Delegates

- [Instructor] Delegates have been available since the start of C Sharp and actually offers usthe first and easiest way into asynchronous programming. So, there is a keyword in C Sharp called delegate for which the compiler will actually generate an entire class. So, with the delegate keyword, we get some methods that'll help us get to asynchronous code. Let me show you in C Sharp. I'll create a new project in Visual Studio. This is Visual Studio 2017, Community Edition. File > New > Project. Create a C Sharp unit test project.

And I'll rename our unit test. Okay, so here, we can start with the simplest case. And in the first demo, what I'll show you is just how to call a method using a delegate. You probably already know this, but just as a reminder here we have our method. Our method does something simple perhaps. Then we have very simple little method. I'll bring in System.Diagnostics for the Debug.WriteLine, and we'll run DoWork as our test.

So from the test menu, when we open the test window, here's the Test Explorer window. We'll compile our code. Control + Shift + B to build, and it succeeds, so it shows up here in the test list, and now I'll run the Debug version of our test. Okay, successful. And so in the output window, there is the Hello World text message. Now, when you're looking at Visual Studio,you may have a lot more text here in the debug window. Just pay attention to the Tools options, and under Debugging, you have the output window where you can switch off things like module load and module unload messages.

If you switch those off, it'll clean out this window quite a bit. You'll end up with the text that we wrote with Debug.WriteLine and a few other details that's still interesting. So, here we have the traditional method call. Our unit test Demo01 method is invoked by the testingenvironment, and then we called the DoWork method. So let's add a delegate. The delegate keyword asks us to describe what our method looks like. We would like a method that returns void. And we're going to take no arguments.

Semicolon, and that is the end of our delegate description. So delegate's the keyword. Void and parentheses describes what our method signature and return type looks like, and this is the name of our type, DoWorkDelegate. So instead of calling the method directly, we can use the type DoWorkDelegate, variable name m, short for method, equals new DoWorkDelegate.The DoWorkDelegate constructor here requires us to provide a method that matches the signature and the return type that is described by this delegate, which will be DoWork.

Now, notice, DoWork does not have a pair of parentheses at the end. M as a variable is a method of type DoWorkDelegate, and so we can call it using the compiler's convenience here by just placing the brackets at the end. So, with parentheses here, this is a method call,and so at this point, we are calling the method DoWork through the delegate named m as a local variable. Let me just confirm this still works. Output window, Hello World.

Works as expected. Now, the thing that a lot of people don't notice, it's not interesting from a day-to-day point of view, is that delegate actually generates a class. The C Sharp compiler based on that keyword will generate a whole class that will be our type-safe means of having a function pointer. So, function pointer just being a reference to that method is more than just pointer. M as an object, dot, its member. And so it is in fact m.Invoke that happens behind the scenes when we have m followed by parentheses.

This builds. There you go. And it doesn't really change the behavior at all. And so now, we have an opportunity to actually look at something asynchronous. So, with the invoke method,we have some other helper methods as well. And this follows the begin/end pattern for asynchronous processing. So, we have the option to do a BeginInvoke followed by an EndInvoke. And so we'll add the parameters in just a moment, but notice the pattern here. So we have a delegate.

So, an object describing how to call a method, and one of the methods is BeginInvoke, and one of the methods is EndInvoke. And so tying those two together allows us to call a method, BeginInvoke, that will cause the method being referenced to be run asynchronously. So a background thread will take care of it, and we will continue running our code on the main calling thread. And then we will eventually reach EndInvoke, and we'll be able to ask for the result of what happened to the method that they called through this delegate.

Let's see how this works. So first of all, BeginInvoke requires some parameters. First, it wants a callback. So it wants to be able to tell us that things have finished. This is optional, and it's asking, hey, is there anything you want to pass to the method when we call it, and I'll say null.Leave that alone for now. We'll come back to this in just a moment. Now, BeginInvoke will try and call that method, but in some cases, it'll have some work to do, so it will go around on a different thread. And so for us to know that it is finished, it will return something to us, an IasynchResult object.

So IAsyncResult. This object being returned to us is what we will use to determine if this method has completed and to get a result from it should we need one. And so at this point, one we've called BeginInvoke, another thread will take care of executing the method that m is referencing, which is DoWork, and our code will continue to run here. And once we're ready, we can go back to m, the delegate, and say we would like to finish that method that we called begin on earlier, and we'd like to know, is it finished? Is there a result we can have? And in some cases, maybe there might be an exception.

And this is the point in time, this is the statement at which we will actually get all of this reeled back to us. Now, the method returns a void, so there's no return type on the method, but we still need to confirm that it is finished. And at this point, we can wait for it to finish, if that's necessary. Notice the parameter here is the async result from before. And this is how these things are tied together. So, the delegate starts a call to the method it's referring to and gives us the async result to hold onto. And then when we say EndInvoke, well, which one do you want to end? You might have one this more than once.

So we would like to end this one, the async result that we retrieved. And so, the BeginInvoke/EndInvoke is paired. They're connected to each other through that async result.And this is practically the same as the synchronous call. We could do things in between to in the beginning invoke EndInvoke. Our method can continue to run other statements on its original thread, and the method DoWork will happen on a different thread. So let's just confirm that this still works.

In the output window, we should have Hello World. Okay. But what has really changed? Well, the thread we are executing our code on. So look at this. We can ask System.Threading.Thread.CurrentThread for its ID. It's called a ManagedThreadId. This of course is .NET. We'll do that as ToString. And so really, all we're asking is to write out onto the console the thread ID of the currently executing thread, and then we'll call our method.

We'll EndInvoke, and we'll finish our method. So the first ID will be that of the method running our unit test. And I'll copy and paste this line. Copy, paste. So immediately after Hello World,we will know the thread ID of the method DoWork. Let's run this again. Output window, thread ID 19, thread ID 17. So we have not yet dealt with multithreading in the sense of thread objects, but already, something has happened in the background.

We were able to identify that our code is actually running on thread number 17 and having noticed that, you now know that at least two threads are involved in executing our code. So, this is the delegate which already has a BeginInvoke/EndInvoke mechanism, and so any delegate can provide you this capability. So, if you're willing to work hard enough, this would be a way to do an asynchronous call for any method that you have a delegate for. The Begin/End pattern for asynchronous programming is relatively simple to use, but it is a patternwith several steps that we have to follow quite carefully.

So as you just saw, we can call BeginInvoke and then subsequently EndInvoke to finish running our method. However, we've still done that on the same test method, and so we didn't really get much benefit from this, except for perhaps having an opportunity to run a few statements before we get to EndInvoke. If we truly want to do this in an asynchronous waywhere we don't know when it finishes but it will notify us, which is different from what we're doing currently. EndInvoke says, we will stop here and wait if it's not done yet.

If we'd rather want the method to run asynchronously and call us back, then we have to do a little bit more work. And so, the example changes a little bit. We can have a method that saysplease call this method when things are finished. And so, the callback, as I have it named down here, is a method that should be called when the asynchronous method actually completes. So the way this works is we would provide this method as a parameter here.

But that parameter requires a delegate, an async callback. So an async callback is a built-in delegate that's already available, and we can call any method that returns void and accepts an IAsyncResult object. And we have one of those, TheCallback. So we'll place this delegate variable here as the first parameter, and so callback variable is its callback delegate object here, which refers to the callback method.

Little bit of indirection for you there. So BeginInvoke on method m, which is the original DoWork, all the way at the top of the screen, and once BeginInvoke kicks off the work for that method, it will happen on a different thread. And at that point, we're essentially done on this originating thread. When the asynchronous method call completes, they will notify us by calling the method provided through this variable, the method called, TheCallback. And so the end operation does not need to happen up here in the original method.

We move this down here. So, we begin, and then we can actually forget about it, and we will be notified if and when this method completes. So down here, what is m? Well, we don't actually have that information yet. So, knowing which method is calling the callback can be a little bit tricky, but there's an easy way to cheat. Notice this parameter over here, null. We can provide some additional object state to the callback method when it's invoked on completion.

And so, I'm going to sneak the variable m in there. What is m? The DoWork delegate. And so I can ask the async result for state, the async state. And I know what I passed in there was the DoWorkDelegate. So, m over here is being passed to us as part of the IAsyncResult object,and the IAsyncResult object conveniently has this async state that is whatever we pass in over there.

So you don't have to have a global variable. You can pass it in, it'll BeginInvoke, and it becomes available as AsyncState. So, on m, which is of type DoWorkDelegate, we can call EndInvoke, and we already have the AsyncResult up here, which is in fact the thing that will correlate to the two calls. Now, at this point, we don't get the value back from DoWork, but it would have done its work already. And you may find that if an exception was thrown, you may need to do a try/catch here to make sure that you catch that exception. Now, I did warn you before, we're in a test framework that does not know we did this.

So, when I run this demo, watch what happens. Test passes, and we get Hello World. That is a matter of luck. Because watch what's happening. Demo01 is our test method. We begin running this method asynchronously, m.Begin. It will run the DoWork method, and on completion, the callback will be called, this one down here. And it is surely a matter of luck that this method did not exit and cause the testing framework to shut down before this task was completed.

So here, we do have to wait, and four seconds is probably enough, but this is a terrible way to do it. Let's just run our test, make sure it works. It's going to count up to four seconds waiting for that sleep, and so we have a fair bit of confidence that the Debug.WriteLine will actually happen and complete before the four seconds is up. This, of course, is a terribly expensive way in terms of time to know whether this thing finished. So we would need to do a little bit more work to make sure our application or our test doesn't forget to wait for this to complete.

And in this case, the test framework is unaware of the asynchronous or the background thread doing this work for us, and so, yes, if this was downloading something or talking to a database, it might well get interrupted and killed as the process exits. So at this point, it would be better to wait for a signal and coordinate between the callback method and the end of this test method. I'll show you how to do that later. So, one way

to fix our problem of the unit test framework needing to wait for our asynchronous method to complete is to do signaling.

So this is one of several options, but in this case, I'll use the very simple WaitHandle type to actually signal that the callback has happened. So, on the one hand, we have a process that needs to pause and just wait for a moment. We don't want to wait for several seconds, not knowing when the action actually completes. We would like to be notified. And so, WaitHandles is one way to do signaling across threads in a thread-safe manner. So we can get a flag and say, look for this flag. If you see it, you can go ahead. If you don't see it, just wait there.

So in our code where we have System.Threading.Thread.Sleep, let's not wait that arbitrary amount of time. Let's rather wait for a signal. Remember, AsyncResult is the object that lets us know about the call that has happened. So m was the delegate on which we call BeginInvoke.And it said if you wanted to know more of if you wanted to call EndInvoke or anything like that, talk to AsyncResult. It knows what's going on. And AsyncResult actually has an AsyncWaitHandle that is used to wait for an asynchronous operation to complete, and that's exactly what we want to do.

It has this convenient method called Wait, and that's all we'll have to do. Now, we already know the callback method will be called when this method completes, and then we will do the right thing by looking at the result, dealing with exceptions, and so forth. But any caller who has access to the AsyncResult can ask to simply block itself, wait right there at that moment until the call actually completes. So instead of polling, inside of having the callback or instead of having a sleep, we can say we will wait here.

Our thread will now block. Now, counterintuitive, of course. We are in the unit test framework, and we want the unit test to just wait until the actual call completes. If this was production code, if this was not a unit test framework, this may or may not be necessary, because you may or may not have other work to continue with. Let's run our test, look at our output, and as expected, we still have Hello World, and the worker method still worked on a background thread, not the main thread that we started on.

And so, all the while, this has been green outcomes in the test. Of course I mentioned, the test framework may not be aware of what's happening in the background. So, the green dot doesn't tell you things are correct. It just means the framework didn't see any things that it's looking out for, like exceptions and asserts. And so there we have a very quick tour of how to use delegates with the BeginInvoke/EndInvoke pair of methods to do pretty much any method call asynchronously. If you have a method, you can create a delegate, use that delegate for BeginInvoke, EndInvoke, and you can call that method asynchronously.