So, back in our Mocking folder, look at this product class.

This product class has a property called ListPrice and a method called GetPrice that takes a customer argument. So here we check if customer IsGold, we give them 30% discount. Otherwise, we return the ListPrice. And here is the definition of the customer class.

Very simple example, but we can use it to see what happens when you abuse mocks. So, let's go ahead and write a unit test for this method for the scenario where customer is a gold customer. So, back in our solution explorer, in the Mocking folder, let's add a new Class, Product

Tests. We add the TestFixture attribute, and write the Test

so, GetPrice, for GoldCustomer, should Apply30Percent Discount. Now, let's create a product object. New Product, and set it's ListPrice to 100

dollars. Okay? Now, let's act, so product GetPrice, we give it a customer object, and set IsGold to true.

Then, get the result, and finally, Assert That, result, is equal to, 70 dollars.

Let's run this test. Okay, it passed, beautiful. Now, back in our product class, let me show you what happens when you abuse mocks. So people who abuse mocks extract (?) them at interface from every class. So here, let's Extract an interface from the customer class, refactor an extract interface. We call it iCustomer with one property, okay? And then,

in GetPrice method, instead of Customer you pass an ICustomer object. Now, back to our unit test, the current unit test is working, because this customer object we're passing of course it implements the ICustomer interface, so this test is perfectly fine. However, people who use mocks everywhere, don't write a unit test like this.

Instead, they write a unit test like this. So I'm going to duplicate this, and change the name of the second test method, let's add two at the end, so we can compare them side by side. So, here we need to create a mock object, so var customer is a new Mock of ICustomer. And then we need to

program this mock, so customer.Set up. C goes to c.IsGold Returns true.

Okay. And then, instead of passing a new customer object we pass customer .Object. Compare these two tests side by side. Which one is simpler? Which one is cleaner? Which one is easier to understand? Of course the first one. And here, you are dealing with a very simple example. So, our unit test has two extra lines of code as this setup of a mock object, imagine in a real life scenario where you are dealing with a more complex code, you're going to have a few mock objects, if you're going to mock everything, and your tests start to get really boggled.

You'll have a test that takes the entire script from top to bottom.

Your tests should be ideally within 3-5 lines of code. This is an ideal unit test. But of course, I don't in real life scenarios, it's not always possible to write such short unit tests, but if you follow the best practices that I've taught you in this course, you can achieve small, maintainable and reliable tests. So, 3-5 lines is ideal, even 5-10 lines is okay, as long as you're not doing this kind of mocking, this is really, really bad and I used to do this before, and I've realized the hard way that this is a recipe for disaster. So one more time, use mocks, but removing external resources from your unit tests.