Getting started

Adding NSubstitute to your test project

First add the [NSubstitute NuGet package](https://nuget.org/List/Packages/NSubstitute) to your test project using [NuGet](https://docs.microsoft.com/en-us/nuget/quickstart/use-a-package) (either the command line executable, or via the package manager in your IDE).

> Install-Package NSubstitute

It is optional but recommended to also install [NSubstitute.Analyzers.CSharp](https://www.nuget.org/packages/NSubstitute.Analyzers.CSharp/) for C# projects, or [NSubstitute.Analyzers.VisualBasic](https://www.nuget.org/packages/NSubstitute.Analyzers.VisualBasic/) for VB projects. NSubstitute will work without the analysers installed, but these packages will [help detect potential misuses of the NSubstitute API](https://nsubstitute.github.io/help/nsubstitute-analysers/).

> Install-Package NSubstitute.Analyzers.CSharp

// or

> Install-Package NSubstitute.Analyzers.VisualBasic

Using NSubstitute in a test fixture

So now you are staring at a blank test fixture (created with your favourite unit testing framework; for these examples we’re using [NUnit](http://www.nunit.org/)), and are wondering where to start.

First, add using NSubstitute; to your current C# file. This will give you everything you need to start substituting.

Now let’s say we have a basic calculator interface:

|  |
| --- |
| public interface ICalculator  {      int Add(int a, int b);      string Mode { get; set; }      event EventHandler PoweringUp;  } |

We can ask NSubstitute to create a substitute instance for this type. We could ask for a stub, mock, fake, spy, test double etc., but why bother when we just want to substitute an instance we have some control over?

|  |
| --- |
| calculator = Substitute.For<ICalculator>(); |

⚠️ Note: NSubstitute will only work properly with interfaces or with virtual members of classes. Be careful substituting for classes with non-virtual members. See [Creating a substitute](https://nsubstitute.github.io/help/creating-a-substitute/#substituting_infrequently_and_carefully_for_classes) for more information.

Now we can tell our substitute to return a value for a call:

|  |
| --- |
| calculator.Add(1, 2).Returns(3);  Assert.That(calculator.Add(1, 2), Is.EqualTo(3)); |

We can check that our substitute received a call, and did not receive others:

|  |
| --- |
| calculator.Add(1, 2);  calculator.Received().Add(1, 2);  calculator.DidNotReceive().Add(5, 7); |

If our Received() assertion fails, NSubstitute tries to give us some help as to what the problem might be:

NSubstitute.Exceptions.ReceivedCallsException : Expected to receive a call matching:

Add(1, 2)

Actually received no matching calls.

Received 2 non-matching calls (non-matching arguments indicated with '\*' characters):

Add(\*4\*, \*7\*)

Add(1, \*5\*)

We can also work with properties using the Returns syntax we use for methods, or just stick with plain old property setters (for read/write properties):

|  |
| --- |
| calculator.Mode.Returns("DEC");  Assert.That(calculator.Mode, Is.EqualTo("DEC"));    calculator.Mode = "HEX";  Assert.That(calculator.Mode, Is.EqualTo("HEX")); |

NSubstitute supports [argument matching](https://nsubstitute.github.io/help/argument-matchers/) for setting return values and asserting a call was received:

|  |
| --- |
| calculator.Add(10, -5);  calculator.Received().Add(10, Arg.Any<int>());  calculator.Received().Add(10, Arg.Is<int>(x => x < 0)); |

We can use argument matching as well as passing a function to Returns() to get some more behaviour out of our substitute (possibly too much, but that’s your call):

|  |
| --- |
| calculator     .Add(Arg.Any<int>(), Arg.Any<int>())     .Returns(x => (int)x[0] + (int)x[1]);  Assert.That(calculator.Add(5, 10), Is.EqualTo(15)); |

Returns() can also be called with multiple arguments to set up a sequence of return values.

|  |
| --- |
| calculator.Mode.Returns("HEX", "DEC", "BIN");  Assert.That(calculator.Mode, Is.EqualTo("HEX"));  Assert.That(calculator.Mode, Is.EqualTo("DEC"));  Assert.That(calculator.Mode, Is.EqualTo("BIN")); |

Finally, we can raise events on our substitutes (unfortunately C# dramatically restricts the extent to which this syntax can be cleaned up):

|  |
| --- |
| bool eventWasRaised = false;  calculator.PoweringUp += (sender, args) => eventWasRaised = true;    calculator.PoweringUp += Raise.Event();  Assert.That(eventWasRaised); |

That’s pretty much all you need to get started with NSubstitute. Read on for more detailed feature descriptions, as well as for some of the less common requirements that NSubstitute supports.

[Creating a substitute »](https://nsubstitute.github.io/help/creating-a-substitute/)

Creating a substitute

The basic syntax for creating a substitute is:

|  |
| --- |
| var substitute = Substitute.For<ISomeInterface>(); |

This is how you’ll normally create substitutes for types. Generally this type will be an interface, but you can also substitute classes in cases of emergency.

Substituting (infrequently and carefully) for classes

⚠️ Warning: Substituting for classes can have some nasty side-effects!

For starters, NSubstitute can only work with *virtual* members of the class, so any non-virtual code in the class will actually execute! If you try to substitute for a class that formats your hard drive in the constructor or in a non-virtual property setter then you’re asking for trouble.

It also means features like Received(), Returns(), Arg.Is(), Arg.Any() and When()..Do() will not work with these non-virtual members. For example: subClass.Received().NonVirtualCall() will not actually run an assertion (it will always pass, even if there are no calls to NonVirtualCall()), and can even cause confusing problems with later tests. These features will work correctly with virtual members of the class, but we have to be careful to avoid the non-virtual ones.

For these reasons we strongly recommend using [NSubstitute.Analyzers](https://nsubstitute.github.io/help/nsubstitute-analysers/) to detect these cases, and sticking to substituting for interfaces as much as possible. (Interfaces are always safe to substitute and do not suffer from any of the limitations that class substitutes do.)

With the knowledge that we’re not going to be substituting for classes, here is how you create a substitute for a class that has constructor arguments:

|  |
| --- |
| var someClass = Substitute.For<SomeClassWithCtorArgs>(5, "hello world"); |

For classes that have default constructors the syntax is the same as substituting for interfaces.

Substituting for multiple interfaces

There are times when you want to substitute for multiple types. The best example of this is when you have code that works with a type, then checks whether it implements IDisposable and disposes of it if it doesn’t.

|  |
| --- |
| var command = Substitute.For<ICommand, IDisposable>();  var runner = new CommandRunner(command);    runner.RunCommand();    command.Received().Execute();  ((IDisposable)command).Received().Dispose(); |

Your substitute can implement several types this way, but remember you can only implement a maximum of one class. You can specify as many interfaces as you like, but only one of these can be a class. The most flexible way of creating substitutes for multiple types is using this overload:

|  |
| --- |
| var substitute = Substitute.For(          new[] { typeof(ICommand), typeof(ISomeInterface), typeof(SomeClassWithCtorArgs) },          new object[] { 5, "hello world" }      );  Assert.IsInstanceOf<ICommand>(substitute);  Assert.IsInstanceOf<ISomeInterface>(substitute);  Assert.IsInstanceOf<SomeClassWithCtorArgs>(substitute); |

Substituting for delegates

NSubstitute can also substitute for delegate types by using Substiute.For<T>(). When substituting for delegate types you will not be able to get the substitute to implement additional interfaces or classes.

|  |
| --- |
| var func = Substitute.For<Func<string>>();    func().Returns("hello");  Assert.AreEqual("hello", func()); |

Partial substitutes and test spies

When required we can also create substitutes that run real code by default, letting us replace [specific parts of a class with substitute behaviour](https://nsubstitute.github.io/help/partial-subs/).

[« Getting started](https://nsubstitute.github.io/help/getting-started/)

[Setting a return value »](https://nsubstitute.github.io/help/set-return-value/)

Setting a return value

The following examples relate to substituting for the following interface:

|  |
| --- |
| public interface ICalculator {      int Add(int a, int b);      string Mode { get; set; }  } |

For methods

To set a return value for a method call on a substitute, call the method as normal, then follow it with a call to NSubstitute’s Returns() extension method.

|  |
| --- |
| var calculator = Substitute.For<ICalculator>();  calculator.Add(1, 2).Returns(3); |

This value will be returned every time this call is made. Returns() will only apply to this combination of arguments, so other calls to this method will return a default value instead.

|  |
| --- |
| //Make a call return 3:  calculator.Add(1, 2).Returns(3);  Assert.AreEqual(calculator.Add(1, 2), 3);  Assert.AreEqual(calculator.Add(1, 2), 3);    //Call with different arguments does not return 3  Assert.AreNotEqual(calculator.Add(3, 6), 3); |

For properties

The return value for a property can be set in the same was as for a method, using Returns(). You can also just use plain old property setters for read/write properties; they’ll behave just the way you expect them to.

|  |
| --- |
| calculator.Mode.Returns("DEC");  Assert.AreEqual(calculator.Mode, "DEC");    calculator.Mode = "HEX";  Assert.AreEqual(calculator.Mode, "HEX"); |

More ways of setting return values

This covers the very basics of setting a return value, but NSubstitute can do much more. Read on for other approachs, including [matching specific arguments](https://nsubstitute.github.io/help/return-for-args), [ignoring arguments](https://nsubstitute.github.io/help/return-for-any-args), using [functions to calculate return values](https://nsubstitute.github.io/help/return-from-function) and returning [multiple results](https://nsubstitute.github.io/help/multiple-returns).

[« Creating a substitute](https://nsubstitute.github.io/help/creating-a-substitute/)

[Return for specific args »](https://nsubstitute.github.io/help/return-for-args/)

Return for specific args

Return values can be configured for different combinations of arguments passed to calls using [argument matchers](https://nsubstitute.github.io/help/argument-matchers). This topic is covered in more detail in the [Argument matchers](https://nsubstitute.github.io/help/argument-matchers) entry, but the following examples show the general idea.

|  |
| --- |
| //Return when first arg is anything and second arg is 5:  calculator.Add(Arg.Any<int>(), 5).Returns(10);  Assert.AreEqual(10, calculator.Add(123, 5));  Assert.AreEqual(10, calculator.Add(-9, 5));  Assert.AreNotEqual(10, calculator.Add(-9, -9));    //Return when first arg is 1 and second arg less than 0:  calculator.Add(1, Arg.Is<int>(x => x < 0)).Returns(345);  Assert.AreEqual(345, calculator.Add(1, -2));  Assert.AreNotEqual(345, calculator.Add(1, 2));    //Return when both args equal to 0:  calculator.Add(Arg.Is(0), Arg.Is(0)).Returns(99);  Assert.AreEqual(99, calculator.Add(0, 0)); |

[« Setting a return value](https://nsubstitute.github.io/help/set-return-value/)

[Return for any args »](https://nsubstitute.github.io/help/return-for-any-args/)

Return for any args

A call can be configured to return a value regardless of the arguments passed using the ReturnsForAnyArgs() extension method.

|  |
| --- |
| calculator.Add(1, 2).ReturnsForAnyArgs(100);  Assert.AreEqual(100, calculator.Add(1, 2));  Assert.AreEqual(100, calculator.Add(-7, 15)); |

The same behaviour can also be achieved using [argument matchers](https://nsubstitute.github.io/help/argument-matchers): it is simply a shortcut for replacing each argument with Arg.Any<T>().

ReturnsForAnyArgs() has the same overloads as Returns(), so you can also specify multiple return values or calculated return values using this approach.

[« Return for specific args](https://nsubstitute.github.io/help/return-for-args/)

[Return from a function »](https://nsubstitute.github.io/help/return-from-function/)

Return from a function

The return value for a call to a property or method can be set to the result of a function. This allows more complex logic to be put into the substitute. Although this is normally a bad practice, there are some situations in which it is useful.

|  |
| --- |
| calculator      .Add(Arg.Any<int>(), Arg.Any<int>())      .Returns(x => (int)x[0] + (int)x[1]);    Assert.That(calculator.Add(1, 1), Is.EqualTo(2));  Assert.That(calculator.Add(20, 30), Is.EqualTo(50));  Assert.That(calculator.Add(-73, 9348), Is.EqualTo(9275)); |

In this example [argument matchers](https://nsubstitute.github.io/help/argument-matchers) are used to match all calls to Add(), and a lambda function is used to return the sum of the first and second arguments passed to the call.

Call information

The function we provide to Returns() and ReturnsForAnyArgs() is of type Func<CallInfo,T>, where T is the type the call is returning, and CallInfo is a type which provides access to the arguments used for the call. In the previous example we accessed these arguments using an indexer (x[1] for the second argument). CallInfo also has a couple of convenience methods to pick arguments in a strongly typed way:

T Arg<T>(): Gets the argument of type T passed to this call.

T ArgAt<T>(int position): Gets the argument passed to this call at the specified zero-based position, converted to type T.

|  |
| --- |
| public interface IFoo {      string Bar(int a, string b);  } |
| var foo = Substitute.For<IFoo>();  foo.Bar(0, "").ReturnsForAnyArgs(x => "Hello " + x.Arg<string>());  Assert.That(foo.Bar(1, "World"), Is.EqualTo("Hello World")); |

Here x.Arg<string>() will return the string argument passed to the call, rather than having to use (string) x[1]. If there are two string arguments to a call, NSubstitute will throw an exception and let you know that it can’t work out which argument you mean.

Callbacks

This technique can also be used to get a callback whenever a call is made:

|  |
| --- |
| var counter = 0;  calculator      .Add(0, 0)      .ReturnsForAnyArgs(x => {          counter++;          return 0;      });    calculator.Add(7,3);  calculator.Add(2,2);  calculator.Add(11,-3);  Assert.AreEqual(counter, 3); |

Alternatively the callback can be specified after the Returns using AndDoes:

|  |
| --- |
| var counter = 0;  calculator      .Add(0, 0)      .ReturnsForAnyArgs(x => 0)      .AndDoes(x => counter++);    calculator.Add(7,3);  calculator.Add(2,2);  Assert.AreEqual(counter, 2); |

[« Return for any args](https://nsubstitute.github.io/help/return-for-any-args/)

[Multiple return values »](https://nsubstitute.github.io/help/multiple-returns/)

Multiple return values

A call can also be configured to return a different value over multiple calls. The following example shows this for a call to a property, but it works the same way for method calls.

|  |
| --- |
| calculator.Mode.Returns("DEC", "HEX", "BIN");  Assert.AreEqual("DEC", calculator.Mode);  Assert.AreEqual("HEX", calculator.Mode);  Assert.AreEqual("BIN", calculator.Mode); |

This can also be achieved by [returning from a function](https://nsubstitute.github.io/help/return-from-function), but passing multiple values to Returns() is simpler and reads better.

Multiple returns using callbacks

Returns() also supports passing multiple [functions to return from](https://nsubstitute.github.io/help/return-from-function), which allows one call in a sequence to throw an exception or perform some other action.

|  |
| --- |
| calculator.Mode.Returns(x => "DEC", x => "HEX", x => { throw new Exception(); });  Assert.AreEqual("DEC", calculator.Mode);  Assert.AreEqual("HEX", calculator.Mode);  Assert.Throws<Exception>(() => { var result = calculator.Mode; }); |

Configuring other calls without using up multiple returns

If a call has been configured with multiple returns values, you can configure a more specific call without using up any of these callbacks using [.Configure()](https://nsubstitute.github.io/help/configure/).

[« Return from a function](https://nsubstitute.github.io/help/return-from-function/)

[Replacing return values »](https://nsubstitute.github.io/help/replacing-return-values/)

Replacing return values

The return value for a method or property can be set as many times as required. Only the most recently set value will be returned.

|  |
| --- |
| calculator.Mode.Returns("DEC,HEX,OCT");  calculator.Mode.Returns(x => "???");  calculator.Mode.Returns("HEX");  calculator.Mode.Returns("BIN");  Assert.AreEqual(calculator.Mode, "BIN"); |

[« Multiple return values](https://nsubstitute.github.io/help/multiple-returns/)

[Checking received calls »](https://nsubstitute.github.io/help/received-calls/)

Checking received calls

In some cases (particularly for void methods) it is useful to check that a specific call has been received by a substitute. This can be checked using the Received() extension method, followed by the call being checked.

|  |
| --- |
| public interface ICommand {      void Execute();      event EventHandler Executed;  }    public class SomethingThatNeedsACommand {      ICommand command;      public SomethingThatNeedsACommand(ICommand command) {          this.command = command;      }      public void DoSomething() { command.Execute(); }      public void DontDoAnything() { }  }    [Test]  public void Should\_execute\_command() {      //Arrange      var command = Substitute.For<ICommand>();      var something = new SomethingThatNeedsACommand(command);      //Act      something.DoSomething();      //Assert      command.Received().Execute();  } |

In this case command did receive a call to Execute(), and so will complete successfully. If Execute() has not been received NSubstitute will throw a ReceivedCallsException and let you know what call was expected and with which arguments, as well as listing actual calls to that method and which the arguments differed.

Check a call was not received

NSubstitute can also make sure a call was not received using the DidNotReceive() extension method.

|  |
| --- |
| var command = Substitute.For<ICommand>();  var something = new SomethingThatNeedsACommand(command);  //Act  something.DontDoAnything();  //Assert  command.DidNotReceive().Execute(); |

Check a call was received a specific number of times

The Received() extension method will assert that at least one call was made to a member, and DidNotReceive() asserts that zero calls were made. NSubstitute also gives you the option of asserting a specific number of calls were received by passing an integer to Received(). This will throw if the substitute does not receive exactly that many matching calls. Too few, or too many, and the assertion will fail.

|  |
| --- |
| public class CommandRepeater {      ICommand command;      int numberOfTimesToCall;      public CommandRepeater(ICommand command, int numberOfTimesToCall) {        this.command = command;        this.numberOfTimesToCall = numberOfTimesToCall;      }        public void Execute() {        for (var i=0; i<numberOfTimesToCall; i++) command.Execute();      }  }    [Test]  public void Should\_execute\_command\_the\_number\_of\_times\_specified() {    var command = Substitute.For<ICommand>();    var repeater = new CommandRepeater(command, 3);    //Act    repeater.Execute();    //Assert    command.Received(3).Execute(); // << This will fail if 2 or 4 calls were received  } |

We can also use Received(1) to check a call was received once and only once. This differs from the standard Received() call, which checks a call was received at least once. Received(0) behaves the same as DidNotReceive().

Received (or not) with specific arguments

We can also use [argument matchers](https://nsubstitute.github.io/help/argument-matchers) to check calls were received (or not) with particular arguments. This is covered in more detail in the [argument matchers](https://nsubstitute.github.io/help/argument-matchers) topic, but the following examples show the general idea:

|  |
| --- |
| calculator.Add(1, 2);  calculator.Add(-100, 100);    //Check received with second arg of 2 and any first arg:  calculator.Received().Add(Arg.Any<int>(), 2);  //Check received with first arg less than 0, and second arg of 100:  calculator.Received().Add(Arg.Is<int>(x => x < 0), 100);  //Check did not receive a call where second arg is >= 500 and any first arg:  calculator      .DidNotReceive()      .Add(Arg.Any<int>(), Arg.Is<int>(x => x >= 500)); |

Ignoring arguments

NSubstitute can also check calls were received or not received but ignore the arguments used, just like we can for [setting returns for any arguments](https://nsubstitute.github.io/help/return-for-any-args). In this case we need ReceivedWithAnyArgs() and DidNotReceiveWithAnyArgs().

|  |
| --- |
| calculator.Add(1, 3);    calculator.ReceivedWithAnyArgs().Add(1,1);  calculator.DidNotReceiveWithAnyArgs().Subtract(0,0); |

Checking calls to properties

The same syntax can be used to check calls on properties. Normally we’d want to avoid this, as we’re really more interested in testing the required behaviour rather than the precise implementation details (i.e. we would set the property to return a value and check that was used properly, rather than assert that the property getter was called). Still, there are probably times when checking getters and setters were called can come in handy, so here’s how you do it:

|  |
| --- |
| var mode = calculator.Mode;  calculator.Mode = "TEST";    //Check received call to property getter  //We need to assign the result to a variable to keep  //the compiler happy.  var temp = calculator.Received().Mode;    //Check received call to property setter with arg of "TEST"  calculator.Received().Mode = "TEST"; |

Checking calls to indexers

An indexer is really just another property, so we can use the same syntax to check calls to indexers.

|  |
| --- |
| var dictionary = Substitute.For<IDictionary<string, int>>();  dictionary["test"] = 1;    dictionary.Received()["test"] = 1;  dictionary.Received()["test"] = Arg.Is<int>(x => x < 5); |

Checking event subscriptions

As with properties, we’d normally favour testing the required behaviour over checking subscriptions to particular event handlers. We can do that by [raising an event on the substitute](https://nsubstitute.github.io/help/raising-events/) and asserting our class performs the correct behaviour in response:

|  |
| --- |
| public class CommandWatcher {      ICommand command;      public CommandWatcher(ICommand command) {          command.Executed += OnExecuted;      }      public bool DidStuff { get; private set; }      public void OnExecuted(object o, EventArgs e) { DidStuff = true; }  }    [Test]  public void ShouldDoStuffWhenCommandExecutes() {    var command = Substitute.For<ICommand>();    var watcher = new CommandWatcher(command);      command.Executed += Raise.Event();      Assert.That(watcher.DidStuff);  } |

If required though, Received will let us assert that the subscription was received:

|  |
| --- |
| [Test]  public void MakeSureWatcherSubscribesToCommandExecuted() {    var command = Substitute.For<ICommand>();    var watcher = new CommandWatcher(command);      // Not recommended. Favour testing behaviour over implementation specifics.    // Can check subscription:    command.Received().Executed += watcher.OnExecuted;    // Or, if the handler is not accessible:    command.Received().Executed += Arg.Any<EventHandler>();  } |

Checking event invocation

We can also use substitutes for event handlers to confirm that a particular event was raised correctly. Often a simple lambda function will suffice, but if we want to use argument matchers we can use a substitute and Received. Both options are shown below:

|  |
| --- |
| public class LowFuelWarningEventArgs : EventArgs {      public int PercentLeft { get; }      public LowFuelWarningEventArgs(int percentLeft){          PercentLeft = percentLeft;      }  }    public class FuelManagement{      public event EventHandler<LowFuelWarningEventArgs> LowFuelDetected;      public void DoSomething(){          LowFuelDetected?.Invoke(this, new LowFuelWarningEventArgs(15));      }  }    // Often it is easiest to use a lambda for this, as shown in the following test:  [Test]  public void ShouldRaiseLowFuel\_WithoutNSub(){      var fuelManagement = new FuelManagement();      var eventWasRaised = false;      fuelManagement.LowFuelDetected += (o,e) => eventWasRaised = true;        fuelManagement.DoSomething();        Assert.That(eventWasRaised);  }    // We can also use NSubstitute for this if we want more involved argument matching logic.  // NSubstitute also gives us a descriptive message if the assertion fails which may be helpful in some cases.  // (For example, if the call was not received with the expected arguments, we'll get a list of the non-matching  // calls made to that member.)  //  // Note we could still use lambdas and standard assertions for this, but a substitute may be worth considering  // in some of these cases.  [Test]  public void ShouldRaiseLowFuel(){      var fuelManagement = new FuelManagement();      var handler = Substitute.For<EventHandler<LowFuelWarningEventArgs>>();      fuelManagement.LowFuelDetected += handler;        fuelManagement.DoSomething();        handler          .Received()          .Invoke(fuelManagement, Arg.Is<LowFuelWarningEventArgs>(x => x.PercentLeft < 20));  } |

[« Replacing return values](https://nsubstitute.github.io/help/replacing-return-values/)

[Clearing received calls »](https://nsubstitute.github.io/help/clear-received-calls/)

# Clearing received calls

A substitute can forget all the calls previously made to it using the ClearReceivedCalls() extension method.

Say we have an ICommand interface, and we want a OnceOffCommandRunner that will take an ICommand and only run it once.

|  |
| --- |
| public interface ICommand {      void Execute();  }    public class OnceOffCommandRunner {      ICommand command;      public OnceOffCommandRunner(ICommand command) {          this.command = command;      }      public void Run() {          if (command == null) return;          command.Execute();          command = null;      }  } |

If we substitute for ICommand we can test it is called on the first run, then forget any previous calls made to it, and make sure it is not called again.

|  |
| --- |
| var command = Substitute.For<ICommand>();  var runner = new OnceOffCommandRunner(command);    //First run  runner.Run();  command.Received().Execute();    //Forget previous calls to command  command.ClearReceivedCalls();    //Second run  runner.Run();  command.DidNotReceive().Execute(); |

ClearReceivedCalls() will not clear any results set up for the substitute using Returns(). If we need to this, we can [replace previously specified results](https://nsubstitute.github.io/help/replacing-return-values) by calling Returns() again.

[« Checking received calls](https://nsubstitute.github.io/help/received-calls/)

[Argument matchers »](https://nsubstitute.github.io/help/argument-matchers/)

# Argument matchers

Argument matchers can be used when [setting return values](https://nsubstitute.github.io/help/return-for-args) and when [checking received calls](https://nsubstitute.github.io/help/received-calls). They provide a way to specify a call or group of calls, so that a return value can be set for all matching calls, or to check a matching call has been received.

The argument matchers syntax shown here depends on having C# 7.0 or later. If you are stuck on an earlier version (getting an error such as CS7085: By-reference return type 'ref T' is not supported while trying to use them) please use [compatibility argument matchers](https://nsubstitute.github.io/help/compat-args) instead.

⚠️ **Note:** Argument matchers should only be used when setting return values or checking received calls. Using Arg.Is or Arg.Any without a call to Returns(...) or Received() can cause your tests to behave in unexpected ways. See [How NOT to use argument matchers](https://nsubstitute.github.io/help/argument-matchers/#how_not_to_use_argument_matchers) for more information.

## Ignoring arguments

An argument of type T can be ignored using Arg.Any<T>().

|  |
| --- |
| calculator.Add(Arg.Any<int>(), 5).Returns(7);    Assert.AreEqual(7, calculator.Add(42, 5));  Assert.AreEqual(7, calculator.Add(123, 5));  Assert.AreNotEqual(7, calculator.Add(1, 7)); |

In this example we return 7 when adding any number to 5. We use Arg.Any<int>() to tell NSubstitute to ignore the first argument.

We can also use this to match any argument of a specific sub-type.

|  |
| --- |
| formatter.Format(new object());  formatter.Format("some string");    formatter.Received().Format(Arg.Any<object>());  formatter.Received().Format(Arg.Any<string>());  formatter.DidNotReceive().Format(Arg.Any<int>()); |

## Conditionally matching an argument

An argument of type T can be conditionally matched using Arg.Is<T>(Predicate<T> condition).

|  |
| --- |
| calculator.Add(1, -10);    //Received call with first arg 1 and second arg less than 0:  calculator.Received().Add(1, Arg.Is<int>(x => x < 0));  //Received call with first arg 1 and second arg of -2, -5, or -10:  calculator      .Received()      .Add(1, Arg.Is<int>(x => new[] {-2,-5,-10}.Contains(x)));  //Did not receive call with first arg greater than 10:  calculator.DidNotReceive().Add(Arg.Is<int>(x => x > 10), -10); |

If the condition throws an exception for an argument, then it will be assumed that the argument does not match. The exception itself will be swallowed.

|  |
| --- |
| formatter.Format(Arg.Is<string>(x => x.Length <= 10)).Returns("matched");    Assert.AreEqual("matched", formatter.Format("short"));  Assert.AreNotEqual("matched", formatter.Format("not matched, too long"));  // Will not match because trying to access .Length on null will throw an exception when testing  // our condition. NSubstitute will assume it does not match and swallow the exception.  Assert.AreNotEqual("matched", formatter.Format(null)); |

## Matching a specific argument

An argument of type T can be matched using Arg.Is<T>(T value).

|  |
| --- |
| calculator.Add(0, 42);    //This won't work; NSubstitute isn't sure which arg the matcher applies to:  //calculator.Received().Add(0, Arg.Any<int>());    calculator.Received().Add(Arg.Is(0), Arg.Any<int>()); |

This matcher normally isn’t required; most of the time we can just use 0 instead of Arg.Is(0). In some cases though, NSubstitute can’t work out which matcher applies to which argument (arg matchers are actually fuzzily matched; not passed directly to the function call). In these cases it will throw an AmbiguousArgumentsException and ask you to specify one or more additional argument matchers. In some cases you may have to explicitly use argument matchers for every argument.

## Matching out and ref args

Argument matchers can also be used with out and ref (NSubstitute 4.0 and later with C# 7.0 and later).

|  |
| --- |
| calculator      .LoadMemory(1, out Arg.Any<int>())      .Returns(x => {          x[1] = 42;          return true;      });    var hasEntry = calculator.LoadMemory(1, out var memoryValue);  Assert.AreEqual(true, hasEntry);  Assert.AreEqual(42, memoryValue); |

See [Setting out and ref args](https://nsubstitute.github.io/help/setting-out-and-ref-arguments/) for more information on working with out and ref.

## How NOT to use argument matchers

Occasionally argument matchers get used in ways that cause unexpected results for people. Here are the most common ones.

### Using matchers outside of stubbing or checking received calls

Argument matchers should only be used when setting return values or checking received calls. Using Arg.Is or Arg.Any without a call to Returns(...) or Received() can cause your tests to behave in unexpected ways.

For example:

|  |
| --- |
| /\* ARRANGE \*/    var widgetFactory = Substitute.For<IWidgetFactory>();  var subject = new Sprocket(widgetFactory);    // OK: Use arg matcher for a return value:  widgetFactory.Make(Arg.Is<int>(x => x > 10)).Returns(TestWidget);    /\* ACT \*/    // NOT OK: arg matcher used with a real call:  //   subject.StartWithWidget(Arg.Any<int>());    // Use a real argument instead:  subject.StartWithWidget(4);    /\* ASSERT \*/    // OK: Use arg matcher to check a call was received:  widgetFactory.Received().Make(Arg.Is<int>(x => x > 0)); |

In this example it would be an error to use an argument matcher in the ACT part of this test. Even if we don’t mind what specific argument we pass to our subject, Arg.Any is only for substitutes, and only for setting return values or checking received calls; not for real calls.

(If you do want to indicate to readers that the precise argument used for a real call doesn’t matter you could use a variable such as var someInt = 4; subject.StartWithWidget(someInt); or similar. Just stay clear of argument matchers for this!)

### Modifying values being matched

When NSubstitute records calls, it keeps a reference to the arguments passed, not a deep clone of each argument at the time of the call. This means that if the properties of an argument change after the call assertions may not behave as expected.

|  |
| --- |
| public class Person {      public string Name { get; set; }  }    [Test]  public void MutatingAMatchedArgument() {      var person = new Person { Name = "Carrot" };      var lookup = Substitute.For<IPersonLookup>();        // Called with a Person that has a .Name property of "Carrot"      lookup.Add(person);        // The Name in that person reference later gets updated ...      person.Name = "Vimes";        // When the substitute is queried, it will check the fields of the person reference it was called with.      // This means the argument it was called with does NOT have a .Name of "Carrot" (it was changed!)      lookup.DidNotReceive().Add(Arg.Is<Person>(p => p.Name == "Carrot"));      // Instead, it now has the updated name:      lookup.Received().Add(Arg.Is<Person>(p => p.Name == "Vimes"));  } |

This looks confusing at first, but if we remember substitutes are pretty much forced to store references to arguments used then it makes sense. The alternative of storing deep-cloned snapshots of every argument to every call received is fairly impractical, especially if we consider objects with very complex hierarchies (e.g. tens of fields, each with an object with tens of fields of its own, etc.). Storing snapshots would also lead to the same confusion in the reverse situation, where we know a substitute was called with a particular reference but the Arg.Is(person) check fails due to a change in one of its fields.

That said, there are times when snapshots like this are useful, and there are a few ways to enable this with NSubstitute.

The first option is to use structs instead of classes for these cases. These are passed by value rather than by reference, so that value will be stored by substitutes and modifications made afterwards will not affect that value.

|  |
| --- |
| public struct PersonStruct {      public string Name { get; set; }  }    [Test]  public void MutatingAStruct() {      var person = new PersonStruct { Name = "Carrot" };      var lookup = Substitute.For<IPersonStructLookup>();        lookup.Add(person);        person.Name = "Vimes";        // `person` was passed by value, and that value still has the original Name      lookup.Received().Add(Arg.Is<PersonStruct>(p => p.Name == "Carrot"));  } |

For cases where that is not possible or wanted then we can manually snapshot the values we are interested in.

|  |
| --- |
| [Test]  public void ManualArgSnapshot() {      var person = new Person { Name = "Carrot" };      var lookup = Substitute.For<IPersonLookup>();      var namesAdded = new List<string>();      // Manually snapshot the value or values we care about:      lookup.Add(Arg.Do<Person>(p => namesAdded.Add(p.Name)));          lookup.Add(person);      person.Name = "Vimes";        Assert.AreEqual("Carrot", namesAdded[0]);  } |

We can then use our standard assertion library for checking the value. This approach can also be helpful for asserting on complex objects, as our assertions can be more detailed and provide more useful information than NSubstitute typically provides in these cases.

[« Clearing received calls](https://nsubstitute.github.io/help/clear-received-calls/)

[Callbacks, void calls and When..Do »](https://nsubstitute.github.io/help/callbacks/)

* [Getting started](https://nsubstitute.github.io/help/getting-started/)
* [Creating a substitute](https://nsubstitute.github.io/help/creating-a-substitute/)
* [Setting a return value](https://nsubstitute.github.io/help/set-return-value/)
* [Return for specific args](https://nsubstitute.github.io/help/return-for-args/)
* [Return for any args](https://nsubstitute.github.io/help/return-for-any-args/)
* [Return from a function](https://nsubstitute.github.io/help/return-from-function/)
* [Multiple return values](https://nsubstitute.github.io/help/multiple-returns/)
* [Replacing return values](https://nsubstitute.github.io/help/replacing-return-values/)
* [Checking received calls](https://nsubstitute.github.io/help/received-calls/)
* [Clearing received calls](https://nsubstitute.github.io/help/clear-received-calls/)
* [Argument matchers](https://nsubstitute.github.io/help/argument-matchers/)
* [Callbacks, void calls and When..Do](https://nsubstitute.github.io/help/callbacks/)
* [Throwing exceptions](https://nsubstitute.github.io/help/throwing-exceptions/)
* [Safe configuration and overlapping calls](https://nsubstitute.github.io/help/configure/)
* [Raising events](https://nsubstitute.github.io/help/raising-events/)
* [Auto and recursive mocks](https://nsubstitute.github.io/help/auto-and-recursive-mocks/)
* [Setting out and ref args](https://nsubstitute.github.io/help/setting-out-and-ref-arguments/)
* [Actions with argument matchers](https://nsubstitute.github.io/help/actions-with-arguments/)
* [Checking call order](https://nsubstitute.github.io/help/received-in-order/)
* [Partial subs and test spies](https://nsubstitute.github.io/help/partial-subs/)
* [Return for all calls of a type](https://nsubstitute.github.io/help/return-for-all/)
* [Threading](https://nsubstitute.github.io/help/threading/)
* [Compatibility argument matchers](https://nsubstitute.github.io/help/compat-args/)
* [NSubstitute.Analyzers](https://nsubstitute.github.io/help/nsubstitute-analysers/)

**Callbacks, void calls and When..Do**

Sometimes it is useful to execute some arbitrary code whenever a particular call is made. We have already seen an example of this when [passing functions to Returns()](https://nsubstitute.github.io/help/return-from-function/#callbacks).

|  |
| --- |
| var counter = 0;  calculator      .Add(0,0)      .ReturnsForAnyArgs(x => 0)      .AndDoes(x => counter++);    calculator.Add(7,3);  calculator.Add(2,2);  calculator.Add(11,-3);  Assert.AreEqual(counter, 3); |

The [Return from a function](https://nsubstitute.github.io/help/return-from-function) topic has more information on the arguments passed to the callback.

**Callbacks for void calls**

Returns() can be used to get callbacks for members that return a value, but for void members we need a different technique, because we can’t call a method on a void return. For these cases we can use the When..Do syntax.

**When called, do this**

When..Do uses two calls to configure our callback. First, When() is called on the substitute and passed a function. The argument to the function is the substitute itself, and we can call the member we are interested in here, even if it returns void. We then call Do() and pass in our callback that will be executed when the substitute’s member is called.

|  |
| --- |
| public interface IFoo {      void SayHello(string to);  }  [Test]  public void SayHello() {      var counter = 0;      var foo = Substitute.For<IFoo>();      foo.When(x => x.SayHello("World"))          .Do(x => counter++);        foo.SayHello("World");      foo.SayHello("World");      Assert.AreEqual(2, counter);  } |

The argument passed to the Do() method is the same call information passed to the [Returns() callback](https://nsubstitute.github.io/help/return-from-function), which gives us access to the arguments used for the call.

Note that we can also use When..Do syntax for non-void members, but generally the Returns() syntax is preferred for brevity and clarity. You may still find it useful for non-voids when you want to execute a function without changing a previous return value.

|  |
| --- |
| var counter = 0;  calculator.Add(1, 2).Returns(3);  calculator      .When(x => x.Add(Arg.Any<int>(), Arg.Any<int>()))      .Do(x => counter++);    var result = calculator.Add(1, 2);  Assert.AreEqual(3, result);  Assert.AreEqual(1, counter); |

**Per argument callbacks**

In cases where we only need callbacks for a particular argument we may be able to use [per argument callbacks like Arg.Do() and Arg.Invoke()](https://nsubstitute.github.io/help/actions-with-arguments) instead of When..Do.

Argument callbacks give us slightly more concise code in a style that is more in keeping with the rest of the NSubstitute API. See [Actions with arguments](https://nsubstitute.github.io/help/actions-with-arguments)for more information and examples.

**Callback builder for more complex callbacks**

The Callback builder lets us create more complex Do() scenarios. We can use Callback.First() followed by Then(), ThenThrow() and ThenKeepDoing() to build chains of callbacks. We can also use Always() and AlwaysThrow() to specify callbacks called every time. Note that a callback set by an Always() method will be called even if other callbacks will throw an exception.

|  |
| --- |
| var sub = Substitute.For<ISomething>();    var calls = new List<string>();  var counter = 0;    sub    .When(x => x.Something())    .Do(      Callback.First(x => calls.Add("1"))          .Then(x => calls.Add("2"))          .Then(x => calls.Add("3"))          .ThenKeepDoing(x => calls.Add("+"))          .AndAlways(x => counter++)    );    for (int i = 0; i < 5; i++)  {    sub.Something();  }  Assert.That(String.Concat(calls), Is.EqualTo("123++"));  Assert.That(counter, Is.EqualTo(5)); |

[« Argument matchers](https://nsubstitute.github.io/help/argument-matchers/)

[Throwing exceptions »](https://nsubstitute.github.io/help/throwing-exceptions/)

# Throwing exceptions

[Callbacks](https://nsubstitute.github.io/help/callbacks) can be used to throw exceptions when a member is called.

|  |
| --- |
| //For non-voids:  calculator.Add(-1, -1).Returns(x => { throw new Exception(); });    //For voids and non-voids:  calculator      .When(x => x.Add(-2, -2))      .Do(x => { throw new Exception(); });    //Both calls will now throw.  Assert.Throws<Exception>(() => calculator.Add(-1, -1));  Assert.Throws<Exception>(() => calculator.Add(-2, -2)); |

[« Callbacks, void calls and When..Do](https://nsubstitute.github.io/help/callbacks/)

[Safe configuration and overlapping calls »](https://nsubstitute.github.io/help/configure/)

# Safe configuration and overlapping calls

*Configure()* is supported in NSubstitute 4.0 and above.

Sometimes we want to configure a call that overlaps a more general call we have previously setup to run a [callback](https://nsubstitute.github.io/help/callbacks) or throw an [exception](https://nsubstitute.github.io/help/throwing-exceptions/). Ideally we would modify the setups so they don’t overlap, but we can also prevent these callbacks from running while we setup the next call by calling Configure()before invoking the method we want to re-configure. Configure() tells NSubstitute we are configuring the following call so that it will not run any callbacks from previous configurations.

|  |
| --- |
| calculator.Add(Arg.Any<int>(), Arg.Any<int>()).Returns(x => { throw new Exception(); });    // Now if we try to add a return value for a more specific returns this will throw  // before it gets a chance to configure the call:  //    calculator.Add(1, 2).Returns(3);    // Instead, we can use Configure to ensure the previous callback does not run:  calculator.Configure().Add(1, 2).Returns(3);    // Now both the exception callback and our other return have been configured:  Assert.AreEqual(3, calculator.Add(1, 2));  Assert.Throws<Exception>(() => calculator.Add(-2, -2)); |

NSubstitute will also assume we are configuring a call if we have an argument matcher in our call, such as Arg.Is(1) in calculator.Add(Arg.Is(1), 2).Returns(3), but it is generally better to be more explicit by using .Configure().

This can be particularly useful with [partial substitutes](https://nsubstitute.github.io/help/partial-subs/) to help avoid real code being executed when configuring calls that would otherwise call the base implementation.

[« Throwing exceptions](https://nsubstitute.github.io/help/throwing-exceptions/)

[Raising events »](https://nsubstitute.github.io/help/raising-events/)

# Raising events

Sometimes it is necessary to raise events declared on the types being substituted for. Consider the following example:

|  |
| --- |
| public interface IEngine {      event EventHandler Idling;      event EventHandler<LowFuelWarningEventArgs> LowFuelWarning;      event Action<int> RevvedAt;  }    public class LowFuelWarningEventArgs : EventArgs {      public int PercentLeft { get; private set; }      public LowFuelWarningEventArgs(int percentLeft) {          PercentLeft = percentLeft;      }  } |

Events are “interesting” creatures in the .NET world, as you can’t pass around references to them like you can with other members. Instead, you can only add or remove handlers to events, and it is this add handler syntax that NSubstitute uses to raise events.

|  |
| --- |
| var wasCalled = false;  engine.Idling += (sender, args) => wasCalled = true;    //Tell the substitute to raise the event with a sender and EventArgs:  engine.Idling += Raise.EventWith(new object(), new EventArgs());    Assert.True(wasCalled); |

In the example above we don’t really mind what sender and EventArgs our event is raised with, just that it was called. In this case NSubstitute can make our life easier by creating the required arguments for our event handler:

|  |
| --- |
| engine.Idling += Raise.Event();  Assert.True(wasCalled); |

## Raising events when arguments do not have a default constructor

NSubstitute will not always be able to create the event arguments for you. If your event args do not have a default constructor you will have to provide them yourself using Raise.EventWith<TEventArgs>(...), as is the case for the LowFuelWarning event. NSubstitute will still create the sender for you if you don’t provide it though.

|  |
| --- |
| engine.LowFuelWarning += (sender, args) => numberOfEvents++;    //Raise event with specific args, any sender:  engine.LowFuelWarning += Raise.EventWith(new LowFuelWarningEventArgs(10));  //Raise event with specific args and sender:  engine.LowFuelWarning += Raise.EventWith(new object(), new LowFuelWarningEventArgs(10));    Assert.AreEqual(2, numberOfEvents); |

## Raising Delegate events

Sometimes events are declared with a delegate that does not inherit from EventHandler<T> or EventHandler. These events can be raised using Raise.Event<TypeOfEventHandlerDelegate>(arguments). NSubsitute will try and guess the arguments required for the delegate, but if it can’t it will tell you what arguments you need to supply.

The following examples shows raising an INotifyPropertyChanged event, which uses a PropertyChangedEventHandler delegate and requires two parameters.

|  |
| --- |
| var sub = Substitute.For<INotifyPropertyChanged>();  bool wasCalled = false;  sub.PropertyChanged += (sender, args) => wasCalled = true;    sub.PropertyChanged += Raise.Event<PropertyChangedEventHandler>(this, new PropertyChangedEventArgs("test"));    Assert.That(wasCalled); |

## Raising Action events

In the IEngine example the RevvedAt event is declared as an Action<int>. This is another example of a delegate event, and we can use Raise.Event<Action<int>>() to raise our event.

|  |
| --- |
| int revvedAt = 0;;  engine.RevvedAt += rpm => revvedAt = rpm;    engine.RevvedAt += Raise.Event<Action<int>>(123);    Assert.AreEqual(123, revvedAt); |

[« Safe configuration and overlapping calls](https://nsubstitute.github.io/help/configure/)

[Auto and recursive mocks »](https://nsubstitute.github.io/help/auto-and-recursive-mocks/)

# Auto and recursive mocks

Once a substitute has been created some properties and methods will automatically return non-null values. For example, any properties or methods that return an interface, delegate, or purely virtual class\* will automatically return substitutes themselves. This is commonly referred to as recursive mocking, and can be useful because you can avoid explicitly creating each substitute, which means less code. Other types, like String and Array, will default to returning empty values rather than nulls.

\* Note: A pure virtual class is defined as one with all its public methods and properties defined as virtual or abstract and with a default, parameterless constructor defined as public or protected.

## Recursive mocks

Say we have the following types:

|  |
| --- |
| public interface INumberParser {      IEnumerable<int> Parse(string expression);  }  public interface INumberParserFactory {      INumberParser Create(char delimiter);  } |

We want to configure our INumberParserFactory to create a parser that will return a certain set of int for an expresion. We could manually create each substitute:

|  |
| --- |
| var factory = Substitute.For<INumberParserFactory>();  var parser = Substitute.For<INumberParser>();  factory.Create(',').Returns(parser);  parser.Parse("an expression").Returns(new[] {1,2,3});    Assert.AreEqual(      factory.Create(',').Parse("an expression"),      new[] {1,2,3}); |

Or we could use the fact that a substitute for type INumberParser will automatically be returned for INumberParserFactory.Create():

|  |
| --- |
| var factory = Substitute.For<INumberParserFactory>();  factory.Create(',').Parse("an expression").Returns(new[] {1,2,3});    Assert.AreEqual(      factory.Create(',').Parse("an expression"),      new[] {1,2,3}); |

Each time a recursively-subbed property or method is called with the same arguments it will return the same substitute. If a method is called with different arguments a new substitute will be returned.

|  |
| --- |
| var firstCall = factory.Create(',');  var secondCall = factory.Create(',');  var thirdCallWithDiffArg = factory.Create('x');    Assert.AreSame(firstCall, secondCall);  Assert.AreNotSame(firstCall, thirdCallWithDiffArg); |

Note: Recursive substitutes will not be created for non-purely virtual classes, as creating and using classes can have potentially unwanted side-effects. You’ll therefore need to create and return these explicitly.

### Substitute chains

It is not really an ideal practice, but when required we can also use recursive mocks to make it easier to set up chains of substitutes. For example:

|  |
| --- |
| public interface IContext {      IRequest CurrentRequest { get; }  }  public interface IRequest {      IIdentity Identity { get; }      IIdentity NewIdentity(string name);  }  public interface IIdentity {      string Name { get; }      string[] Roles();  } |

To get the identity of the CurrentRequest to return a certain name, we could manually create substitutes for IContext, IRequest, and IIdentity, and then use Returns() to chain these substitutes together. Instead we can use the substitutes automatically created for each property and method:

|  |
| --- |
| var context = Substitute.For<IContext>();  context.CurrentRequest.Identity.Name.Returns("My pet fish Eric");  Assert.AreEqual(      "My pet fish Eric",      context.CurrentRequest.Identity.Name); |

Here CurrentRequest is automatically going to return a substitute for IRequest, and the IRequest substitute will automatically return a substitute for IIdentity.

Note: Setting up long chains of substitutes this way is a code smell: we are breaking the [Law of Demeter](https://en.wikipedia.org/wiki/Law_of_Demeter), which says objects should only talk to their immediate neighbours, not reach into their neighbours’ neighbours. If you write your tests without recursive mocks this becomes quite obvious as the set up becomes quite complicated, so if you are going to use recursive mocking you’ll need to be extra vigilant to avoid this type of coupling.

## Auto values

Properties and methods returning types of String or Array will automatically get empty, non-null defaults. This can help avoid null reference exceptions in cases where you just need a reference returned but don’t care about its specific properties.

|  |
| --- |
| var identity = Substitute.For<IIdentity>();  Assert.AreEqual(String.Empty, identity.Name);  Assert.AreEqual(0, identity.Roles().Length); |

[« Raising events](https://nsubstitute.github.io/help/raising-events/)

[Setting out and ref args »](https://nsubstitute.github.io/help/setting-out-and-ref-arguments/)

# Setting out and ref args

Out and ref arguments can be set using a [Returns() callback](https://nsubstitute.github.io/help/return-from-function), or using [When..Do](https://nsubstitute.github.io/help/callbacks).

|  |
| --- |
| public interface ILookup {      bool TryLookup(string key, out string value);  } |

For the interface above we can configure the return value and set the output of the second argument like this:

|  |
| --- |
| //Arrange  var lookup = Substitute.For<ILookup>();  lookup      .TryLookup("hello", out Arg.Is(""))      .Returns(x => {          x[1] = "world!";          return true;      });    //Act  var value = "";  var result = lookup.TryLookup("hello", out value);    //Assert  Assert.True(result);  Assert.AreEqual(value, "world!"); |

## Matching after assignments

Be careful when using an argument matcher with a reference we also assign to. The assignment can cause previously matching arguments to stop matching.

|  |
| --- |
| var counter = 0;  var value = "";  var lookup = Substitute.For<ILookup>();  lookup      .TryLookup("hello", out Arg.Is(value)) // value is "", matcher will check for ""      .Returns(x => {          x[1] = "assigned"; // Assign to 2nd arg          counter++;         // Count this matching call          return true;      });    // value is "", this will match!  lookup.TryLookup("hello", out value);  // Call matches, counter is now 1:  Assert.AreEqual(1, counter);    // value is now "assigned" but arg matcher is still looking for "", will NOT match anymore!  lookup.TryLookup("hello", out value);  // Call does NOT match anymore, counter is still 1:  Assert.AreEqual(1, counter); |

[« Auto and recursive mocks](https://nsubstitute.github.io/help/auto-and-recursive-mocks/)

[Actions with argument matchers »](https://nsubstitute.github.io/help/actions-with-arguments/)

# Actions with argument matchers

In addition to [specifying calls](https://nsubstitute.github.io/help/argument-matchers), matchers can also be used to perform a specific action with an argument whenever a matching call is made to a substitute. This is a fairly rare requirement, but can make test setup a little simpler in some cases.

Warning: Once we start adding non-trivial behaviour to our substitutes we run the risk of over-specifying or tightly coupling our tests and code. It may be better to pick a different abstraction that better encapsulates this behaviour, or even use a real collaborator and switch to coarser grained tests for the class being tested.

## Invoking callbacks

Say our class under test needs to call a method on a dependency, and provide a callback so it can be notified when the dependent object has finished. We can use Arg.Invoke() to immediately invoke the callback argument as soon as the substitute is called.

Let’s look at a contrived example. Say we are testing an OrderPlacedCommand, which needs to use an IOrderProcessor to process the order, then raise and event using IEvents when this completes successfully.

|  |
| --- |
| public interface IOrderProcessor {      void ProcessOrder(int orderId, Action<bool> orderProcessed);  }    public class OrderPlacedCommand {      IOrderProcessor orderProcessor;      IEvents events;      public OrderPlacedCommand(IOrderProcessor orderProcessor, IEvents events) {          this.orderProcessor = orderProcessor;          this.events = events;      }      public void Execute(ICart cart) {          orderProcessor.ProcessOrder(              cart.OrderId,              wasOk => { if (wasOk) events.RaiseOrderProcessed(cart.OrderId); }          );      }  } |

In our test we can use Arg.Invoke to simulate the situation where the IOrderProcessor finishes processing the order and invokes the callback to tell the calling code it is finished.

|  |
| --- |
| [Test]  public void Placing\_order\_should\_raise\_order\_processed\_when\_processing\_is\_successful() {      //Arrange      var cart = Substitute.For<ICart>();      var events = Substitute.For<IEvents>();      var processor = Substitute.For<IOrderProcessor>();      cart.OrderId = 3;      //Arrange for processor to invoke the callback arg with `true` whenever processing order id 3      processor.ProcessOrder(3, Arg.Invoke(true));        //Act      var command = new OrderPlacedCommand(processor, events);      command.Execute(cart);        //Assert      events.Received().RaiseOrderProcessed(3);  } |

Here we setup the processor to invoke the callback whenever processing an order with id 3. We set it up to pass true to this callback using Arg.Invoke(true).

There are several overloads to Arg.Invoke that let us invoke callbacks with varying numbers and types of arguments. We can also invoke custom delegate types (those that are not just simple Action delegates) using Arg.InvokeDelegate.

## Performing actions with arguments

Sometimes we may not want to invoke a callback immediately. Or maybe we want to store all instances of a particular argument passed to a method. Or even just capture a single argument for inspection later. We can use Arg.Do for these purposes. Arg.Do calls the action we give it with the argument used for each matching call.

|  |
| --- |
| var argumentUsed = 0;  calculator.Multiply(Arg.Any<int>(), Arg.Do<int>(x => argumentUsed = x));    calculator.Multiply(123, 42);    Assert.AreEqual(42, argumentUsed); |

Here we specify that a call to Multiply with any first argument should pass the second argument and put it in the argumentUsed variable. This can be quite useful when we want to assert several properties on an argument (for types more complex than int that is).

|  |
| --- |
| var firstArgsBeingMultiplied = new List<int>();  calculator.Multiply(Arg.Do<int>(x => firstArgsBeingMultiplied.Add(x)), 10);    calculator.Multiply(2, 10);  calculator.Multiply(5, 10);  calculator.Multiply(7, 4567); //Will not match our Arg.Do as second arg is not 10    Assert.AreEqual(firstArgsBeingMultiplied, new[] { 2, 5 }); |

Here our Arg.Do takes whatever int is passed as the first argument to Multiply and adds it to a list whenever the second argument is 10.

## Argument actions and call specification

Argument actions act just like the [Arg.Any<T>() argument matcher](https://nsubstitute.github.io/help/argument-matchers) in that they specify a call where that argument is any type compatible with T (and so can be used for [setting return values](https://nsubstitute.github.io/help/return-for-args) and [checking received calls](https://nsubstitute.github.io/help/received-calls)). They just have the added element of interacting with a specific argument of any call that matches that specification.

|  |
| --- |
| var numberOfCallsWhereFirstArgIsLessThan0 = 0;  //Specify a call where the first arg is less than 0, and the second is any int.  //When this specification is met we'll increment a counter in the Arg.Do action for  //the second argument that was used for the call, and we'll also make it return 123.  calculator      .Multiply(          Arg.Is<int>(x => x < 0),          Arg.Do<int>(x => numberOfCallsWhereFirstArgIsLessThan0++)      ).Returns(123);    var results = new[] {      calculator.Multiply(-4, 3),      calculator.Multiply(-27, 88),      calculator.Multiply(-7, 8),      calculator.Multiply(123, 2) //First arg greater than 0, so spec won't be met.  };    Assert.AreEqual(3, numberOfCallsWhereFirstArgIsLessThan0); //3 of 4 calls have first arg < 0  Assert.AreEqual(results, new[] {123, 123, 123, 0}); //Last call returns 0, not 123 |

[« Setting out and ref args](https://nsubstitute.github.io/help/setting-out-and-ref-arguments/)

[Checking call order »](https://nsubstitute.github.io/help/received-in-order/)

# Checking call order

Sometimes calls need to be made in a specific order. Depending on the timing of calls like this is known as temporal coupling. Ideally we’d change our design to remove this coupling, but for times when we can’t NSubstitute lets us resort to asserting the order of calls.

|  |
| --- |
| [Test]  public void TestCommandRunWhileConnectionIsOpen() {    var connection = Substitute.For<IConnection>();    var command = Substitute.For<ICommand>();    var subject = new Controller(connection, command);      subject.DoStuff();      Received.InOrder(() => {      connection.Open();      command.Run(connection);      connection.Close();    });  } |

If the calls were received in a different order then Received.InOrder will throw an exception and show the expected and actual calls.

We can use standard [argument matchers](https://nsubstitute.github.io/help/argument-matchers/) to match calls, just as we would when [checking for a single received call](https://nsubstitute.github.io/help/received-calls/).

|  |
| --- |
| [Test]  public void SubscribeToEventBeforeOpeningConnection() {    var connection = Substitute.For<IConnection>();    connection.SomethingHappened += () => { /\* some event handler \*/ };    connection.Open();      Received.InOrder(() => {      connection.SomethingHappened += Arg.Any<Action>();      connection.Open();    });  } |

[« Actions with argument matchers](https://nsubstitute.github.io/help/actions-with-arguments/)

[Partial subs and test spies »](https://nsubstitute.github.io/help/partial-subs/)

# Partial subs and test spies

Partial substitutes allow us to create an object that acts like a real instance of a class, and selectively substitute for specific parts of that object. This is useful for when we need a substitute to have real behaviour except for a single method that we want to replace, or when we just want to spy on what calls are being made.

**WARNING:** Partial substitutes will be calling your class’ **real code by default**, so if you are not careful it is possible for this code to run **even while you are configuring specific methods to be substituted**! For this reason partial substitutes are not generally recommended, so avoid them where possible (especially if your code deletes files, contacts payment gateways, or initiates underground lair self-destruct routines). In some cases they can be quite handy though; just be sure to handle with care.

## Replacing a single method

In this example we want to test the Read() method logic without running ReadFile().

|  |
| --- |
| public class SummingReader {    public virtual int Read(string path) {      var s = ReadFile(path);      return s.Split(',').Select(int.Parse).Sum();    }    public virtual string ReadFile(string path) { return "the result of reading the file here"; }  } |

By default ReadFile may access a file on the file system, but we can Substitute.ForPartsOf<SummingReader>() and override ReadFile to return a substitute value, rather than loading data from a real file, using Returns:

|  |
| --- |
| [Test]  public void ShouldSumAllNumbersInFile() {    var reader = Substitute.ForPartsOf<SummingReader>();    reader.Configure().ReadFile("foo.txt").Returns("1,2,3,4,5"); // CAUTION: real code warning!      var result = reader.Read("foo.txt");      Assert.That(result, Is.EqualTo(15));  } |

Now the real Read method will execute, but ReadFile will return our substituted value instead of calling the original method, so we can run the test without having to worry about a real file system.

Note the **CAUTION** comment. If we had not used [Configure()](https://nsubstitute.github.io/help/configure/) here before .ReadFile() then the real ReadFile method would have executed before we had a chance to override the behaviour (reader.ReadFile("foo.txt") returns first before .Returns(...) executes). In some cases this may not be a problem, but if in doubt make sure you call Configure() first so NSubstitute knows you are configuring a call and don’t want to run any real code. (This still does not guarantee real code will not run – remember, NSubstitute will not prevent non-virtual calls from executing.)

The *Configure()* method is only available in NSubstitute 4.0 and above. For verisons prior to 4.0 we need to use *When .. DoNotCallBase* described below.

## Void methods and DoNotCallBase

We can’t use .Returns() with void methods, but we can stop a void method on a partial substitute from calling the real method using When .. DoNotCallBase. (This also works for non-void methods, although generally we use Configure() and Returns() to override the base behaviour in these cases.)

|  |
| --- |
| public class EmailServer {    public virtual void Send(string to, string from, string message) {      // Insert real email sending code here      throw new NotImplementedException();    }      public virtual void SendMultiple(IEnumerable<string> recipients, string from, string message) {      foreach (var recipient in recipients) {          Send(recipient, from, message);      }    }  }    [Test]  public void ShouldSendMultipleEmails() {    var server = Substitute.ForPartsOf<EmailServer>();    server.WhenForAnyArgs(x => x.Send("", "", "")).DoNotCallBase(); // Make sure Send won't call real implementation      server.SendMultiple(      new [] { "alice", "bob", "charlie" },      "nsubstitute",      "Partial subs should be used with caution."); // This won't run the real Send now, thanks to DoNotCallBase().      server.Received().Send("alice", "nsubstitute", Arg.Any<string>());    server.Received().Send("bob", "nsubstitute", Arg.Any<string>());    server.Received().Send("charlie", "nsubstitute", Arg.Any<string>());  } |

## Test spies

Even without substituting for specific parts of a class, the instance returned by Substitute.ForPartsOf<T> records all calls made to virtual members, so we can [check Received() calls](https://nsubstitute.github.io/help/received-calls/) made to any partial substitute.

[« Checking call order](https://nsubstitute.github.io/help/received-in-order/)

[Return for all calls of a type »](https://nsubstitute.github.io/help/return-for-all/)

# Return for all calls of a type

We can return a specific value for all calls to a substitute using sub.ReturnsForAll<T>(T value). This will cause sub to return value for all calls that return something of type T and are [not already stubbed](https://nsubstitute.github.io/help/return-for-all/#returns_vs_returnsforall).

**Note: we need using NSubstitute.Extensions to import the .ReturnsForAll<T>() extension method.**

The type must match exactly: .ReturnsForAll<Cat>(cat) will not set a return value for a call that returns Animal, even if Cat inherits from Animal. To return for the super-type, use .ReturnsForAll<Animal>(cat). (If you’d like a change in this behaviour, please [let us know](https://groups.google.com/forum/#!forum/nsubstitute)).

There is also an overload that takes a Func<CallInfo,T> so the value to return will be calculated each time.

## Fluent interface example

One example of where this can be useful is a builder-style interface where each call returns a reference to itself.

|  |
| --- |
| // using NSubstitute.Extensions;    public interface IWidgetBuilder {    IWidgetBuilder Quantity(int i);    IWidgetBuilder AddLineItem(string s);    IWidgetContainer GetWidgets();  }    public class ProductionLine {    IWidgetBuilder builder;    public ProductionLine(IWidgetBuilder builder) {      this.builder = builder;    }      public IWidgetContainer Run() {      return builder                .Quantity(2)                .AddLineItem("Thingoe")                .AddLineItem("Other thingoe")                .GetWidgets();    }  }    [Test]  public void ShouldReturnWidgetsFromBuilder() {    var builder = Substitute.For<IWidgetBuilder>();    builder.ReturnsForAll<IWidgetBuilder>(builder);    var line = new ProductionLine(builder);      var result = line.Run();      Assert.That(result, Is.EqualTo(builder.GetWidgets()));  } |

In this test builder will return a reference to itself whenever a call returns a value of type IWidgetBuilder, so the chained calls will all work on the same builder instance.

## Returns vs. ReturnsForAll

Calls will only use .ReturnsForAll values when NSubstitute can’t find an explicitly stubbed value for the call. So if we set sub.GetWidget().Returns(widget), that will take precendence over any values stubbed by sub.ReturnsForAll<Widget>(otherWidget).

|  |
| --- |
| [Test]  public void ReturnsTakesPrecedence() {    var widget = new Widget();    var otherWidget = new Widget();    var sub = Substitute.For<IWidgetExample>();    sub.GetWidget(1).Returns(widget);    sub.ReturnsForAll<Widget>(otherWidget);      Assert.That(sub.GetWidget(1), Is.SameAs(widget));    Assert.That(sub.GetWidget(42), Is.SameAs(otherWidget));  } |

[« Partial subs and test spies](https://nsubstitute.github.io/help/partial-subs/)

[Threading »](https://nsubstitute.github.io/help/threading/)

# Threading

It is fairly standard for production code to call a substitute from multiple threads, but we should avoid having our test code configure or assert on a substitute while it is also be used from other threads in production code. See [Issue #256](https://github.com/nsubstitute/NSubstitute/issues/256) for an example of how doing this can result in problems.

To avoid this sort of problem, make sure your test has finished configuring its substitutes before exercising the production code, then make sure the production code has completed before your test asserts on Received() calls.

[« Return for all calls of a type](https://nsubstitute.github.io/help/return-for-all/)

[Compatibility argument matchers »](https://nsubstitute.github.io/help/compat-args/)

# Compatibility argument matchers

NSubstitute [argument matchers](https://nsubstitute.github.io/help/argument-matchers) depend on having C# 7.0 or later (as of NSubstitute 4.0). This lets them be used with out and ref parameters, but it also means that if you are stuck on an earlier version of C# you may get an error like the following when trying to use a matcher like Arg.Is(123):

CS7085: By-reference return type 'ref T' is not supported.

If you have C# 7.0-compatible tooling installed you can [set <LangVersion /> in your test csproj file](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version) to 7 or higher, or to latest or default.

Stuck with pre-7.0 tooling? Then use Arg.Compat instead of Arg, or use CompatArg in the NSubstitute.Compatibility namespace. Arg.Compat will work everywhere Arg does, with the exception of matching out and ref args. For example, if the documentation mentions Arg.Is(42), you can instead use Arg.Compat.Is(42). CompatArg is a bit trickier to setup, but may make migrating between Arg and Arg.Compat easier in some cases. Both options are described below.

## Using Arg.Compat

The simplest work-around if you are stuck on pre-C#7 is to use Arg.Compat. wherever you would normally use Arg.. To migrate existing code, replace all instances of Arg. with Arg.Compat..

|  |
| --- |
| calculator.Add(1, -10);    // Instead of `Arg.Is<int>(x => x < 0)`, use:  calculator.Received().Add(1, Arg.Compat.Is<int>(x => x < 0));    // Instead of `Arg.Any<int>()`, use:  calculator      .Received()      .Add(1, Arg.Compat.Any<int>());    // Same for Returns and DidNotReceive:  calculator.Add(Arg.Compat.Any<int>(), Arg.Compat.Is(42)).Returns(123);  calculator.DidNotReceive().Add(Arg.Compat.Is<int>(x => x > 10), -10); |

## Using NSubstitute.Compatibility.CompatArg

If you have a project with lots of existing arg matchers then migrating to Arg.Compat can require a lot of code changes. A smaller change is to instead use an instance of the CompatArg class in the NSubstitute.Compatibility namespace. This approach may also make it easier to upgrade to the newer Argmatchers in future.

|  |
| --- |
| [TestFixture]  public class SampleCompatArgFixture {        // Declare Arg field. Any existing `Arg` references will now go via `CompatArg`, instead      // of the new `Arg` type that is incompatible with older C# compilers.      private static readonly NSubstitute.Compatibility.CompatArg Arg = NSubstitute.Compatibility.CompatArg.Instance;        [Test]      public void DemonstrationOfCompatArgs() {          var calculator = Substitute.For<ICalculator>();            calculator.Add(1, -10);            // Arg.Is will now go via CompatArg. It is equivalent to Arg.Compat.Is.          calculator.Received().Add(1, Arg.Is<int>(x => x < 0));      }  } |

This works particularly well if a common test base class is used.

|  |
| --- |
| public class BaseTestFixture {        // Declare Arg field. Any existing `Arg` references will now go via `CompatArg`, instead      // of the new `Arg` type that is incompatible with older C# compilers.      protected static readonly NSubstitute.Compatibility.CompatArg Arg = NSubstitute.Compatibility.CompatArg.Instance;    } |

If you are later able to update the C# compiler your project is using, you can remove the CompatArg field and all Arg references will go through standard arg matchers (and you’ll now be able to use them with out and ref parameters!).

## Argument matchers in expression trees

As of NSubstitute 4.0 argument matchers can no longer be used in expression trees. Doing so will cause a compilation error:

CS8153: An expression tree lambda may not contain a call to a method, property, or indexer that returns by reference

The Arg.Compat matchers can be used to work around this issue. They do not return by reference so they are fine to use in expression trees.

|  |
| --- |
| public interface IFoo { void DoStuff(int i); }    [Test]  public void Sample() {      var sub = Substitute.For<IFoo>();      sub.DoStuff(42);      // If `specify` takes an `Expression<Action>` argument, this will fail with CS8153:      //   specify(() => sub.Received().DoStuff(Arg.Any<int>()));        // Instead use compat matcher:      specify(() => sub.Received().DoStuff(Arg.Compat.Any<int>()));        // Or re-write without expression tree if possible:      sub.Received().DoStuff(Arg.Any<int>());  } |

[« Threading](https://nsubstitute.github.io/help/threading/)

[NSubstitute.Analyzers »](https://nsubstitute.github.io/help/nsubstitute-analysers/)

**NSubstitute.Analyzers**

The [NSubstitute.Analyzers](https://github.com/nsubstitute/NSubstitute.Analyzers) project uses [Roslyn](https://docs.microsoft.com/en-us/visualstudio/extensibility/dotnet-compiler-platform-roslyn-extensibility) to add code analysis during compilation to detect possible errors using the NSubstitute API.

For example: as noted in the warning in [Creating a substitute](https://nsubstitute.github.io/help/creating-a-substitute/), non-virtual members of a class can not be intercepted by NSubstitute. This can cause our tests to behave unpredictably if we attempt to use .Returns() or .Received() with them. NSubstitute.Analyzers can detect attempts to substitute for non-virtual members and raise an [NS1000](https://github.com/nsubstitute/NSubstitute.Analyzers/blob/dev/documentation/rules/NS1000.md) warning at compile-time, including suggestions on how to fix it. It’s much nicer to find these problems at compile-time than during test execution, or worse, not finding them until much later when we realise our tests were not doing exactly what we thought they were!

The NSubstitute NuGet package will work fine without any additional packages, but we highly recommend also adding the appropriate Analyzers package to your project (C# or VB):

* [NSubstitute.Analyzers.CSharp](https://www.nuget.org/packages/NSubstitute.Analyzers.CSharp/) for C# projects
* [NSubstitute.Analyzers.VisualBasic](https://www.nuget.org/packages/NSubstitute.Analyzers.VisualBasic/) for VB projects

**Diagnostics**

See [NSubstitute.Analyzers documentation](https://github.com/nsubstitute/NSubstitute.Analyzers/tree/dev/documentation) for a list of the potential issues NSubstitute.Analyzers can help detect.

**Spelling**

The Australian members of our team would like to point out they would have preferred to use the English spelling NSubstitute.Analysers for this project, but have begrudgingly agreed to use the American (mis)spelling instead. :)

[« Compatibility argument matchers](https://nsubstitute.github.io/help/compat-args/)