me again an array. And then I will have firstName:Bethany, lastName:Smith. Now I need to search for the first name. And remember this part is going to look like this firstName:Bethany. I'm then going to split this part based on the colon, and I only need this part. So I'm going to say take from this entire string the Substring which starts at colon, but actually only start at the next character. That's why we see the +1 here. That will return me the firstName. And I'm doing exactly the same for the lastName, the email, and the other values, the DateTime, and the hourlyRate, they need to be passed. Then I have all my data. I can then based on the value of EmployeeType, again, a 1, a 2, a 3, 4, or 5, create the employee as a new employee, a new manager, and so on. And then I load them again into the employees list. And then write to the console. Then I've loaded the data again. Let's try this out. So we have the file. You don't have any employees right now, but I'm going to select 4 here. We now have two employees loaded in-memory. Let's view the employees we currently have. And there we go, we've successfully loaded the data again from the file.

Summary

And here's a recap of what we have seen in this module. We have learned about the file and directory classes, which are built into .NET, and which we can use from our C# code to work with files and directories, including creating files, checking if directories exist, and so on. Using the File class, we also wrote text to a file, and we've also used methods on the File class to read that text again. Even though we can try to write perfect code, at some point, things will go wrong. In the next module, we'll see how we can handle exceptions that happened in our application gracefully.

Handling Exceptions

Module Introduction

Although we do our utter best to write great C# code, things will go wrong while our application is running. And we can make mistakes, things that we didn't think of might happen in our application, but also things beyond what we as developers can control might happen. We should, in fact, write our

code so that it can handle any errors that occur while executing. If possible, we should try to handle errors, or at least show to the user that something went wrong instead of just crashing entirely. In this module, we will learn about handling exceptions in our code. Let's see what we are going to learn in this module. We'll start to learn about exceptions and how C# and .NET go about this. We'll then learn about the try/catch block, the language feature in C# that allows us to work with exceptions and handle them gracefully. We'll also learn that we can handle different types of exceptions in code. And finally, we'll learn about finally. ore about that by the end of the module.

Understanding Exceptions in Code

So to get started, let's first try to understand the concept of exceptions a bit more. Like I mentioned in the introduction, no matter how much we try, we will fail at catching all types of errors in the code we write. Errors can and will occur while applications are executing. We can think of an error as any problem that may occur and will prohibit the application from executing normally. In most cases, these errors will cause an application crash, which of course is a bad experience for the user of our application. Different things can cause our application to fail. Maybe we have failed to validate user input, and that user may have entered a 0. If we then use that value in a division, we'll get an error since we're trying to divide by 0. By no means can the compiler catch this, so this will be a runtime exception. But sometimes the error that will occur will be completely out of our control. A file we may want to write to may be locked by the operating system, or the user trying to access the file may not have the required permission to do so, or a database we want to access might be unavailable because of a power outage in the data center. We can go on and on and on. The list of possible causes is pretty long. If we don't take any action in our code while running our application from Visual Studio with the debugger attached, we'll get the experience that you see here. Visual Studio will break on the line that caused the error, and it will show exception details. We have already seen this when we were learning about the debugger. If, however, we run the application without the debugger attached, the application will just crash without any useful information for the end user. There must be a better way to handle this.

Using a try/catch Block

In C#, we can write code that will handle exceptions using what is known as a try/catch block. Let's learn about this important language construct and its keywords. C# has a structured way of handling errors. Instead of wrapping each line of code that could possibly result in an error with error handling code, we can use structured error handling. In this case, we let our application code execute, but it will be wrapped, let's say, covered by exception handling code. In other words, our application code will look as before; it'll just be wrapped. If exceptions do occur in that block, these will be called, and we can handle them. This is based on the use of the try/catch block in C#. Let's look at this. You can see the structure of the try/catch block, which will lead to using structured error handling in C#. In most cases, when we use this, we will have a try, as well as at least one catch, although that's not even required. The code that we basically want to attempt and that could throw an exception we'll put inside the try block. A try block is surrounded again with curly races, and it can contain as many statements as we want. This means that all code that may result in an error happening and .NET generating or throwing an exception will go here. So including code that writes to a file, works with the database, and so on. If all goes well, then you basically won't notice a change compared to not having a try/catch block in your code. The code will just execute as before. However, if things do go wrong and an exception is thrown, execution will jump out of the try block and into the catch block, which is located below the try. So if something goes wrong in the code wrapped inside the try, the catch will be next to be invoked. We get an exception, and an exception is an object created by .NET which contains all the information about the error that occurred. So we're getting a lot of information here about what went wrong. We'll see in the demo in just a minute what exactly we're getting. The way that we handle the exceptions in C# also means that the normal code is not cluttered with error checks and the like. All error handling happens here in the catch. As you can also see here, the exception I'm getting in is of type exception, and that is a built-in type in .NET again. In many cases, the exception that is drawn by .NET will have even a more specific type. There's an entire hierarchy of exceptions, which I will show you. But here we are catching the base exception. Exception is the base type for all types of exceptions in .NET. In one try catch, we can, in fact, handle the different types of exceptions with different code. In that case, we can include multiple catch statements, and each can then handle a specific exception type. Based on the exception that occurred in the try block, .NET will search for the matching catch block and will automatically select that one. Here you can see an example of using a try catch block. I'm asking the user for input using Console.ReadLine. We have as developer with this

approach little control over what the user will enter. Maybe they will enter a number, maybe they enter an entire book, we don't know! On the next line, things can actually go wrong. I'm just calling int.Parse on the input. If the input was a valid integer, well, things will go through. But if the user entered something different, something that our code wasn't expecting, this will fail, and it'll throw an exception. Notice the terminology here. I'm saying that our code will throw an exception. The .NET runtime is closely monitoring things at this point and will see the exception occurring. The exception here will be of type FormatException, and so a matching catch block will be searched and hopefully found. Control will then be transferred to the catch block you see here as well. The code in here can now execute. We have the ability to handle the error gracefully so that we could show a piece of text to the user warning that this value wasn't entered in the correct format. And after finishing the catch statement, flow of the application will continue on the next line after the try catch. The application hasn't crashed, and that's a very important aspect here, too. By introducing a try/catch block, we not only can give feedback to the user or in general handle the error, the application also hasn't crashed and will continue to execute. Now, you may be thinking now, should I wrap all my code in a try/catch block? A totally valid question. And let me see how I can answer it best. It's definitely a good idea to add a try/catch block where things can go wrong. It's better to add one too many than the other way around. However, a try/catch block will bring with it a performance hit. Although in most cases, it is pretty limited. Knowing when an exception might occur in our own code is probably pretty simple. But when you are using types that come from the .NET class library, it's not always easy to know which type of exceptions might be thrown. Say that you're working with the File class, which we used in the previous module. How do we know what type of exceptions might be thrown? Well, the good news here is that the Microsoft documentation shows the possible errors thrown by these types. That allows us to write one or more specific catch blocks for these types of exceptions.

Demo: Working with try/catch

Let us return to our application and introduce error handling using try/catch block. I'm going to give in our application our user an extra option. I want them to also select an employeeId and see the details of that specific employee. So let's add option 5 here and call that Load specific employee. Of course, we also need to extend our switch here with an extra case. It's going to be case 5, and I'm going to create a method on Utilities class LoadEmployeeById. As you can see at this point, we don't have that

method yet. We can, in fact, let Visual Studio create that method for us. If I just type the method name and it doesn't exist, I can hover over the light bulb again and say generate method. If we now go to the Utilities, we'll find that Visual Studio has created the method for us. And although it generated the method, we still need to write the code ourselves. That Visual Studio can't do yet. So we'll ask the user first here to enter the employeeId they want to see. That will be an Employee. And we'll search in the employees list. We can basically also use the index method here, so we'll pass in selectedId, so on that employee, I can now simply call the DisplayEmployeeDetails method. Pretty simple. Now, there is a possible error here. Let us try it out. We are going to be a good user and we're going to load in the employees first from the file. So we're going to select option 4. That loaded into employees. Now I want to see the second employee, so ID 2, there we go, that works. But if I try that again and I select abc, then the application blows up and throws us a format exception saying that we tried parsing something, abc in this case, that could not be parsed into an integer. Now because the debugger is attached, we went into break mode, we've seen that already, but of course, if the user is using the application, they won't have Visual Studio, and the application will just silently crash. We are going to try and catch that exception and handle it gracefully using a try/catch block. I'm going to, in this case, surround my code with a try block. So the code that I already had, I'm going to put in a try block. There we go. What I've done at this point is I've asked Visual Studio, okay, try out this code, and if it fails, well, then you can go into this catch block. I'm going to catch here a format exception. In other words, when an exception of type format exception happens in this code, then this catch block is going to be triggered and the code in there will execute. Now if the user enters something incorrectly and we do arrive in this catch block, we can simply say to the user, well, sorry, that is not the correct format to enter an ID. And when I try the same thing, we first load in the data, and then we go again to option 5, and we now enter abc, the application won't crash, it will handle the incorrect input gracefully, and it will continue running as well. After the exception was raised and the catch block was executed, the application just continued executing without crashing. And that's what we need.

Demo: Using the Exception Details

Now, we already have very basic exception handling in the application. The good thing about this way of working with exceptions is that the exceptions thrown by .NET are full objects created by .NET. That means that we get a lot more information than just an error code, for example. Indeed, the exception

contains several properties that give a lot of information about the exception that occurred. The Message property is the one you'll probably use most. It's a human readable error message that usually contains useful and valid information about the exception. We'll see it in just a minute. Very often multiple exceptions will occur. If, for example, something goes wrong with storing a value in a database, multiple exceptions might be thrown in that case. The exception that we're getting might contain a nested exception, available to the InnerException. Another useful property is a StackTrace, which will contain information about where the exception occurs. In a simple application like ours, we don't have a lot of nested calls. So, calls where one method calls on to the next and so on. In larger applications, the exception might actually come from a deeper level and then the StackTrace can prove very useful, too. Finally, the HelpLink contains a link with more information, a way you can read more about the type of exception that happens. In the snippet here, you can see that I'm using the information that I'm getting from the FormatException that was thrown. We write to the console the Message and the StackTrace. Now, in many cases, it's not useful to show the StackTrace to users, as it may even confuse them. However, this could be useful to write to a log file. The log file is a file where the application can write information while running. I will show you in the next demo how I can use this exception detail information. Now, we also should think of adding a try/catch block to the LoadEmployees method because in there we're doing File.IO, we're doing file parsing, and so on. So let us introduce a try/catch block here as well. In fact, what I'm going to do is I'm going to surround the entire if block that contains all the code that we've written in the previous module with a try catch. So, I'm going to go here and do a try, and search for the end of the if. You can follow this line here, that easily shows where the if ends. So there we go. Now all the code is in a try block. Now I'm going to add my catch block here. Now, one of the possible exceptions that we'll get here is that while we're reading the file, which we're doing, for example, here, the file is not found anymore. Maybe another process has deleted the file. In that case, we're going to get a FileNotFoundException. Let's try catching that. So here below the try, I'm now catching a FileNotFoundException, and I've called that exception if it gets thrown fnfex. This is the name of the instant of the exception that is going to be thrown by .NET. So if an error occurs, .NET is going to create an object of type FileNotFoundException, and this is going to be its identifier. Now, I can show some information to the user about what went wrong. In the previous demo, we just showed to the user a Message saying that the input was incorrect. If I now also want to give some more information to the user, I can do that

using the options on the exception. First, we can add some generic information, and next to the general information, I may also want to show to the user extra information about the exception. What do I want to show perhaps may be on the exception, so the fnfex object, I want to show the Message. The Message is one of the built-in properties only exception type that will contain a human readable piece of information about the exception that occurred. I may also want to show where the exception occurred. That information I can easily find in the StackTrace. And finally, we'll also want to reset the color. There we go. Let us now run the application. I'm going to cheat a little bit. I'm going to put a breakpoint over here, so after we've checked that the file does exist, and then I'm going to delete the file, and we'll see what happens then. So let's run the application again. We're going to try loading our employees from the file, so we'll select option 4. So now the file is confirmed to exist because we're inside of the if block. Now, I said I'm going to, well, instead of removing it, let's rename the file, and it's also going to cause an exception. Now, the file can't be found anymore. I'll just let it run now. Now notice what happened. First, the application hasn't crashed. That's already very important. Let me see my exception, so that's the one I wrote to the console. Then we see the Message, and then we see the Stack Trace. So the Stack Trace really gives me information that I can use to track where the error occurred. In this case, on line 126 in the Utilities.cs file. And that was this line. It was actually trying to read the lines from the file. But since the file didn't exist anymore, we got an exception.

Catching Several Types of Exceptions

So far, we had just one single catch statement. In many occasions, it will prove useful to catch multiple types of exceptions and handle these differently. We can always write a catch statement that catches the general exception type. That is the base class for all exceptions that can be called. However, if the code in our try block could potentially throw different types of exceptions, we may want to handle these differently per type. Here's an example of this. In the try block, I'm again asking the user for input, and I'm parsing that. We've seen this before. This could throw a FormatException. But no exception will be thrown by that line if the user simply enters 0. However, in the next line, we fail to check that that value is effectively not 0, and we divide by 0. That's of course not allowed, and that, too, will throw an exception, a different type of exception, though. It'll throw a divide by 0 exception. Note, and I have included also a catch block that catches this type of exception. I can now handle both type of exceptions differently so that I can also give the user a correct and meaningful response about

what happened. .NET comes with many types of exceptions we can use and catch. As we have seen, the documentation of .NET specifies which exception might be thrown. These exceptions all fit in a hierarchy where Exception is the base class. Here you can see a few of these built-in exceptions. As said, Exception is the base class for all types of exceptions. Below that in the hierarchy, we see two other Exception types, namely SystemException and ApplicationException. Many of the Exception types used by .NET, in fact, inherit directly or indirectly from the built-in system Exception, like the ones we already saw, including the DivideByZeroException and the FormatException. We can also create our own custom Exception types by creating a class that inherits from Exception. Now, since Exception is the base class of all types of exceptions, it can also be seen as a sort of a catch-all. Let me explain. Here's again the snippet we saw before, but now I've included one more catch, which will catch Exception to the base type. When now an error occurs in the try block and an Exception is thrown of a type of Exception that we're not specifically catching, control will go to this last catch block. Do note that the .NET will search for the corresponding catalog top to bottom. So if you would place this at the top, it would catch all exceptions, and the most specific ones would be unreachable.

Demo: Catching Multiple Exception Types

I'm going to in the next demo at support for multiple types of exceptions. Now, the code in the LoadEmployees, and in fact, some of the other code as well, can actually throw more types of exceptions than what we are catching. Now, at this point, we're only catching the FileNotFoundException. But maybe user has an old version of the file, and that maybe doesn't include yet the EmployeeType. And then we might be looking for employeeSplits 5, and it doesn't exist, and that would cause an IndexOutOfRangeException. All this to say that we probably will need to catch multiple types of exceptions. And we can do that. We can put here multiple catch blocks based on the type of exception that occurred while executing the code within the try block. The correct catch block will be searched based on the exception that was thrown. So let us, for example, also add a catch block here that will catch IndexOutOfRangeExceptions. So now I have multiple catch blocks, but there could be other things that actually go wrong, so we can add as many catch blocks as we want. And, in fact, we should always make sure that we put the most specific type of exceptions first. In fact, what I'm going to now add is also a general catch block that catches every type of exception if it's not handled by a more specific type of exception. The Exception class, the Exception type, that is, is the

most generic type of exception. All exceptions will inherit from the base Exception. And so any exception now and is not handled by a more specific catch block will be handled by this catch block, and still the application won't crash because we've called it correctly.

Demo: Adding a finally Block

In the final part of this module, I want to show you the use of the finally block, a third player in the try/catch statement. So far, if you have code that throws an exception somewhere in the try block, we'll probably have code in that try block that doesn't execute. That can, in some cases, be an issue. Imagine that in the try block, we have opened the file. When doing so, a log is placed on that file so that other processes can't access it. In normal circumstances, we can close the file at the end of the try block. But now imagine that something goes wrong, and we end up in the catch block. You would actually need to check in the catalog if the file log still exists, and if so, remove it, therefore, duplicating the code that we may already have in the try block. This can be solved by adding a finally block. A finally block will always execute, no matter if we went through the try block entirely or we went to a catch block. Here you can see the finally block. Finally will execute either after the try has finished successfully or after any of the catch blocks. The finally will always run and give us a place where we can include code that we need to be sure of it has executed. So closing the file safely and removing a log could be done here. In the final demo, I'll show you the use of the finally block. Notice that in all of the code execution parts, so in the try block, I'm doing this ResetColor, but also in all the catch blocks, I'm doing that ResetColor. It seems to me that that is something that we need to do no matter what path we followed. That we can put in a finally block. We can comment it out over here, and over here as well, over here as well, and over here as well. I can now add a finally block that no matter what code execution part was followed will be executed. So if we went successfully through the try block, this will be used, and also if one of the catches was executed, this will also get executed. I'll do here my Console.ResetColor. Let's put a breakpoint here and let's execute the application once more. Let's call number 4 hitting the breakpoint, and so normally nothing will go wrong. See, we successfully loaded our employees, and now we exit the try block, but we still enter the finally block to reset the color correct. If an error would have occurred and the catch block would be called, that finally block would also have been called.

Summary

We've reached the end of this module, and we now know how we can let our code safely execute and make our C# applications more robust. By adding a try/catch block, we wrap our code, and when something goes wrong, we can be sure that the catch block will take over. We've also seen the use of the finally block, which will always execute. In the final module of this course, I'm going to show you where you can go next in your C# journey.

Next Steps in C#

Learning More About C#

You have come a really long way learning C#. You're now well-equipped as a C# developer to get started building your own applications. Although you now already know a lot, there's still a lot more to learn. In this final module, I want to briefly highlight the most important areas of the C# language that you should look into next. Object-orientation is a hugely important topic. C# is first and foremost an object-oriented language. We have covered in this course already the different aspects of object-orientation, but there are definitely more things to learn in this area. With the OO knowledge that you have gained, you can start working on the design of a real system based on OO principles. When you start applying these, you'll pick up a lot of new things, too. We have in this course persisted data in a file. That works fine, but many applications will use a database to persist data. Databases allow us to work with what is known as a relational model, a representation of the data with relations between the different entities. C# and .NET come equipped with all we need to work with databases, both databases locally on our machine, or remotely hosted in the cloud. There are two main ways to interact with data in databases which come with the framework. ADO.NET is the oldest technology part of .NET that allows us to work with databases. The other one is Entity Framework Core, or EF Core for short. EF Core is what is known as ORM, an Object-Relational Mapper. Using it, we can map database tables to entities, classes really, in our code. Using both EF Core and ADO.NET, we can query a database or manipulate data. You'll want to learn more about how to connect with databases in your code as it is, as said, a very common task to do. LINQ is another important area that we

haven't covered in this course. LINQ stands for Language-Integrated Query, and using it we can write SQL-like queries directly in our C# code. These queries can be used to query all sorts of collections, like the List of T. But LINQ can also be used in combination with EF Core to write queries in code and let these be executed against the database. Very often, you'll therefore see EF Core being used in combination with LINQ. In our application, we have written our code in a regular synchronous mode. And yes, that is normal for our application. Now, why am I saying that? Well, next to synchronous programming, C# also supports asynchronous coding. Async coding you'll encounter when you are writing code that can take a while to finish, not because you've written complex logic most of the time. Now instead, because it can take a while for the process to be ready, this could be reading data from a database or launching a web service request to an external API. By writing our code in an async mode, we can avoid writing blocking code, meaning that our application can still continue to perform other tasks while the long-running task is still executing. This way, we avoid that our application hangs in the UI, which typically results in a bad user experience. C# has support to write async code in a very easy way, namely with the async/await pattern. C# is just a language, which allows us to write all types of applications. In this course, we are focused entirely on the C# language, and that is definitely the goal of this course. But in order to build more real-life applications, you'll need to apply your C# knowledge in combination with other project types and frameworks. If you want to build desktop applications, take a look at either WPF or WinForms development. WPF, short for Windows Presentation Foundation, is a modern framework for building rich UIs for Windows, including things like animations. WinForms is an older technology, but still very much alive and used for millions of applications. When you want to target the web .NET has many options where you can apply your C# knowledge. ASP.NET Core is the place to start. It's the platform to build web apps using C# and .NET. It actually has several options to create the UI for your web application, including MVC, Razor Pages, and Blazor. MVC and Razor Pages are what is known as server-side technologies, allowing us to render the HTML on the server using C# code. Blazor is a client-side technology that allows you to write to C# that directly executes in the browser. If you want to expose functionality through an API, which can be consumed by other applications you build or even a third party, ASP.NET Core can also be of help. Using ASP.NET Web API, we can create modern APIs that communicate using REST. Finally, also for mobile, your C# knowledge can be of help. Microsoft allows C# developers to write code using Xamarin or Maui, two frameworks that we can use to build mobile applications that run on

iOS or Android. As you can see, there's loads more to learn to grow your skills and knowledge as a C# developer. Check out the C# and .NET paths here on Pluralsight, as we have many hours of great content to teach you all of the above and much more. And with that, the only thing left for me is to congratulate you on completing this course. Please consider giving it a rating. Thanks for watching. Bye for now!