Introducing AutoFixture

Contents

[Introduction 2](#_Toc522270884)

[Benefits of AutoFixture 2](#_Toc522270885)

[Simplifying the Arrange Phase of Tests 3](#_Toc522270886)

[Anonymous Test Data 3](#_Toc522270887)

[Testing Framework Support 3](#_Toc522270888)

[Introducing the Fixture Class 4](#_Toc522270889)

[Getting Started in Visual Studio 4](#_Toc522270890)

[Summary 6](#_Toc522270891)

[Creating Anonymous Test Data 6](#_Toc522270892)

[Introduction 6](#_Toc522270893)

[Creating Anonymous Strings 6](#_Toc522270894)

[Creating Anonymous Numbers 7](#_Toc522270895)

[Creating Anonymous Dates and Times 8](#_Toc522270896)

[Creating Anonymous Value Sequences 9](#_Toc522270897)

[Creating Custom Objects 10](#_Toc522270898)

[Creating Complex Object Graphs 11](#_Toc522270899)

[Creating Objects with DataAnnotations 12](#_Toc522270900)

[Creating Sequences of Custom Objects 12](#_Toc522270901)

[Summary 13](#_Toc522270902)

[Customizing Object Creation 13](#_Toc522270903)

[Introduction 13](#_Toc522270904)

[Customizing Individual Fixture Instances 14](#_Toc522270905)

[Customizing Individual Object Creation 15](#_Toc522270906)

[AutoFixture Pipeline Overview 17](#_Toc522270907)

[Customizing the AutoFixture Pipeline 17](#_Toc522270908)

[Summary 19](#_Toc522270909)

[Writing Less Test Code and Improving Test Maintenance 19](#_Toc522270910)

[Introduction 19](#_Toc522270911)

[Refactoring with Manual Test Data Creation 20](#_Toc522270912)

[Creating Refactor-safe Test Code 21](#_Toc522270913)

[Parameterized xUnit.net Tests with Anonymous Data 21](#_Toc522270914)

[Auto Mocking with Moq 23](#_Toc522270915)

[Auto Mocking with Rhino Mocks 24](#_Toc522270916)

[Combining Auto Mocking and Auto Data in xUnit.net 25](#_Toc522270917)

[Summary 26](#_Toc522270918)

# Introduction

Hi. I'm Jason Roberts from Pluralsight. Welcome to this course on Better .NET Unit Tests with AutoFixture. In this course we're going to be looking at the open source AutoFixture library and we're going to learn how this library can improve our unit tests in a number of different ways. By the end of this course, we'll learn how to use AutoFixture to create anonymous test data. so, for example, this test where we're subtracting the value 1, we'll essentially be able to turn into something that looks more like this. So even though in this example it's a very simple test, we can start to see a number of benefits of AutoFixture.

## Benefits of AutoFixture

So what are some of the benefits that we can expect to gain by using AutoFixture in our unit tests?

* The first of these is the potential to **improve readability** of our unit tests because we're going to be **using AutoFixture to generate anonymous test data**, test data that we don't really care about, the readability of the test code can potentially be improved, so rather than the future reader of the test code having to worry about what the individual values mean in the context of the test, for example, subtracting the value 1, by using AutoFixture we can abstract away this idea of anonymous test data.
* Because we're getting AutoFixture to generate some test data for us, we can actually improve our **programming productivity**. This is because we don't have to spend as much time creating setup test data that we don't care about. We're getting AutoFixture to do this work for us so we have to write less code.
* Using AutoFixture can also help us **reduce the test maintenance overhead**. So for example, if we're creating a lot of manual test data and then we change the production code, this might mean that we then need to go and change all of the tests to reflect the change in production code. If we're using AutoFixture to automatically create test data then this can mean that we no longer have to spend as much time changing unit tests to change the test data creation that relates to the production changes that we've made.
* And finally, if you're practicing test-driven development, using AutoFixture **can help to reduce the TDD friction**, so this is related to the reduction in test maintenance because when we're practicing TDD, we're going to be evolving the design of the system with many small changes. So every time we write a new test and then write some new production code, we don't want all of our existing tests to break because of test data creation, and again, AutoFixture can help us in this regard because we don't have to manually go and change existing tests just to produce some test data.

## Simplifying the Arrange Phase of Tests

So essentially we can say that using AutoFixture allows us to reduce the amount of arrange code in our unit tests, but if you haven't heard of this idea of arrange code, it's part of the concept where we can split the logical phases of the test into **the arrange phase, the act phase, and the assert phase**.

* So in the arrange phase of a unit test, we're going to get things set up as we want them to be before we actually perform any action on the thing we're testing. So this includes creating the thing that we want to test and also setting up any test data that we need.
* The act phase is where we actually perform some action on the thing we want to test. So, for example, calling the subtract method on a calculator. So this is where we're actually exercising the thing that we want to test. T
* he final logical phase in the unit test assert phase. So now we've performed some action on the thing we want to test, we want to test that the result is correct and we do this by making assertions based on the outcome of what happened in the act phase. It's this logical arrange phase that AutoFixture is designed to help us reduce, specifically helping us to arrange our initial test data or the state of the system. It does this by allowing us to create anonymous test data.

## Anonymous Test Data

So I've used the term anonymous test data a couple of times already in this introduction, but what exactly do we mean when we say anonymous test data? Well, ***anonymous test data can be thought of as data that is required to be present for the test to be able to execute, but where the value itself is unimportant.*** If we take a look at this code from earlier, we can see that in the first test method we're subtracting the value 1, so we can call this known test data, because we know that the value is 1, but if we look at this test code, we're subtracting a number and then just checking that the value is less than 0, because we assume that a calculator starts at the starting value of 0. So here, as long as we subtract a positive integer, it doesn't actually matter what this test data is. In the second example using AutoFixture, we can see that we don't actually specify the actual literal value of the number that we want to subtract. This represents the fact that for this test we don't actually care what the number is, only that we subtract some value. So in this sense, because we don't care about the value, we can call this anonymous test data and it's this anonymous test data that AutoFixture helps us create.

## Testing Framework Support

***AutoFixture is not dependent on any one test framework,*** so the basic AutoFixture NuGet package can be installed into your test project, regardless of which test framework you're using. So, for example, AutoFixture can be used with the NUnit testing framework with MSTest, with the xUnit.NET testing framework, and even newer testing frameworks such as the Fixie testing framework. There are some additional NuGet packages that provide additional AutoFixture support for some of these testing frameworks and we'll look at those later in this course.

## Introducing the Fixture Class

One of the central classes in AutoFixture is the fixture class. We can create an instance of this class and then use it to create our anonymous test data. ***The fixture class provides the create method and calling this method allows us to generate a piece of anonymous test data.*** So, for example, we can use this create method to create an instance of a char, to create anonymous string test data, to create anonymous test data for numbers, so for example, ints, decimals or floats. We can create anonymous guids, instances of dateTimes, and we can even use the create method to create instances of our own custom types and if this custom type contains properties, AutoFixture will also populate the custom types properties with anonymous test data. The fixture class also contains a number of other methods that allow us to take control of how we build our custom types and even create anonymous collections of objects. We'll look at the fixture class in depth in the next module and we'll see how we can create all these types of anonymous test data.

## Getting Started in Visual Studio

So let's see now how we can actually get started with AutoFixture in Visual Studio. In this demo we're going to start off by creating a traditional test where we're going to just use some literal values for the test data. We're then going to create a more descriptive test to convey the notion that the actual test data is anonymous. We'll then see how we can use AutoFixture to create our test data. To do this we're going to install AutoFixture from NuGet and then we're going to see how we can use the fixture's create method. So let's head over to Visual Studio.

So our demo code project here represents the production code that we want to test and in here we've got this simple calculator class, and here we've just got a subtract method, which will subtract an int, and we have a value property here that represents the current value of the calculator. So the first thing we're going to do is create a new project in our solution to hold our tests, so I'm just going to right click, add a new project, and we're creating a class library project here, and I'm going to call this (typing) democode.tests. Let's just create this project.

For this demo I'm going to be using the xUnit.net testing framework so the first thing we need to do is head over to NuGet here and we're going to install the xUnit.Net main package here. So let's install that, and once this main package is installed, we're also going to install the Visual Studio Runners package, so this will just enable the test explorer in Visual Studio to discover and execute the xUnit.net tests. So let's install this package, and then just close.

So let's start by writing a traditional test. So the first thing I'm going to do is just rename this class, calculator tests, and just use Resharper here, just to clean up these unused using directives. So let's begin here with the fact attributes and if you haven't used xUnit.net before, this just marks a method as being a test method, and then we're just going to create a method (typing), and I'm just going to call this test method traditional. So we're going to have three phases in this test method. (typing) The arrange phase, the act phase, and the assert phase. So in our arrange phase here, we need to create an instance of the calculator that we want to test. So here I'm using the acronym SUT, which stands for System Under Test, though this just helps us to recognize the thing that we're testing in any given test. So let's create a new calculator object. And we just need to go ahead and add a reference to our actual demo code project from our test project. We now need something in our act phase here (typing), so here we're going to call the subtract method of our calculator and we're just going to subtract the number 1. So again we're using some known test data here, the literal value 1. Finally comes the assert phase. So here I'm just going to assert that the current value of the calculator now is less than 0, because we've subtracted 1. So if we build this, and then go and run our test in test explorer here, this test should pass.

So this is okay, but the future reader of this test might be left wondering what literal value 1 here, actually means. Does the value 1 here have any significance to this test, or is it just some arbitrary value? So let's try and write a test now that expresses this idea that this literal value 1 is actually an arbitrary value and we don't actually care what it is. So I'm just going to paste in another test here (typing), and I've just called this test manual anonymous data, and we can see here we've got the arrange, act, and assert phases, as before, and as before, in the arrange phase, we're creating a new instance of our system under test, our calculator, and this time we're creating a variable with the name anonymous number that attempts to represent the concept that we don't actually care what this number is, and here I've just used a completely different number, 394, and then in the act phase, we're going to use this anonymous number instead of a literal value here.

Let's just build this and we can run it, and it passes. So this expresses the idea that the value being subtracted may not actually be important, but we still have a literal value, 394, which may still make the reader wonder about exactly what this value means. So let's now write a test using AutoFixture to generate an anonymous value for us. So the first thing we need to do is install the AutoFixture NuGet package into our tests project. (typing) So I'm just going to search for AutoFixture and we can go ahead and install the main AutoFixture package here. So that's AutoFixture installed. Let's head back to our test class. So let's create a new test here that's going to use AutoFixture. (typing) So again, start with the fact attribute, (typing) and I'm just going to call this new test AutoFixture anonymous data. So again, we're going to start with our arrange phase (typing).

We're going to create our instance of our system under test, but we're also going to create an instance of AutoFixture's fixture class. (typing) So here I'm just going to use Resharper to add a using statement (typing), and we can see at the top here, we've now got a using statement for AutoFixture, and if we head back to our new test here, we can now write our act phase. So again, we're going to call the subtract method, but this time we're going to use our fixture object to create an anonymous number for us. So to do this, we can use our fixture instance and call it create method. So when we call the create method we can provide a generic type parameter here (typing). So in this case we want to create an anonymous int to subtract from the calculator, so for the anonymous tight parameter here, I'm just going to specify int. So what this will do is it will use the fixture object to create an instance of some anonymous integer data. This integer data then will be passed to the subtract method of our calculator and subtracted from the value. Just as before, we can write our assert phase, and if we build this, our test shows up and we can run this, and it passes. So notice in this AutoFixture based test here, we no longer have an literal value for the amount being subtracted, because we don't actually care what it is. So this is a very simple example just to show the basic use of AutoFixture. In the next module we'll look at using the fixture class in a lot more detail.

## Summary

So that brings us to the end of this introductory module on AutoFixture. In this module we learned that AutoFixture helps us in the arrange phase of our unit tests. It does this by allowing us to create and use anonymous test data, so this is test data where we don't actually care what the value is, but we still need it to perform the test. We learned about AutoFixture's fixture class and how we can use it to create anonymous test data of different types. We learned that AutoFixture is not reliant on any specific test framework and we can use it regardless of the underlying test framework that we're using to write and execute our tests. Finally, we saw a demo of how we can install the open source AutoFixture NuGet package and then create an instance of the fixture class and use it to create some anonymous integer data. Join me in the next module when we'll be learning more about the fixture class and how we can use it to create various types of anonymous test data including simple strings, strings with seed data, numbers, custom objects, and even collections of objects.

# Creating Anonymous Test Data

## Introduction

Hi. Welcome back. In this module we're going to look in a bit more detail about how we create anonymous test data of various types. So in this module we're going to start off by seeing how we can create anonymous strings and numbers. We'll also learn how we can use AutoFixture to create anonymous DateTime and TimeSpan instances. We'll learn that in addition to creating a single instance of a piece of anonymous data. We can also create collections or sequences of anonymous data and we'll learn that as well as creating instances of simple types. AutoFixture will also populate complete object graphs with anonymous test data. In this module we'll also learn that if we're using AutoFixture to create an instance of an object, which has data annotations applied, in what circumstances AutoFixture will respect these data annotations.

## Creating Anonymous Strings

So in the first section of this module, we're going to see how we can create anonymous test data for primitive types such as strings. So in the first demo, we're going to see how we can create anonymous strings. So we'll start off by creating a simple anonymous string and then see how we can add additional preceding text to the anonymous string by using a seed value, and we'll also learn how we can create anonymous char data, so let's head over to Visual Studio.

So here's our solution and in this demo code project here we've got our production code and throughout this module we'll be writing tests to create anonymous data for these classes. We've also got our test project here and in this test project we've got AutoFixture installed and we're going to be using xUnit.net to write our tests. So let's have a look at this name joiner class (typing) and we can see here it's got a simple method join, which takes a first name and a last name, and it just joins the first name and the last name and adds a space between the two of them. So let's go and write a test. So I'm just going to create a class here and just call this strings and just clean up these usings, (typing) and now we've got a test class here we can write our first xUnit.net test. So I'm going to start with the fact attribute (typing), and let's first start here in this test by seeing how we can use AutoFixture to create anonymous strings without any kind of seed value. So let's start in the arrange phase here and the first thing we're going to do is create our fixture object and again, we've added the using statement for AutoFixture here, and we also need to create our system under test, which in our case here is the name joiner class. So we want to create a piece of anonymous string data for the first name and for the last name that we're going to pass to the name joiner's join method, here. So let's start off (typing) by using our fixture instance, and again we're going to call the create method, and we're going to tell AutoFixture that we want to create some string anonymous data, so we do this by providing string as the generic type parameter and we can also do the same thing for the last name. (typing) So now we have two pieces of anonymous string data, we can go ahead and write our act phase. So our act phase here is simply calling our name joiner's join method, passing the first name anonymous data, and the last name anonymous data and then we're going to get the results back of the join method. So now we have our results, we can write our assert phase, so what we want to do here is we want to assert that the expected value is going to be (typing) the combination of our first name and last name anonymous data with a space in between them and the actual value here is the result from our join method. So let's build this and we can run this test and the test passes. Sometimes when using anonymous string data and the test fails, it can be a bit harder to understand exactly what the assert failure is, so let's simulate this by changing the name joiner class here, and I'm just going to introduce a typo here, where we're joining the last name to the last name twice instead of using the first name. If we run our test now it should fail, which it does, and if we have a look at our assert failure message here, you can see that we've got this anonymous data, so this is just a random generation of strings, but we can see that it's a bit harder to understand exactly what the expected and actual results should be and that they refer to the last name or first name. To help with this, AutoFixture allows us to create seeded anonymous strings. So let's head back to our test and what we're going to do is we're going to modify our fixture create method calls here and we're going to provide an extra string here, which is called the seed string, and the value we specify here will be prefixed to the actual auto generated string. So here I'm just going to say, first, and here I'm going to say last. If we run the test again now, we'll get more of a helpful failure message. So we can see now in our expected message here, we're expecting first, but we actually got last, and notice here that the rest of the string is still an auto generated anonymous value, has just been prefixed with the values first and last. In addition to strings we can also create anonymous char data, so I'm just going to paste in a new test here and I'm just going to put a break point here and just build, and if we debug this test, we can see this time when we call fixture. create, we're passing a char as the generic type parameter, and if we hover over this, we can see that the anonymous character that we've got here is a colon, and if we just run the test again in debug mode, we can see next time we get a different char; in this case it's an equals character. (typing) So that's how to generate anonymous string test data and optionally use a seed value for the strings and also how to create anonymous char test data.

## Creating Anonymous Numbers

So let's see now how we can use AutoFixture to create some anonymous numeric test data. So let's head over to Visual Studio, and here we can see we've got a couple of classes, this int calculator class, which has a subtract and an add method, and when we call the subtract to add method, it's simply going to modify the value property here of the calculator itself, and we have the same kind of thing here, but this time using decimals. So let's go ahead and create a test class. (typing) Just going to call this numbers (typing). So let's start off testing the add method of the int calculator. So I'm just going to start (typing) with a test called int, and we're going to start off here (typing), with our arrange phase and here we're creating a new instance of the fixture, creating a new instance of the thing we're testing, the int calculator, and then using our fixture instance here to create an anonymous int value. So now we can go ahead and use our anonymous int value in our act phase. So here we're just calling the int calculator's add method and passing it the anonymous integer value. Because the int calculator's initial value is 0, if we add a value to it, the new value of the calculator should be equal to the value that we've added, so in our assert phase here, we can simply write that the expected value of the calculator is whatever anonymous integer we've added, and we're checking that to whatever number we've added to the calculator is now the calculator's new value in this value property here. So let's go ahead and build this, and we can head over here and run this test and the test should pass, which it does. If we need to create different types of numeric value, we can do that. So I'm just going to paste in this test here and this time we're using the decimal calculator and notice that our fixture creates method here is using decimal as the generic type parameter, and as before, we're adding this number and checking the value as the same. So let's build this and run our test, and our test passes. So we can use AutoFixture to create anonymous numeric data for all of the basic numeric types in the .NET framework. So I'm just going to paste this test in here and we're just creating a fixture as before and we're creating a whole series of different values. I'm just going to put a break point here and build, and we'll debug this test. (typing) I just remove, reduce the size of this, and if I hover over these different numeric types, we can see that for byte value, we've got an anonymous value of 122. We've got an anonymous double value here, and anonymous short value here, a long value, a short byte, a float, an unsigned short, an unsigned integer, and an unsigned long value. So we can use AutoFixture for all of these numeric types.

## Creating Anonymous Dates and Times

So we can also use AutoFixture to create anonymous dates and times for us. So here we have this log message create a class and it contains a single method called create, which constructs a log message for us, from a message that we pass in as a string and also a DateTime for the message. To construct the new log message here, we're going to take just the year portion from the DateTime passed in, and also set its message, and if we have a quick look at this log message class, we can see that it just contains an int for the year and a message string property. So let's see how we can use AutoFixture to create an anonymous DateTime. I'm just going to add a new test class and call that date and times (typing) and we'll start off with a test method called DateTimes. So let's start off with our arrange phase here (typing), and again we're creating an instance of a fixture and we're using the fixture's create method and this time using DateTime as the generic parameter. We can now write our act phase. So we want to test that we get the correct log message produced by the log message creator so we're going to call the log message creator's static method (typing), create, and we need to pass the message string and the DateTime. So in this test we're just going to be checking that the correct year is set in the returned log message, so we don't really care what message we pass in, so this is a good candidate for some anonymous string data. So here I'm just going to use our fixture instance, (Typing) call its create method, (typing) and ask for an anonymous string. (typing) The second parameter to our log message creator's create method is a DateTime, so we're just going to pass in the DateTime that we've created here in our arrange phase, so let's add that. (typing) And now we can write our assert phase. So here we're asserting that the log message that's created for us has the correct year set, so again we're accessing the log message's year property here, and we're just going to check that it's set to the same year as we've got generated for us in our anonymous DateTime, so let's build this (typing), and run our test, and our test passes. As well as anonymous DateTimes, we can also generate anonymous TimeSpans so I'm just going to paste in a new test here and put a break point and we'll just build and debug this test, and if I hover over this TimeSpan value here, we can see that AutoFixture has created an anonymous TimeSpan for us because we're specifying TimeSpan as the generic type parameter here.

## Creating Anonymous Value Sequences

So now we know how to create anonymous values for simple things such as strings, numbers and DateTimes, let's see how we can use AutoFixture to generate sequences of values and also how we can use it to generate complex types. So in this demo we're going to see how we can use AutoFixture to create sequences of anonymous values.

So to do this, we're going to learn about the fixture class' ***CreateMany*** method and we're going to see how we can use this to generate sequences of strings and integers, for example. We'll then see how we can use the convenience ***AddMany*** to method to generate anonymous sequences and add them to an existing collection. Finally we'll see how we can use a custom creation function for creating the elements in the sequence. So let's head to Visual Studio and I've already created this collection's test class in the test project, so let's start off by seeing how we can use AutoFixture to create a sequence of anonymous strings. (typing) So just start off with this test method here. We'll start off by creating a new fixture instance and in this example we're going to see how we see how we can create a sequence, an IEnumerable of type string. So to do this we're going to use (typing) the fixture instance as before, but instead of using the create method here, instead we're going to use the CreateMany method. (typing) To do this we can specify generic type, so in our example we want to create a sequence of anonymous strings. And just for demonstration purposes here, I'm just going to add a foreach loop that's going to output each message to the debug window. So let's build this (typing), and we'll run this test in debug mode (typing, and we'll go and have a look at the debug output. So we can see here that we've got three anonymous strings that have been generated by the CreateMany method. By default, AutoFixture will generate a sequence of three values when we call the CreateMany method. If we need to, we can tell AutoFixture to generate a specific number of anonymous values in the sequence. (typing) So we're just going to paste in another test method here, and as before we're creating a new fixture instance, and we're using the CreateMany method, this time to generate a sequence of int values, but this time we've used an overload of the CreateMany method, which allows us to specify how many elements in the output sequence that we want to generate. So here we're saying generate a sequence of six anonymous integers and again we've just got a debug right line for each element here. Let's build this and debug the test. (typing) And we'll take a look at the output, and we can see this time we've got six anonymous integers generated as opposed to the default three. If the thing we're testing has a thing that we want to populate as part of our arrange phase, we can use another method from the fixture class, so I'm just going to paste in a new test method here and here we're testing the debug message buffer class as our system under test and if we just have a look at this class, we can see that we get this message property, which is a list of strings and we also have this write messages, and again for demo purposes we'll just output each message to the debug window. So back to our test here. Rather than using the CreateMany method as before, this time we're going to use the fixtures AddManyTo method to generate a sequence of anonymous values and automatically add them to an existing collection. So here we're telling AutoFixture to generate 10 strings, and add them to the existing collection, in this case the debug message buffer's messages list of strings, and then we're just outputting all of the messages that have been added to the debug window. So let's go ahead and run this test again in debug mode so we can see the outputs, and we'll have a look at the output window here (typing), and we can see here that AutoFixture has added 10 anonymous strings to the messages property of our system under test. When we're using the AddManyTo method, we can optionally tell AutoFixture how to generate the items in the sequence. (typing) So I'm just going to paste a new test in here. Again, we're using the debug message buffer, but we're using an overload of the AddToMany method here (typing), which allows us to specify a creator function that will return a string for each element in the generated output sequence. So at the minute here I've just got a lambda expression, which will return the single value high as part of this function and if we build this and debug the test, and check out the test output, we can see now that we've got three high strings that have been generated, and again, three is the default number generated by AutoFixture for a sequence. So if we want the same value for each item in the sequence, we can do this, but we probably want a different value for each element in the sequence. So I'm just going to uncomment this line, and here what we're doing, we're using an instance of the random class here and we're just generating a number between 1 and 10 for each element in the sequence, and then just converting this number to a string. So if we run our test again, in debug mode, and check the output, we can see this time that we've got three different random numbers generated from the AddManyTo method.

## Creating Custom Objects

So let's see now how we can use AutoFixture to create instances of our own custom types. So here we have this email message class and this class has a ToAddress message body, subject and isImportant properties, and we also have this constructor here, which takes the ToAddress message body and isImportant flag. We've also got here a few members of this class for demo classes. I've got this private string field called \_SomePrivateField, this public string field, SomePublicField, and also this private property, SomePrivateProperty. We've also got in our production code here, this email message buffer, and this email message buffer contains a list of emails and a method to send all of the emails, which is just going to output them to the debug window and we also have a method here, which will add an email to the list of emails in this class. So let's go and write a test for this email message buffer that checks that when we add a message using this add method, it actually does get added to the emails list here. So let's go and create a test class (typing), and I'm just going to call this test class custom object. We'll just add that. (typing) And let's go and add a first test method here to test to this email message buffer's add method, but in this first example, in this first test, we're going to create all the test data manually. So I'm just going to paste in this test, and if have a look at this test, we're creating the instance of our system under test, our email message buffer. We're creating the test email message that we're going to use to test the email message buffer's add method. So when we create an email message as part of the constructor, we need to specify a ToAddress, a message, is important or not. Once we've created a message we can also set its subject property as an optional thing. We're then calling our system under test's add method, so we're adding this email message to the email message buffer, and in our assert phase we're checking that once we've added a message that the count of messages in the email's property from our email message buffer is now 1. So let's build this (typing) and run its test, and the test passes. If we have a look at our manually implemented test here, this whole email message is in effect unimportant about exactly what we set these properties to. So it doesn't really matter what the ToAddress message body IsImportant and subject are because in this test we're just checking that when we add a message the count is correct. So this is a good candidate again to make this anonymous test data and by using AutoFixture to create our email message, we're also going to simplify the arrange phase and reduce the amount of code. (typing) So let's write a new test, this time using AutoFixture to create our email message test data. (typing) So we'll start with an empty test here, and we'll start with our arrange phase. (typing) So here we're creating our fixture instance, and also the instance of our system under test as before. This time instead of manually creating the email message, we're going to use AutoFixture (typing). So again we're going to use the fixture's create method and for the type that we want to create, we're going to use our email message class. (typing). Now AutoFixture is going to create an anonymous email message for us, and the act and assert phases will be the same as the previous test. So let's go ahead and build this, and we'll run our AutoFixture test, and the test passes as before. So what exactly is AutoFixture creating when it creates this email message? Well, let's put a break point and debug this test and we'll go and examine what AutoFixture sets for each property and field. So if we hover over this message and expand it, we can see what AutoFixture has created. So not only has AutoFixture created an instance of an email message for us, it's also set any public properties and any public fields to anonymous test data. So we can see here that \_somePrivateField hasn't been set by AutoFixture because it's private, and also SomePrivateProperty has also not been set by AutoFixture, again because it's private and not public. But for the public fields and properties of email message, AutoFixture has actually populated them with anonymous test data. We can also see that AutoFixture has automatically prefixed the public string properties and fields with the title of the property or field. So for example, we've got message body, SomePublicField, subject, and ToAddress prefixed and followed by the remainder of an anonymous string. So that's how we can use AutoFixture to create instances of our own custom types and how AutoFixture will automatically populate any public fields and public properties with anonymous test data.

## Creating Complex Object Graphs

So not only can AutoFixture create a complex object for us, if that object references other custom objects that we've created, AutoFixture will actually go ahead and create a complex object graph for us. So in this demo we're going to start off by looking at a test, which uses annual object creation when we're creating a complex object graph of custom types. We'll then see how AutoFixture can radically reduce the amount of code we need to write in the arrange phase. So let's head over to Visual Studio and here I've got this test, which has this arrange phase followed by a hypothetical act and assert phase. So in this arrange phase, we need to create an auto object to use in our test. If we have a look at this auto object, we can see that an order has a related customer and a list of order items that belong to this order, in addition to a date and time and an ID. The order item contains a product name and a quantity and the customer class just contains the customer name here. So in our arrange phase of our test here, to create an order means we have to create not only an order, but the customer that it belongs to, and also a list of order items. So we have quite a lot of setup code here in our arrange phase, which can be greatly simplified by using AutoFixture. So if all of this test data was a good candidate to be anonymous test data in that we didn't rely on any specific data item here for the test, we can create an AutoFixture version of this arrange phase. So let's go and do this now (typing). So I'm just going to paste in a new test method, which uses AutoFixture to create an order for us. Notice that our arrange phase is now down to just two lines of code as opposed to all of these lines of code in the manual creation version. (typing) So let's put a break point in this auto creation test and just build, and we'll go and debug this auto creation test and have a look at what AutoFixture has created for us. (typing) So let's examine this anonymous order item now. So I'll just expand this down, and the first thing we can see, as we'd expect, the simple properties ID and order date have been filled in by AutoFixture using anonymous data. But we can also see that the rest of the object graph has also been created. So for example, the related customer has been created for us and the customer name property has been populated with an anonymous string, but also the order items property, if we expand this down, has been populated with three items for us. So AutoFixture has automatically created these items. Each order item in the collection has been given an anonymous product name and an anonymous quantity. If we expand the second item, we've got a different product name and quantity, and the third item, again a different product name and quantity. So that's how we can use AutoFixture to create graphs of complex anonymous test data.

## Creating Objects with DataAnnotations

If the properties of the type we're creating has data annotations applied, AutoFixture will take notice of these. So here in Visual Studio we've got this arrange phase of a test, where we're creating an instance of this player character object. So I've got a break point here. If we debug this test, (typing) and examine the player character here, we can see that the current health property has been given an anonymous integer value and we also have these two strings, GameCharacterName and RealName. But notice these strings have different lengths. The anonymous test data for GameCharacterName is a lot shorter than the RealName string. So why is AutoFixture creating different length strings for these string properties in this automatically created class. Let's stop the test and we'll go and examine this player character class. So this player character class is using the data annotations from the system component model name space, and we can see here the GameCharacterName property has this string length attribute applied from the data annotations name space. So here we're saying that the GameCharacterName should have a maximum string length of 8, whereas notice the RealName property here doesn't have a string length attribute applied. So let's go and apply the string length attribute (typing) to the RealName property and for this let's choose a length of, say, 4. If we debug our test again now and examine the RealName property, we can see here that the RealName now has a string length of 4 instead of the default length and that's because we've applied the string length attribute here to the RealName and given it a maximum value for this string of 4.

## Creating Sequences of Custom Objects

Just as AutoFixture can create sequences of simple types, we can also use it to create sequences of our own custom objects. So in this demo we're going to see how we can create sequences of simple custom types and also what happens when we create a sequence of complex object graphs. So here in Visual Studio I've got this test class, collections of custom types, and in this first example we're creating a sequence of custom objects. So again we're using the CreateMany method from the fixture object, and here we're creating a sequence of email messages. If I put a break point here, and debug this test, (typing) if we have a look at this email's variable here, and expand the sequence, we can see that AutoFixture has created three anonymous email messages for us, each one having its property set, just in the same way as if we were creating simple types. Let's just comment this out (typing) and uncomment this, (typing) and again, just as with simple types, when we're using CreateMany we can specify how many anonymous types we want in the sequence. So here we're just generating 100 email messages and again we can debug this, (typing) and we can see this time, if we expand the results, we've now got 100 email messages created. Each email message with its own set of anonymous test data. Just as with simple types we can also use the AddManyTo method with complex types. Again I'll put a break point here and debug this test, and if we have a look at our email message buffer's emails property, we can see that we've got now 100 emails added and again, each one with its own set of anonymous test data. (typing) Just as when we use AutoFixture to create a single instance of a complex object graph, when we use the CreateMany method to create a complex object graph, so in this case 100 orders, it will still populate the object graph 100 times. So if we put a break point here and debug this, (typing) and examine this order's variable, (typing) we can see that first off we've got our 100 order objects in the collection and for each one of these we've also got the complete object graph created. We've got our customer here, and our order items here. So that's how we can use AutoFixture to not only create sequences of simple types, but also our own custom complex types.

## Summary

So that brings us to the end of this module. In this module we learned how we can use the fixture's create method and supply a generic type parameter to create anonymous strings and numbers such as ints, decimals and longs. We also learned that we can create anonymous DateTimes and anonymous TimeSpans. And as well as the create method we can also use the fixture object's CreateMany method to create a sequence of anonymous data. We also learned that if we already have an existing list and we want to add some anonymous test data to it, rather than the CreateMany method, we can instead use the fixture's AddManyTo method to add anonymous test data to the existing collection. In this module we also learned that if we have a complex object graph, that when we create an instance of the root object, AutoFixture will also create instances for any child objects and populate those child objects with anonymous test data. We also learned that if any properties of the type we're creating have the data annotation string length attribute applied, then AutoFixture will respect the maximum length property of the string length attributes. Up until this point we've mainly been relying on the default creation behavior from AutoFixture. Join me in the next module when we'll see how we can customize anonymous object creation.

# Customizing Object Creation

## Introduction

Hi. Welcome back. In the previous module we saw how we can use AutoFixture to create anonymous objects of different types, but we didn't attempt to take control of exactly how AutoFixture will be creating these anonymous objects. In this module we're going to learn how we can exercise more control over the creation of anonymous objects. In this module we're going to start off by looking at how we can customize the individual fixture instances that we create in our tests, so once we've created our fixture instance in our test code, we can tell AutoFixture to return a specific instance for a specific type of object. Next we'll learn how we can take control and customize the building of an individual object using a fixture. We'll then get a brief overview of how the AutoFixture pipeline works and how when a request comes in, AutoFixture satisfies this request and produces an anonymous object. We'll then learn how we can create a class that we can inject into the AutoFixture pipeline to customize object creation.

### Customizing Individual Fixture Instances

So let's start off by seeing how we can customize an individual fixture instance. So in this demo we're going to see how we can inject a specific instance for a specific type and then every time the fixture tries to resolve an instance of this type, we'll get our specified instance back rather than the default behavior of generating some anonymous data.

So in this demo we're going to start by trying to get AutoFixture to create an anonymous instance for a specified type and we'll see that if the type implements any kind of validation, for example, in the property setters, that we may get errors when we try to AutoFixture. We'll then see how we can customize the fixture instance and inject a specific value for a specific type to resolve this problem. We'll also learn that we can use this method to inject our own instances of our own custom types in addition to the primitive .NET types. So let's head over to Visual Studio and we're going to start off by looking at this FlightDetails class. So this class represents some flight details for an airline flight. So for example, we've got a departure airport code and arrival airport code, a flight duration, airline name, and a list of meal options. Notice in the setters for the departure airport code here, and also the arrival airport code, we've got this ensure valid airport code check. So if we look at this code, we can see that this is going to check that the airport code is the correct length, three characters long, and the airport code is all uppercase. If it's not, we'll get this application exception thrown saying that the airport code is an invalid airport. So let's go and add a test class. (typing) I'm just going to call this customizing entire fixtures (typing) and I'm just going to paste in a test method here. (typing) So we can see here that we're creating a fixture instance as before and then we're asking the fixture instance to create an instance of a flight details object. So let's build this (typing) and we'll try and run this test. Notice that this test fails and if we just expand the message here, we can see here that it's saying that the departure airport code followed by this anonymous string is an invalid airport code. So AutoFixture is generating this long anonymous string for us, but this won't actually be allowed to be set in our flight details because it doesn't match the validation criteria. It's not three characters long and it's not all uppercase. ***So what we can do here is before we try and create the instance of our flight details, we can customize this fixture instance by using its inject method. (typing) The inject method allows us to choose a concrete instance for a specific type of object. So if we want to inject a specific instance for all strings, we can do this using the inject method and simply supply a value for the type of object. So in this case we're going to create a string and we're going to inject the airport code LHR into the fixture instance.*** If I put a break point at the end of this test and debug this test, (typing) if I hover over the flight instance now, and notice that we've no longer got an exception, we can see that all of the string instances have been set to LHR. But notice here that in addition to the airport codes, it's also set the airline name to LHR, and if we expand the meal options here, it's also injected three meal option strings, again using LHR. So when we use this fixture inject method, it's going to return this instance, in this case a string saying LHR, every time the fixture tries to create an anonymous string. (typing) So if we add a new line of code here, just creating a simple anonymous string into a variable and debug this test again, we can see that our flight, anywhere where there's a string type, it's being set to LHR, but also stand alone strings have been given this LHR so anything in this test using this instance of the fixture will get LHR any time an anonymous string is requested. We'll see later in this module a more subtle solution to this problem. (typing) In addition to injecting single types to customize an individual fixture instance, we can also inject complex types. So I'm just going to paste this test in here. (typing) So in this example, again we're creating our fixture instance. This time we're using the inject method, but we're injecting an instance of a complex object, an actual instance of flight details here. Now any time this fixture instance is asked for a flight details object, it's going to return this specific instance rather than the default anonymous instance. So let's put a break point here and debug this test, and if we hover over flight 1 here, so this is our first anonymous flight details instance, and expand this down, we can see that AutoFixture has returned the object that we've injected. So, for example, we've got the departure airport code PER, and the arrival airport code LHR, and the flight duration 10 hours. If we look at the flight 2 variable here, again this has returned the identical instance, departure airport code PER, arrival airport code LHR, and flight duration of 10. So that's how we can use the fixture inject method to inject specific instances that will be returned whenever the fixture is asked for an anonymous value of that type.

## Customizing Individual Object Creation

If we need to take more control over the individual creation of objects using AutoFixture, we can do this using the ***build***

method. So in this demo we're going to see how we can use the build method to overcome one of the problems in the previous demo. We're going to look at a few different ways of doing this such as omitting the specific properties for the airline codes when the object is built using AutoFixture, and we'll also see how we can instruct AutoFixture to omit setting all of the properties when creating an object.

We'll also see how we can perform specific actions during object creation using the ***Do method***. So in Visual Studio here I've already created this test class, customizing individual object creation, and we're just going to start off with the same problem as the previous demo. We're trying to build these flights details using the default fixture behavior, and if we run this, we'll get the same error as before. We can see here that the airport code is invalid. So the first thing we're going to do, (typing) is we're going to see how rather than using the create method we can use the fixtures build method to take control over exactly how these flight details are constructed by AutoFixture. So we start by creating our fixture, (typing) and then rather than the create method, we're going to use the build method, so we're to start off by using our fixture as before, (typing) but we're going to call this build method and as the generic type parameter here, we're going to specify our flight details, and when we're using this build method, we can chain up a number of customizations (typing) and at the end of the chain, we simply call the create method to actually construct the flight details and populate the variable. ***So now we've got our basic chain of build and create, we can add some customizations before the create.*** So in this example, using the flight details class, we can't automatically set the airport code related fields because there's some validation on them. So one way around this is to instruct AutoFixture to not actually populate those specific fields. To do this we can chain the without method and provide a lambda expression to choose which property we don't want populated. (typing) So in our case, we can say that we don't want the arrival airport code populated when we build this flight details object. So now AutoFixture won't even try and populate the arrival airport code string. (typing) We can do the same thing, again calling the without method, and this time (typing), use a lambda expression to also select the departure airport code. So let's build this and try and run this test, and notice this time the test passes because we're not hitting any validation errors because we're not even trying to set the arrival airport code or the departure airport code. And if we put a break point here, and debug this, and just see what the flight details instance is here, and notice here that the airline name has still been populated with an anonymous string, as has the flight duration with an anonymous timespan and even our meal options here with anonymous strings. But notice that the arrival airport code and departure airport code remain null and that's because AutoFixture hasn't even tried to set these properties because we've specified the without method here, for both the arrival and departure airport codes. If we don't want any of the auto properties for the object actually setting, we can use a different method here, (typing) so I'm just going to paste in this test code. So again we're calling the build method and ending the build chain with the create method here, but rather than specifying specific properties to include, we're ***using the omit auto properties method.***

So now AutoFixture won't attempt to set any automatic properties of the flight details class. So we can put a break point here, just build, and debug this, and if we have a look at our constructed flight, we can see this time that none of the properties have actually been set, and that's because of this omit auto properties call. In addition to omitting specific properties or all properties we can also take control of what gets populated into specific properties. So let's start with a new test here, create our fixture, start with our build method (typing), and end with the create method, (typing) but this time rather than omitting properties we're actually going to set specific properties to specific values.

So we do this using the with method and again we can use a lambda expression here (typing) to specify what we want to set. So the first parameter of this with method is the lambda expression, so here we're going to say we want to set the arrival airport code, (typing) and the second parameter of the with method is the value that we want to set. So here I'm going to tell AutoFixture to set the arrival airport code to LAX, and we can do the same thing for the departure airport code. Again use the with method, (typing) specify a lambda expression to choose which property we want to set, this time the departure airport code, and choose a value to set. So here I'm going to choose LHR. So if you put a break point here, just build, and we'll debug this. If we have a look at our constructed flight details instance now we can see that the arrival airport code has been set to LAX, and the departure airport code has been set to LHR, as specified in our with method calls here.

***When using this build method to customize object creation, we can also perform actions during the build process. We do this using the do method.*** So let's start off here (typing) by creating a new test method, create our fixture, and we're going to start here, again calling the build method. ***We're going to specifically set the arrival and departure airport codes (typing), but in this example we're going to use the do method to call the add method of our list of meal options. So the first thing we want to do is tell AutoFixture not to automatically populate this meal options list of strings. So again to do this we use the without method as we used above and now we can use the do method to perform an action during the building of our flight details.***

So again we can use a lambda expression here. So the action we're going to perform is to call the meal options list add method and a string item to this collection. So let's add some chicken and let's duplicate this line (typing) and add some fish. Finally, we need to call the create method as before to actually create and return the instance. So let's put a break point here and build, and we'll debug this test, and if we have a look at our flight details instance now, we can see as before we've got the specific airport code set and if we have a look at our meal options this time, we don't have the automatic population because we excluded it using the without method and because of our two do methods, we've added the values chicken and fish to the meal options here.

|  |
| --- |
| var flight = fixture.Build<FlightDetails>()  .With(x => x.DepartureAirportCode, "LHR")  .With(x => x.ArrivalAirportCode, "LAX")  .Without(x => x.MealOptions)  .Do(x => x.MealOptions.Add("Chicken"))  .Do(x => x.MealOptions.Add("Fish"))  .Create(); |

## AutoFixture Pipeline Overview

Another way we can customize the creation of objects using AutoFixture is to actually hook in to the underlying AutoFixture pipeline processing. So we're going to take a look at demo of doing this in a just a moment, but first let's get an overview of this pipeline process.

So we start with a request coming into our fixture. So, for example, this could be a request for something simple such as a string, or a complex object such as flight details. So this starts the AutoFixture pipeline that consists of three phases.

1. The first phase in the AutoFixture pipeline allows us to add ***customizations*** to handle different requests, so this is a collection of customizations that by default is empty so request processing happens here unless we add our own customizations.
2. The next phase in the AutoFixture pipeline is a phase where the ***default specimen builders*** execute. So we can think of a specimen builder as something that knows how to respond to a specific request. So, for example here we have a default specimen builder that knows how to respond to a string request and generate some anonymous string data.
3. The final phase in the AutoFixture pipeline is what's known as the ***residue collectors***. So if a request can't be satisfied earlier in the pipeline, this is the last chance for AutoFixture to do some processing on the request.

***The outcome of this pipeline is our anonymous data object.***

Because the customization phase appears before the default specimen builders phase, any customizations that are added here that satisfy a request can bypass the default specimen builders that come with the default fixture pipeline. To add a customization into the AutoFixture pipeline, we need to create an object that implements the AutoFixture ISpecimenBuilder interface and then we can create a new instance of our class and add it to the collection of customizations in the AutoFixture pipeline.

## Customizing the AutoFixture Pipeline

So let's see a demo now of how we can create a custom ISpecimenBuilder class and once we've created this class, how we can add it to the fixtures customizations collection. So let's head over to Visual Studio, and we can see here that I've just got this new customizing AutoFixture pipeline test class and as before we've got the same error scenario here, (typing) and if we run, the test fails due to the validation on the airport codes as before. So let's go and look at this custom ISpecimenBuilder that I've already created. So this class implements AutoFixture's ISpecimenBuilder interface and when we implement this interface we need to provide an implementation for the create method here. So when we add this custom specimen builder to the customizations collection, all of the requests will go through our custom specimen builder here. As part of this create method we get this request object, and as we can see here, the type of this is object. So the first thing our airport code specimen builder needs to do is to determine if the current request being processed has provided a request object that we can cast to a property info object.

Because our custom specimen builder here is going to check for properties that have the word airport code in them and are of type string, if we can't cast the incoming request to a property info object, we know that we're not dealing with a property and so we can ignore the rest of our custom code specification because it doesn't apply, as it's not a property.

So first off, we perform the cast to property info and then we check that property info is null. If it's null it means that we can't convert the request to a property info object so we're not dealing with a property, which means that our customization does not apply to the current request being processed, so we want it to be processed by some other part of the pipeline later on. To tell the AutoFixture pipeline that we can't handle the current request, we simply return a new instance of the AutoFixture's no specimen class and just pass it the request object that we received so it can be processed later in the pipeline. So if this cast to property info succeeded, we know that we're dealing with a property so now we need to check, are we dealing with an airport code property. So here I've got this piece of logic. The first thing it's doing is checking if the name of the property contains the string airport code and also that the type of the property is string property. So if we had a property called airport code and it was type integer, then this would be false, but if we've got a string property that has a name that contains airport code, then this will be true. So if we do have an airport code property that we can satisfy with our custom request here, we can return a random airport code string.

Again, if it's not an airport code property, that is the name doesn't contain airport code or it's not a property of type string, again we tell AutoFixture that we can't satisfy the request in our custom request here and again we return a new instance of no specimen, passing the request, so that the request can be satisfied later in the pipeline.

Our random airport code method here is a very simple implementation, but you could imagine a more complex version, and it just returns either AAA or BBB, depending on the current time being divisible by 2. So now we have our custom specimen builder that implements ISpecimenBuilder and we have our custom logic, we can customize an instance of a fixture to use this custom specimen builder.

So let's head back to our test class here and I'm just going to paste in this new test method. So notice here after we've created our fixture instance, we're accessing its customizations property and adding an instance of a new customization, in this case our airport code specimen builder. So now when we ask this fixture instance to process a request, all of the requests will go through our custom airport code specimen builder and if it matches the logic in our airport code specimen builder, we'll get a new random airport code returned rather than a default anonymous string.

So here we're using the fixture to create an instance of our flight details as before. Let's put a break point and just build and if we run this test, it should pass, which it does. So let's just debug this test and have a look at what we've got in our flight details instance, and we can see this time our arrival airport code and departure airport code have been set to one of the random airport codes in our customized specimen builder. So at the moment both of these have been set to the letter AAA, but notice here this hasn't affected our meal options. It's only affected properties, whose names contain airport code and that's our type string. If we debug this again and have a look at our flight details, we can see this time that we've got a different set of random airport codes generated, the arrival airport has BBB, and the departure airport code has AAA.

Because our custom airport code specimen builder is going to handle any requests for string properties whose name contains airport code, if we have another type that we ask our fixture to create and this type also contains a string property containing airport code, our custom specimen builder here will still execute. So if we go to our solution explorer and have a look at this airport class, we can see this airport class has a property called airport code of type string in addition to an airline name and as before, we've got this ensure valid airport code that gets checked when we set the airport code here. (typing) So let's just uncomment this line of code, which is asking our customized fixture to create an instance of an airport class. Let's just debug this, and if we have a look at our airport instance, we can see that because we've customized our fixture and we've got a matching property here, airport code of type string, then our custom specimen builder gets executed rather than the default anonymous string behavior, but notice this hasn't affected the airline name property where we've still got the default anonymous string being generated.

## Summary

So that brings us to the end of this module. In this module we started off by learning how we can customize an entire fixture instance by using the inject method and when we use the inject method, we can provide an instance for a type that will be used any time AutoFixture generates an item of that type. Next we learned how we can customize the creation of individual objects by using the fixture's build method and how we can chain together a number of options to do things like emitting certain properties or providing values to specific properties. We also learned that we can perform actions using the do method. Next we got an overview of the AutoFixture pipeline and how the pipeline starts with a customizations phase that we can populate with our own custom ISpecimenBuilder classes. We saw the structure of an ISpecimenBuilder class, and then how to use it in a fixture by creating an instance of it and adding it to the customizations collection of our fixture instance. Join me in the final module when we'll be learning how we can use AutoFixture to write less test code and improve test maintenance by making test code more refactor safe.

# Writing Less Test Code and Improving Test Maintenance

## Introduction

Hi. Welcome back to this final module. In this module we're going to learn how we can use AutoFixture to result in less test code and how by using AutoFixture we can improve overall test maintenance and make our code more refactor safe. So to start off, in this module we're going to be looking at an example where our tests don't use AutoFixture and then we're going to go and change some production code and see that the existing tests break. We'll then reset the production code back to its previous state, refactor the unit tests to this time use AutoFixture, and then perform the same modification to the production code and notice that with using AutoFixture we no longer have to go and change all of the existing tests. Next we're going to learn how if we're using parameterized tests in xUnit.net how we can introduce AutoFixture in this use case and how we can actually get AutoFixture to inject anonymous values into our parameterized tests. In this module we'll also see how we can use AutoFixture as an auto mocking framework and how we can combine AutoFixture with the MoqU mocking framework and also the Rhino Mocks mocking framework. So we'll learn how we can configure AutoFixture to not only create our system under test, but also satisfy any dependencies of the system under test with mock objects. Finally, we'll see how we can combine the auto anonymous data approach and the auto mocking approach to greatly reduce the amount of test code in our arrange phase.

## Refactoring with Manual Test Data Creation

So let's see what happens now if we go and refactor some production code where we're not using AutoFixture. So in this demo we're going to change some production code and then we're going to see how it results in test code compilation errors and then how we have to go and waste time by manually fixing these broken tests. So let's head over to Visual Studio. So here we have a number of tests for this email message buffer class and if we have a look at this class, we can see that we get a list of emails held in this emails field here, and we have this public method, send all, which will loop through all of the emails in the emails buffer and send them. And here in the send message, we're just simulating the facts that we're sending the email with this debug write line statement. We also have this add method where we can add an email message to the message buffer, and also this property which gets the count of the number of emails left in the buffer. There's also this send limited method, which will only send a limited number of messages at one time. So let's go back to the tests and we can see that we've got three tests here, should add a message to the buffer, so we're adding a message here and then just checking that the message count is 1 and notice here that we're manually creating test data for the email message. So we're not using AutoFixture here to create any test data.

The second test here should remove message from buffer when sent. Again, we're creating some manual test data, adding the message to the buffer, and then calling send all, and then again checking that the count is now 0, because we should have sent the message. And the final test here should only send specified number of messages. This time we're creating three pieces of manual test data, these three email messages. We're adding these messages to the buffer and then we're telling the buffer to send only two messages, which should result in us having one message left in the buffer. So let's run all these to make sure they pass, which they do. So imagine a new requirement comes up that we have to set the subject on creation of an email message. So let's go and modify the production code to add a new parameter to the email message. So let's head over to the email message class, and here in the constructor, I'm just going to add a new parameter called subject, and we're going to set the subject property here to the subjects passed into the constructor. So let's try and build the solution (typing), and notice here that we get a number of compilation errors. So notice now that we have to go and modify all of these existing tests because they no longer compile. So we now need to go and expend effort by fixing these tests manually. So let's go back to our test class, and anywhere that we're creating this manual test data, we're getting a compilation error. (testing)

So let's go and add the new subject, (typing) and we'll need to go and make this change throughout this test class. (typing) So if we try and build again now it should build successfully, and the test should still pass. So we only had three existing tests here, but you can imagine in a larger code base the additional effort required to fix all of the tests. In the next demo we're going to see how we can refactor our tests to make use of AutoFixture and how by doing this will help to make them more refactor safe.

## Creating Refactor-safe Test Code

So let's see how we can help to create more refactor-safe test code. So in this demo we're going to take the tests from the previous demo and we're going to refactor them to use AutoFixture. We'll then go and change the production code again and we'll notice that this time no fixes are required in our test code.

So the first thing to notice here, I've reset this email message constructor so we no longer have the subject as a parameter, and let's go to our tests here and refactor them to use AutoFixture. So I'm going to take this first test, should add message to buffer, (typing) and I'm going to replace the existing arrange code with some code that instead uses AutoFixture. So here we're just creating the fixture instance and using AutoFixture to create an instance of our email message, and add it to the email buffer. (typing) Next we're going to do the same thing with the arrange phase for the should remove messages from buffer when sent test. So here we're going to replace this arrange code with some AutoFixture code. So again, we're just using AutoFixture to create an anonymous email message and add that to the buffer, and we'll do the same thing for the final test. (typing) Again, just replacing the existing arrange phase with the AutoFixture arrange phase. So again here we're adding three instances of email messages, but this time we're just using AutoFixture to create anonymous versions of them. So the act and the assert phase stay the same was with previous tests. So the first thing to notice is overall we have less code in our test reference here and we're not distracted by our manually created test data. So let's just build this and we'll run all the tests to make sure they pass, which they do, so let's go and change the production code as we did in the previous demo and see what happens this time. So again, let's add a new parameter for the constructor for the subject, and we'll set the property to the passed in subject. So when we did this in the previous demo and we tried to build, we got a lot of compilation failures from our unit tests. Let's see what happens now if we build.

Notice this time that the build is successful and we didn't get any compilation errors. We didn't have to go and manually expend effort by fixing up these existing tests, and if we run all these tests just to make sure they still work, they still pass.

## Parameterized xUnit.net Tests with Anonymous Data

When we're using xUnit.net and we have a similar set of tests that only differ in their test data, we can use xUnit.net's InlineData attribute to run the same test with a different set of test data. We do this by creating a parameterized test method. So in this demo we're going to start off by seeing how we can refractor three individual tests to use xUnit.net's InlineData attribute. We'll then see how we can take this one step further and introduce AutoFixture into the mix to actually create anonymous test data automatically for all of our parameters in our test method. Finally, we'll see how we can use AutoFixture's inline AutoData attribute to allow us to specify multiple test cases for a single method and also allow us to get AutoFixture to satisfy only some of the parameters. So let's head over to Visual Studio. And we can see here that we've got three tests for our calculator. The first one is to add two positive numbers. The next adds 0 and a positive number. And the third, add a negative and a positive number together. So we're just calling the add method in each case (typing), and then checking the value of the resulting calculator. So if we look at these three tests, they're all pretty much doing the same thing, testing the add method with two numbers, and then checking the resulting value. So these three tests only differ in the test data being used. So in this example we can make use of xUnit.net's theory attribute in conjunction with its InlineData attribute. So let's go and replace these three tests (typing) with a single test method, and here the first thing to notice is we've replaced the fact attribute with the theory attribute and we've got three instances of the inline data attribute, again from xUnit.net, and notice that this is replicating the data that we had in the individual tests. Adding the values 1 and 2, adding the values 0 and 2, and adding the values -5 and 1. Also notice here that our test method now has parameters so each of the values from the InlineData attribute will be supplied to the parameters A and B. We then use these parameters in our add calls, adding A and then adding B, and in our assert we're checking that whichever parameters we got passed in added together equal the calculator's value. So let's build this, and we'll run the test, (typing) and notice in the test results here we get three lots of results and we can see that the inline data attribute has been supplied the values 1 and 2, the values 0 and 2, and the values -5 and 1. So we've had this test executed three times, each time with different sets of test data. So let's see now how we can introduce AutoFixture into the mix. So if we have a look at the NuGet packages here, I've already installed this AutoFixture with xUnit.net data theory's NuGet package. This package adds a number of additional features to the basic xUnit.net package and it's going to allow us to use attributes to automatically supply parameterized tests with anonymous test data. As a quick note, note that I'm using a version of xUnit 1.9.2 as opposed to the newly released xUnit version 2. At the time of recording this course, AutoFixture's NUnit.NET specific extensions don't target the newly released xUnit 2. So let's recreate the add two positive numbers tests, but this time using AutoFixture's xUnit.net extensions to automatically supply anonymous test data. (typing) So I'm just going to paste in this new test and we can see here again we're using the standard xUnit.net theory attribute, but this time we're using the AutoData attribute from AutoFixture's xUnit.net extensions. Behind the scenes, the AutoData attribute will use a fixture instance to create sets of anonymous test data that automatically get passed to our test method parameters. So let's build this (typing) and run this test, and our test passes. Let's put a break point in our test here and debug this test. (typing) And if we have a look at our parameters here, we can see that AutoFixture has provided an anonymous integer value here, 80, and another anonymous int value here, 21. (typing) We can also get AutoFixture to provide the system under test for us, so let's add a third parameter here for our calculator and we can delete the actual manual creation line in our test code. So now, not only will AutoFixture create our values A and B, but it will also create our calculator instance for us, saving us another line in our test method body. So let's build this and run our test again, (typing) and the test still passes. If we want to specify multiple test cases for a single test method and also optionally be able to choose some values and let AutoFixture create the remaining values, we can use the InlineAuto data attribute from the AutoFixture's xUnit.net extensions. So I'm just going to paste in another test here (typing), again using the xUnit.net theory attribute, but this time using AutoFixture's xUnit.net extensions inline AutoData attribute, to simulate the three tests that we had originally. So in the first instance here, we're simulating the add two positive numbers tests. Because we haven't specified any specific values here, AutoFixture will populate all of the parameters for our test. So we'll get an anonymous A, B, and system under test created for us. In the next example here, we're simulating the add 0 and positive number test that we had originally and this time we're specifying a single parameter. So in this instance our chosen value 0 will be provided to the integer A parameter and AutoFixture will populate the rest of the parameters, in this case B and system under test, using AutoFixture. In the final test case here, we're manually specifying the first parameter value-5, so again in this case, A will be set to -5 and B and our SUT parameters will be generated by AutoFixture.

So let's build this, run our tests, and our tests now pass and if we expand the test explorer, (typing) we can see that our test method has been executed three times, the first time using all anonymous values, the second time using our specified 0 value for the parameter A, and the third time using our specified -5 parameter, for the value A, and letting AutoFixture create the rest.

## Auto Mocking with Moq

If our system under test has some dependencies, then we can use mocks to satisfy these dependencies in our test code. So in this demo we're going to start off by seeing how we can use the Moq mocking framework to create a mock dependency manually and supply it to our system under test. Next we'll see what happens by default if we ask AutoFixture to create a system under test that takes an abstract dependency such as this IEmailGateway.

Next we'll learn about the AutoMoqCustomization that we can add to a fixture instance to automatically satisfy any abstract dependencies for system under test with a Moq mock instance. And finally, we'll learn about the fixtures freeze method that allows us to get access to a mock that's automatically created by AutoFixture for our system under test. So let's start off by taking a look at this modified email message buffer class and we can see here that the constructor for this email message buffer class requires an IEmailGateway instance to be supplied to it, as a dependency. This instance is stored in this email gateway field, and if we come down and have a look at the actual send method, we can see that this stored email gateway is called whenever we want to send an email message. The IEmailGateway interface here simply just contains this send method. So let's write a test to check that the email gateway is called when sending a message. So here we're going to write a test using Moq to create a mock gateway and we'll use AutoFixture to add an anonymous email message, but at this point we're not using AutoFixture to actually create our system under test that has a dependency. So if we start off by looking at our in store NuGet packages, so notice here, I've already got the Moq NuGet package installed (typing), and we've already got this empty test class, email message buffer tests.

So let's start off with a test that manually creates a dependency for our system under test. So I' just going to paste in this test code here. So we can see in our arrange code here, we're using Moq to create a mock version of an IEmailGateway and then once we've gotten this mock version, when we create our system under test, our email message buffer, we pass this mock version to it to satisfy its dependency. We then simply add a new message using AutoFixture. In our act phase we call the send all method. And in the assert phase here we're asserting that the mock gateway's send method was called and it was called exactly once. (typing) So let's build this, and run this test, (typing) and our test passes. So let's see what happens now if we try to use AutoFixture to create our system under test. (typing) I'm just going to paste in this new test code and in our arrange phase here, we're just creating our fixture and then we're trying to use the fixture to create an instance of our email message buffer system under test. Because our email message buffer contains this abstract dependency for an IEmailGateway, by default, AutoFixture won't know how to satisfy this abstract dependency. So let's build this (typing) and let's try and run this test (typing), and notice this test fails with the message here, but AutoFixture wasn't able to create an instance of an IEmailGateway, in this case, because it's an abstract type. We can turn AutoFixture into an auto mocking container that effectively changes the AutoFixture pipeline so that any interfaces or abstract types that are not satisfied by the regular pipeline get Moq mocks created for them automatically. This means when we create our email message buffer, AutoFixture will use Moq to create a mock email gateway and automatically supply it to the email message buffer when it creates it. If we have another look at our NuGet dependencies here, we can see that we've already got this AutoFixture with auto mocking using Moq package installed and this is going to allow AutoFixture to be customized and behind the scenes use Moq to create mock dependencies. (typing) So let's see an example now of how we can also use AutoFixture as an auto mocking container. So in our arrange phase here, we're starting by creating our fixture, and then we're calling our fixture's customize method and we're passing it a new instance of this auto mock customization object and we get access to this object from the NuGet package that we just looked at. So now we've customized our fixture. When it sees our IEmailGateway dependency, it will call Moq and create a mock version of that IEmailGateway interface. And once again, we use AutoFixture to create our system under test, add an email message, and then send the emails. So if I build this and run the test, notice that this time we don't get any errors because we've added this auto mock customization so we'll get the IEmailGateway dependency satisfied by Moq. But notice in our test here, in our assert phase, we don't have a reference to the mock version of the IEmailGateway, so we can't write our assert to check that it was called. We can take advantage of the fixture's freeze method here to get a reference to the mock email gateway that will passed to the email message buffer when AutoFixture creates it. So let's add a new test, and as before, we're customizing our fixture with an auto mock customization, but this time we're calling the fixture's method. So what the freeze method will do is it will give us an instance of the type we specify in the generic type parameters and also use this same instance any time a mock IEmailGateway is asked for. So when we come to create our system under test, the mock IEmailGateway that's passed to our email message buffer, will be the same one that we received back from the freeze method in this mock gateway variable. As before, we just add a message, call the send all method, but this time because we've got access to our frozen mock gateway, we can once again verify that the send method was called. So let's build this, and run the test, (typing) and we can see this time the test passes as before. And once again, if we go and modify some production code, so in this time let's head up to the email message buffer and head up to its constructor, (typing) and I'm just going to add an arbitrary extra parameter here in the constructor (typing). If we try and build, (typing) notice that we get a compilation error again for our tests, but if we have a look at this error, it's for our manually created mock test. Once again, our AutoFixture versions don't require any additional changes just to make them compile against the new production code.

## Auto Mocking with Rhino Mocks

AutoFixture also supports a number of other mocking frameworks including Rhino Mocks, so let's have a look at this in Visual Studio. So here's our manual version of the mocking test. This time instead of using Moq, using Rhino Mocks. So we can see this follows a similar pattern. We're generating a mock gateway, this time using Rhino Mocks. We're manually creating our system under test, and passing this mock gateway to it. We're adding a message using AutoFixture, sending all the messages, and again asserting that the mock gateway's send message was called. So let's build this and run the test, (typing) and the test passes. (typing) So let me just uncomment this other test and in this version we're going to be using AutoFixture to create our system under test, whereas last time we customized our fixture to use Moq, this time we're using an auto Rhino Mock customization. (typing) This customization comes from the AutoFixture with auto mocking using Rhino Mock's NuGet package. (typing) So now we've added auto mocking support to our fixture for Rhino Mocks, we can ask AutoFixture to create our system under test, add a message, send all messages, and then assert that the mock send method was called. Notice in this version using Rhino Mocks that we can't use the freeze method as we did with Moq. That's because Rhino Mocks generates mocks in a slightly different way to Moq, but we can emulate what the freeze method does here. So first off, we're generating the mock gateway manually by calling into Rhino Mocks, so this gives us our reference to a mock gateway that we can use in our assert phase, and then we're using the inject method as we saw earlier in this course to ensure that this mock gateway instance is provided any time an IEmailGateway is requested. So if we build this, and run this test, our test passes.

## Combining Auto Mocking and Auto Data in xUnit.net

So for the final demonstration of this course, we're going to see how we can combine the concepts of auto mocking and auto data in xUnit.net. So in this demo we're going to be creating our own custom attribute called AutoMoqDataAttribute and we're going to see how we can use this to greatly reduce the arrange phase of our code. Finally, we'll see how we can also use the frozen attribute on our mock gateway to simulate calling the freeze method of a fixture. So let's head over to Visual Studio and here's our auto mocking example from before. So we're using the auto mock customization here and also calling the freeze method so we can get access to the mock gateway that we're going to use in the assert phase. Earlier in this module we saw how we can use the auto data attribute to automatically supply anonymous values to test method perimeters. We can take this concept one step further and also get the auto data attribute to supply mock values for any abstract dependencies of our system under test. So the first thing we're going to do here is create a new attribute called AutoMoqDataAttribute and here we're inheriting from AutoDataAttribute and if we have a look at this, we can see this is part of the AutoFixture xUnit extensions package. This AutoFixture AutoDataAttribute itself inherits from xUnit's data attribute, which allows data to be provided to a theory in an xUnit.net test. If we have a look here, we can see that when we create an instance of our AutoMoqDataAttribute, it's going to call into the base constructor and pass a new AutoFixture fixture and customize it, again using an auto mock customization that we saw in the previous demos. So now when we apply this AutoMoqDataAttribute to a test, not only will it supply anonymous test data, but it will also create mock versions of dependencies. So now we've defined our attribute (typing), let's uncomment this test and we can see here that we've applied the AutoMoqDataAttribute that we've created here to our test (typing), and as before, we can add some parameters to our test method that will be satisfied by AutoFixture. So the first parameter I've added here is simply an email message. The second parameter here is a mock IEmail gateway so we can have access to the mock gateway in our assert phase, and finally our system under test. So now AutoFixture will create instances of all of these parameters for us. This means in our test body here, we don't have to create any of them ourselves, which greatly reduces the amount of test code we have in our arrange phase here. So let's build this and try and run the test (typing) and notice that our test fails here and if we expand this up, we can see that the mock send method wasn't called. That's because the version of the mock gateway that we're getting supplied in this parameter here is different from the one that AutoFixture will provide when it's creating our email message buffer system under test and as before, that's because we haven't frozen this mock gateway. (typing) Here we have the same version of this test, but notice this time I've applied the frozen attribute to the MoqIEmailGateway. So just as with the freeze method that we called before, the frozen attribute will ensure that we get the same instance of the mock gateway in this mock gateway parameter here, as is supplied to the system under test. If we build this, and run this (typing), notice this time our test passes because we're getting the same mock instance (typing), used in our assert here that's supplied to our system under test. If we compare this test, which has essentially three lines of code in the test body, to the first version up here, which has many more than three lines of code, we can see by taking this approach we can significantly reduce the amount of code in our test methods, at the same time as making our code more refactor safe.

## Summary

So that brings us to the end of this module. In this module we started off by looking at how we can create more refactor safe test code by using AutoFixture to create our system under test. This means if production code changes, we won't always have to go and manually change our test code just to make the solution compile. Next we learned about auto data and inline auto data attributes that allows us to parameterize test methods and run the same test multiple times using different test data. Next we learned how we can use the auto mock customization class and customize a fixture with it to allow AutoFixture to use Moq behind the scenes to create mock versions of dependencies for our system under test. We also learned how we can do a similar thing using Rhino Mocks by using the AutoRhinoMockCustomization class. Finally, we saw how we can combine the ideas of automatic data and auto mocking to create our own auto mock data attributes that we can apply to our test methods. We also saw when doing this that we can apply the frozen attribute to one of our test parameters if we want the same instance of that object to be reused throughout the test. If you want to learn more about AutoFixture, be sure to check out the AutoFixture GitHub page here, and if you want to find out more about the creator of AutoFixture, Mark Seemann, you can find him at his blog, and also on Twitter. If you want to learn more about data driven tests using NUnit or xUnit.net, be sure to check out the other Pluralsight courses. I'm Jason Roberts from Pluralsight and I hope you enjoyed this course.