Alright, so here in our unit testing project, I'm going to add a new class called StackTests. And just like always, apply the TestFixture attribute.

Alright, we're ready to write our first test, now here we're dealing with an interesting situation. If you look at the implementation of the stack, you can see that we have the private read only feed of type list, so this is the internal or implementation detail of the stack, and this public members stack we have represent the public API or the public contract, for the public interface of this class.

In our tests, we should only use these public members. So we are not so we're not going to work with this list object. Because chances are in the future, you may decide to use an array instead of a list. Or some other kind of data structure.

So, back to our test class. So first I'm going to write a couple of tests for the push method. So, Push, now what she we test here? Our

Push method takes an object. Here we need to test 2 scenarios.

One is where the object is null, in that case the push method should throw an argue and null exception.

The other scenario we need to test is where we give it a valid object

and we ensure that the push method has this object for a stack. So the first scenario is ArgumentIsNull, it should Throw ArgumentNull Exception.

So, I'm going to create a stack, new Stack, note that we have another stack class defined in system.collection name space, but here we should use the stack that is defined in testninja.fundamentals. So Stack of string.

Now, I'm going to Assert That here I'll pass a lambda expression, when we call stack.Push with null argument this method should Throw ArgumentNullException. Let's verify this Okay, here's our new test, and it successfully passed. Beautiful.

Now the second test. Push Valid Argument, it should AddTheObject

ToTheStack. So first I'm going to create this stack object, new Stack of strings, now, we call the push method, give it a valid argument, like a as

a string, and finally we assert That stack.Count is equal to one.

So once again, we don't have access to this private

field in this class, so we're not going to go on this list to verify

that is contains this object. You have to use the public API, or public members of the stack to make sure it's behaving properly, in this case the simplest way to verify this, is to use the Count property. Now it is possible that the Count property is not implemented properly.

For example look currently this is implemented as an expression body property, so Count goes to simply return the list of Count. But what if this property was implemented like this, then our test would pass, but we actually have a bug in the implementation, stack.Count.

So, I'm going to write a separate test for this property, because it has some logic. So, Test method, Count I want to test that when this stack is empty, Count is 0. Then when you add an object to the stack, Count is

1. This way I can ensure the count property is implemented properly. So, back here EmptyStack is the scenario, it should ReturnZero.

So, let's create a Stack object, new set of strings, we don't add anything here, we simply Assert that stack.Count is equal to 0. Let's run this Test. Okay, our test failed. We expected 0, however we got 1.

Because here I replaced list.count with 1. So let's revert this back.

We return list.count, now back in our test class I'm going to run this test one more time. Okay, so our count property is working, excellent. And we have two tests for the Push method. Now let's write a few tests for the Pop method.

So, test method Pop, what are the scenarios we need to test? One is when we need to call this method on an empty stack. In that case we should get an invalid operation exception. So Pop EmptyStack should ThrowInvalidOperation Exception. That's one scenario.

Now the second scenario.

So Pop, in this case we want to test this method when our stack has

a few objects, so Stack WithAFewObjects, what is the expected behavior of this method? It should return the object on top of the stack and remove it

from the stack at the same time.

So here we need to write two separate tests.

One to ensure that this method returns the object on the top,

and the other to ensure that this method removes that object from the

stack.

So, Returns ObjectOnTheTop. and I'm going to duplicate this here the second expected behavior is RemoveThisObjectFromTopOf the stack. Okay? So let's implement this test method one by one. I'm going to create an empty stack new Stack of strings. And a Assert That, when we call stack.Pop we get an InvalidOperationException.

So ThrowsInvalid OperationException. Let's run this test, beautiful, now earlier I told you about writing trust worthy tests, so here, even though this test is passing, I want to make sure that it's testing the right thing, I want to make sure this is a test I can rely upon in the future.

So, I'm going to go back in the stack class, here is our Pop method.

This is the line that is responsible for making that test pass. So if I comment this out, and go back here, run this test, it should fail. So, let's run it.

Okay, this time it failed, because we expected an InvalidOperationException, however we got ArgumentOutOfRangeException. So this verifies that my test was testing the right thing. But why did we get ArgumentOutOf RangeException? Let's have a quick look.

The reason was for an empty stack, Count is 0, so we try to access an item and index minus one in this list, and that's why we got ArgumentOutOfRange Exception. So, I'm going to bring these two lines back, okay now let's move onto the next test. What if you're dealing with a stack that has a few objects? You want to make sure that this Pop method returns the object on top of the stack.

So, in the arrange part, I'm going to create a stack with three objects. Stack, new Stack, of string. I'm going to Push three objects here, 1, 2, and 3. So this is our Arrange part. As you can see the Arrange part is not always one line of code, it can do multiple lines. So, you bring

this object into the right state and its ready for testing this particular scenario. Okay? Now, you Act. So we call stack to Pop. You get the result,

and finally we need to Assert.

We want to Assert.That result is EqualTo c. is EqualTo c. Let's run this test. Okay, the test is passing, once again, I want to make sure that there is no bug in this test. So, Back in our Stack class, this is the line that is responsible for making that test pass. So I'm going to change this

temporarily and return ist of 0. I'm going to return the first element in the list.

So this should break my test right? Back to our test,

let's run this one more time, Okay, this time it failed because we expected

c, but we got a. So when you write your tests after your production code, use this technique to (?) your tests in the right thing (?). Now back in the stack class, let's revert this back.

And the last test for the Pop method, we want to make sure

that this method actually removes this object on top of the stack. So I'm going to copy this code, into this new test I'm going to change the Assertion, so here we don't really care about the result, just Pop an object and then Assert that stack.Count Is EqualTo 2.

Now, let's run this test.

Okay, the test passed, beautiful, and by the way these two failed tests they are not actually failed anymore, these are the results of previous test execution. So if I run the tests inside stack tests again. All of them are going to pass. There you go. Now finally, we need to test

the Peek method.

So, Peek, what are the scenarios here? The first scenario is when we call this method on an empty stack. Doesn't make sense. There is nothing to Peek. So this method should throw an InvalidOperationException. So, empty stack, ThrowInvalidOperationException.

Alright, next test. Peek, Stack, WithObjects. Here we want to make sure that this method returns the object on top of the stack. So ReturnObjectOnTopOf TheStack. Also, I want to write one more test here, to make sure that unlike the Pop method, thismethod does not remove that object from the step. So, another test, Peek, StackWithObjects. DoesNotRemoveTheObject OnTopOfTheStack. Okay?

Now, let's implement them one by one. So, if I create an empty stack, stack of strings, and Assert that when we call stack.Peek, you get an exception, so this throws InvalidOperation Exception. Let's run this test. It's passing, beautiful. Let's move on to the next test, now we need a stack with a few objects. So I'm going to borrow some code from this other test this here, and Act part we should call the Peek method, get the result, and finally Assert that result IsEqualTo c Let's run the test.

It's passing, excellent, and the last test. I'm going to copy all this code from here. into this other test, here we don't care about the result, we just want to check the count is not changed. So we assert that stack.Count is still three, is EqualTo 3. Let's run this test.

Beautiful. So we have a total of 9 tests for the stack class and they're all passing. And by the way, this is one way to test the stack class. Chances are the way you have written your test is different, and it doesn't really matter as long as you have tested various scenarios, and verify the stack class is behaving as you expect. So don't get hung up if the name of your methods or how you have implemented each test method is different than mine. Even if I tried to test this class tomorrow, chances are my code is going to look different from what you see in this lecture.