Alright, in this lecture you're going to write your first unit test. So here we have this project called test ninja that I'm going to use throughout this course. I gave you the link to this project in the last lecture, but if you missed that, you can see it attached to this lecture.

So here we have this folder, fundamentals, let's take a look at this reservation class, so imagine you're building an application for reserving a table at a restaurant. A reservation class has currently one property which specifies the user who made this reservation. Now in a real world application, this class can have many other properties, but we don't want to get distracted with too much detail. This is the method we want to test, can be cancelled by a given user.

So here I have implemented a simple business rule. If the user is admin, or the user who made this reservation, they can cancel this reservation, otherwise they can't cancel this reservation, okay? And if you scroll down, we can see our user class, very simple, with one property. IsAdmin.

So let's go ahead and write the first unit test, for this method. Here, in the solution explorer. I'm going to add a new project to the solution. So add, new project. Under installed, visual C# test, you should see this template, Unit Test Project. If you don't see it here, chances are you have not installed visual studio property. So go and download the latest version of Visual Studio, currently I'm using Visual Studio 2017 Community Edition. So here we select Unit Test Project, and we give it a name.

So the name of our main project is test ninja, and I'm going to call this project testninja.unit tests. This is the convention we follow to organize our test project. We want to separate the unit test and integration tests, because unit tests run pretty quickly. Whereas integration tests take more time. We want to separate them, so we can run unit tests more frequently and integration tests before committing our code to a repository.

Okay? Let's go ahead, so Visual Studio creates this new project. And you can see here we have a file, unit test 1, it's a simple c#class that has the TestClass attribute, and it has one method called TestMethod1 one with this attribute, test method. These two attributes you see here, belong to MSTestFramework.

So the test runner we have in visual studio looks at all the classes decorated, with a test class attribute. Then, it looks at all the methods in this class decorated with a test method attribute, and it will run them. So, the first thing I want to do here is change the name of this class. So, press control R and then R again. Now you can rename this, I'm going to call this ReservationTests. Because we are testing the reservation class, so the convention we follow is the name of the class, plus tests. Enter.

Now, similarly I'm going to rename this method once again control R and R. The method I want to test is CanBeCancelledBy. Now here we add an underline, you specify a scenario underlined again, and ExpectedBehavior here. What is this? Let's go back to our reservation class, when writing unit tests for this method, we want to test all the scenarios.

What scenarios do we have here? We have three scenarios.

One scenario is when the user is an admin.

The other scenario is when the user is the same person who made this reservation.

And the third scenario is when someone else tries to cancel this reservation. So here we have three scenarios, or three execution paths. So there are three ways to get out of this method. That's why we call this three execution paths, okay? So, back to our unit test class, the first scenario we want to test is User IsAdmin. Now what is the expected

behavior if the user's admin this method should return true. So ReturnsTrue. This is the convention we use to name and organize our test methods. Every test method has three parts. The first part specifies the name of the method on the test, the second part is the scenario of our testing, and the third part is the expected behavior. Now inside every test method, we have three parts, Arrange, Act, and Assert. This convention is called triple A, or Arrange, Act, and Assert.

The Arrange part is where we initialize our objects.

We've prepared the object we want to test, in this case we want to create an instance of the reservation class, so, we create a reservation object, new, reservation, now here the Reservation class is not recognized, so you need to press control and period. And in this context menu, select add reference to test ninja. Okay, so we have the reservation object, the Act part is where we act on this object, and that basically means we're

going to call a method, that's the method we're going to test. In this case reservation.CanBeCancelled

By. Now, I want to give you an argument, in this case we want to give it ta user, that is an admin user. So here I create a user object and set IsAdmin to true. So this is the act part. Now we want to get the

result and verify that it's correct. So, I'm going to store the result in a variable called result, and in the Assert part, you verify that result is correct. In this case, result should be true. So here I'm going to use one of the helper classes in MSTestFramework. That is called Assert,

this class has a bunch of static methods that we use to make an assertion. For example, if you want to compare two objects, we can use AreEqual, other methods here like IsFalse, is instance of type, is not

InstanceofType and so on. So here, you want to assert that this result is true.

So we call assert, .IsTrue and pass result. So this is how we write a unit test.

Now to run this test, we go on the top, from the test menu, go to Run, and select All Tests. This is what I don't like about Visual Studio, because here we don't have the ability to run only a particular test method, like this test method we wrote in this lecture. We can only run all the tests in this project. And this is why I like ReSharper, because with ReSharper I can run only this test, Or, I can run all the tests in this class, so it's more flexible. I want to show you that later in the section. So, let's go on the top, from the test menu, Run, All Tests.

And note the shortcut. Remember this so you can quickly run all your tests.

This should open this

Text Explorer window, if you don't see that, you can find it

on the top, under tests, Windows, Test Explorer. Okay? So we can

see, we have a passing test, we have a green mark, we can see the name of our test method, and the time it took to run this test. In a real world application you're going to have hundreds or thousands of unit tests. So when you run your tests this window is going to be populated with all the tests in your application. Now let me should you an example of a failing test.

So, I'm going to go back to the reservation class, and create a bug in our code. So here, I'm going to return false.

Let's say the developer who wrote this code made this mistake. Now as I told you before, with unit tests, we can catch bugs before deploying our application. So when we create a unit test class, and verify the expected behavior of a method, in this case you're verifying that this method is returning true, with this unit test, we can catch bugs earlier, and the software development life cycle. So, back to our test explorer window, I'm going to run, call the test again. This time, our test failed. So we can see the benefit of unit tests, in action. Now here's an exercise for you. As I told you before, this method has three execution paths, or three scenarios, I want you to use what you learned in this lecture, and write the other two unit tests for this method. In this next lecture, you're going to see my solution.