Moq is intended to be simple to use, strongly typed (no magic strings!, and therefore full compiler-verified and refactoring-friendly) and minimalistic (while still fully functional!).

# Outline



## Methods

```
using Moq;

// Assumptions:

public interface IFoo
{
    Bar Bar { get; set; }
    string Name { get; set; }
    int Value { get; set; }
    bool DoSomething(string value);
    bool DoSomething(int number, string value);
    Task<bool> DoSomethingAsync();
    string DoSomethingStringy(string value);
    bool TryParse(string value, out string outputValue);
    bool Submit(ref Bar bar);
    int GetCount();
    bool Add(int value);
}

public class Bar
{
    public virtual Baz Baz { get; set; }
```

```
    public virtual bool Submit() { return false; }
}

public class Baz
{
    public virtual string Name { get; set; }
}


var mock = new Mock<IFoo>();
mock.Setup(foo => foo.DoSomething("ping")).Returns(true);


// out arguments
var outString = "ack";
// TryParse will return true, and the out argument will return "ack",
lazy evaluated
mock.Setup(foo => foo.TryParse("ping", out outString)).Returns(true);


// ref arguments
var instance = new Bar();
// Only matches if the ref argument to the invocation is the same
instance
mock.Setup(foo => foo.Submit(ref instance)).Returns(true);


// access invocation arguments when returning a value
mock.Setup(x => x.DoSomethingStringy(It.IsAny<string>()))
                .Returns((string s) => s.ToLower());
// Multiple parameters overloads available


// throwing when invoked with specific parameters
mock.Setup(foo =>
foo.DoSomething("reset")).Throws<InvalidOperationException>();
mock.Setup(foo => foo.DoSomething("")).Throws(new
ArgumentException("command"));


// lazy evaluating return value
var count = 1;
mock.Setup(foo => foo.GetCount()).Returns(() => count);


// async methods (see below for more about async):
mock.Setup(foo => foo.DoSomethingAsync().Result).Returns(true);
```

## Async Methods

There are several ways to set up "async" methods (e.g. methods returning a `Task<T>` or `ValueTask<T>`):

- Starting with Moq 4.16, you can simply `mock.Setup` the returned task's `.Result` property. This works in nearly all setup and verification expressions:

```
mock.Setup(foo => foo.DoSomethingAsync().Result).Returns(true);
```

- In earlier versions, use setup helper methods like `setup.ReturnsAsync`, `setup.ThrowsAsync` where they are available:

```
mock.Setup(foo => foo.DoSomethingAsync()).ReturnsAsync(true);
```

- (You could also do the following, but that will trigger a compiler warning about synchronous execution of the async lambda:)

```
mock.Setup(foo => foo.DoSomethingAsync()).Returns(async () => 42);
```

## Matching Arguments

```
// any value
mock.Setup(foo => foo.DoSomething(It.IsAny<string>())).Returns(true);


// any value passed in a `ref` parameter (requires Moq 4.8 or later):
mock.Setup(foo => foo.Submit(ref It.Ref<Bar>.IsAny)).Returns(true);


// matching Func<int>, lazy evaluated
mock.Setup(foo => foo.Add(It.Is<int>(i => i % 2 == 0))).Returns(true);


// matching ranges
mock.Setup(foo => foo.Add(It.IsInRange<int>(0, 10,
Range.Inclusive))).Returns(true);


// matching regex
mock.Setup(x => x.DoSomethingStringy(It.IsRegex("[a-d]+",
RegexOptions.IgnoreCase))).Returns("foo");
```

## Properties

```
mock.Setup(foo => foo.Name).Returns("bar");


// auto-mocking hierarchies (a.k.a. recursive mocks)
mock.Setup(foo => foo.Bar.Baz.Name).Returns("baz");

// expects an invocation to set the value to "foo"
mock.SetupSet(foo => foo.Name = "foo");

// or verify the setter directly
mock.VerifySet(foo => foo.Name = "foo");
```

- Setup a property so that it will automatically start tracking its value (also known as Stub):

```
// start "tracking" sets/gets to this property
mock.SetupProperty(f => f.Name);

// alternatively, provide a default value for the stubbed property
mock.SetupProperty(f => f.Name, "foo");


// Now you can do:

IFoo foo = mock.Object;
// Initial value was stored
Assert.Equal("foo", foo.Name);

// New value set which changes the initial value
foo.Name = "bar";
Assert.Equal("bar", foo.Name);
```

- Stub all properties on a mock:

```
mock.SetupAllProperties();
```

# Events

```
// Setting up an event's `add` and `remove` accessors (requires Moq 4.13
or later):
mock.SetupAdd(m => m.FooEvent += It.IsAny<EventHandler>())...;
mock.SetupRemove(m => m.FooEvent -= It.IsAny<EventHandler>())...;

// Raising an event on the mock
mock.Raise(m => m.FooEvent += null, new FooEventArgs(fooValue));

// Raising an event on the mock that has sender in handler parameters
```

```
mock.Raise(m => m.FooEvent += null, this, new FooEventArgs(fooValue));

// Raising an event on a descendant down the hierarchy
mock.Raise(m => m.Child.First.FooEvent += null, new
FooEventArgs(fooValue));

// Causing an event to raise automatically when Submit is invoked
mock.Setup(foo => foo.Submit()).Raises(f => f.Sent += null,
EventArgs.Empty);
// The raised event would trigger behavior on the object under test,
which
// you would make assertions about later (how its state changed as a
consequence, typically)

// Raising a custom event which does not adhere to the EventHandler
pattern
public delegate void MyEventHandler(int i, bool b);
public interface IFoo
{
  event MyEventHandler MyEvent;
}

var mock = new Mock<IFoo>();
...
// Raise passing the custom arguments expected by the event delegate
mock.Raise(foo => foo.MyEvent += null, 25, true);
```

## Callbacks

```
var mock = new Mock<IFoo>();
var calls = 0;
var callArgs = new List<string>();

mock.Setup(foo => foo.DoSomething("ping"))
    .Callback(() => calls++)
    .Returns(true);

// access invocation arguments
mock.Setup(foo => foo.DoSomething(It.IsAny<string>()))
    .Callback((string s) => callArgs.Add(s))
    .Returns(true);

// alternate equivalent generic method syntax
mock.Setup(foo => foo.DoSomething(It.IsAny<string>()))
    .Callback<string>(s => callArgs.Add(s))
    .Returns(true);

// access arguments for methods with multiple parameters
mock.Setup(foo => foo.DoSomething(It.IsAny<int>(), It.IsAny<string>()))
```

```
    .Callback<int, string>((i, s) => callArgs.Add(s))
    .Returns(true);

// callbacks can be specified before and after invocation
mock.Setup(foo => foo.DoSomething("ping"))
    .Callback(() => Console.WriteLine("Before returns"))
    .Returns(true)
    .Callback(() => Console.WriteLine("After returns"));

// callbacks for methods with `ref` / `out` parameters are possible but
require some work (and Moq 4.8 or later):
delegate void SubmitCallback(ref Bar bar);

mock.Setup(foo => foo.Submit(ref It.Ref<Bar>.IsAny))
    .Callback(new SubmitCallback((ref Bar bar) =>
Console.WriteLine("Submitting a Bar!")));

// returning different values on each invocation
var mock = new Mock<IFoo>();
var calls = 0;
mock.Setup(foo => foo.GetCount())
        .Callback(() => calls++)
        .Returns(() => calls);
// returns 0 on first invocation, 1 on the next, and so on
Console.WriteLine(mock.Object.GetCount());

// access invocation arguments and set to mock setup property
mock.SetupProperty(foo=>foo.Bar);
mock.Setup(foo => foo.DoSomething(It.IsAny<string>()))
    .Callback((string s) => mock.Object.Bar=s)
    .Returns(true);
```

## Verification

```
mock.Verify(foo => foo.DoSomething("ping"));

// Verify with custom error message for failure
mock.Verify(foo => foo.DoSomething("ping"), "When doing operation X, the
service should be pinged always");

// Method should never be called
mock.Verify(foo => foo.DoSomething("ping"), Times.Never());

// Called at least once
mock.Verify(foo => foo.DoSomething("ping"), Times.AtLeastOnce());

// Verify getter invocation, regardless of value.
mock.VerifyGet(foo => foo.Name);
```

```
// Verify setter invocation, regardless of value.
mock.VerifySet(foo => foo.Name);

// Verify setter called with specific value
mock.VerifySet(foo => foo.Name ="foo");

// Verify setter with an argument matcher
mock.VerifySet(foo => foo.Value = It.IsInRange(1, 5, Range.Inclusive));

// Verify event accessors (requires Moq 4.13 or later):
mock.VerifyAdd(foo => foo.FooEvent += It.IsAny<EventHandler>());
mock.VerifyRemove(foo => foo.FooEvent -= It.IsAny<EventHandler>());

// Verify that no other invocations were made other than those already
verified (requires Moq 4.8 or later)
mock.VerifyNoOtherCalls();
```

## Customizing Mock Behavior

- Make mock behave like a "true Mock", raising exceptions for anything that doesn't have a corresponding expectation: in Moq slang a "Strict" mock; default behavior is "Loose" mock, which never throws and returns default values or empty arrays, enumerables, etc. if no expectation is set for a member

```
var mock = new Mock<IFoo>(MockBehavior.Strict);
```

- Invoke base class implementation if no expectation overrides the member (a.k.a. "Partial Mocks" in Rhino Mocks): default is false. (*this is required if you are mocking Web/Html controls in System.Web!*)

```
var mock = new Mock<IFoo> { CallBase = true };
```

- Make an automatic recursive mock: a mock that will return a new mock for every member that doesn't have an expectation and whose return value can be mocked (i.e. it is not a value type)

```
var mock = new Mock<IFoo> { DefaultValue = DefaultValue.Mock };
// default is DefaultValue.Empty

// this property access would return a new mock of Bar as it's
"mock-able"
Bar value = mock.Object.Bar;

// the returned mock is reused, so further accesses to the property
```

```
return
// the same mock instance. this allows us to also use this instance
to
// set further expectations on it if we want
var barMock = Mock.Get(value);
barMock.Setup(b => b.Submit()).Returns(true);
```

- Centralizing mock instance creation and management: you can create and verify all mocks in a single place by using a MockRepository, which allows setting the MockBehavior, its CallBase and DefaultValue consistently

```
var repository = new MockRepository(MockBehavior.Strict) {
DefaultValue = DefaultValue.Mock };

// Create a mock using the repository settings
var fooMock = repository.Create<IFoo>();

// Create a mock overriding the repository settings
var barMock = repository.Create<Bar>(MockBehavior.Loose);

// Verify all verifiable expectations on all mocks created through
the repository
repository.Verify();
```

## Miscellaneous

- Setting up a member to return different values / throw exceptions on sequential calls:

```
var mock = new Mock<IFoo>();
mock.SetupSequence(f => f.GetCount())
    .Returns(3)  // will be returned on 1st invocation
    .Returns(2)  // will be returned on 2nd invocation
    .Returns(1)  // will be returned on 3rd invocation
    .Returns(0)  // will be returned on 4th invocation
    .Throws(new InvalidOperationException());  // will be thrown on
5th invocation
```

- Setting up a member to verify that a set of calls happened in sequence:

```
var _fooService = new Mock<IFoo>(MockBehavior.Strict);
var _barService = new Mock<IBar>(MockBehavior.Strict);
var _bazService = new Mock<IBaz>(MockBehavior.Strict);
var sequence = new MockSequence();
```

```
_fooService.InSequence(sequence).Setup(x ⇒
x.FooMethod(a)).ReturnsAsync(b);
_barService.InSequence(sequence).Setup(x ⇒
x.BarMethod(c)).ReturnsAsync(d);
_bazService.InSequence(sequence).Setup(x ⇒
x.BazMethod(e)).ReturnsAsync(f);

// In your test, if the system under test calls fooService then
barService
// then bazService's methods, the test will pass.
// Otherwise if it calls them in a different order than they were
set up,
// you will get a test failure specifying that it was missing a
corresponding setup
```

- Setting expectations for protected members (you can't get IntelliSense for these, so you access them using the member name as a string).
  Assuming the following class with a protected function should be mocked:

```
public class CommandBase {
    protected virtual int Execute();            // (1)
    protected virtual bool Execute(string arg); // (2)
}
```

```
// at the top of the test fixture
using Moq.Protected;

// In the test, mocking the `int Execute()` method (1)
var mock = new Mock<CommandBase>();
mock.Protected()
     .Setup<int>("Execute")
     .Returns(5);

// If you need argument matching, you MUST use ItExpr rather than It
// planning on improving this for vNext (see below for an
alternative in Moq 4.8)
// Mocking the `bool Execute(string arg)` method (2)
mock.Protected()
    .Setup<bool>("Execute",
        ItExpr.IsAny<string>())
    .Returns(true);
```

- Moq 4.8 and later allows you to set up protected members through a completely unrelated type that has the same members and thus provides the type information necessary for IntelliSense to work. Pickin up the example from the bullet point above,

you can also use this interface to set up protected generic methods and those having by-ref parameters:

```
// Completely unrelated Interface (CommandBase is not derived from
it) only created for the test.
// It contains a method with an identical method signature to the
protected method in the actual class which should be mocked
interface CommandBaseProtectedMembers
{
    bool Execute(string arg);
}

mock.Protected().As<CommandBaseProtectedMembers>()
    .Setup(m => m.Execute(It.IsAny<string>()))  // will set up
CommandBase.Execute
    .Returns(true);
```

## Advanced Features

```
// get mock from a mocked instance
IFoo foo = // get mock instance somehow
var fooMock = Mock.Get(foo);
fooMock.Setup(f => f.GetCount()).Returns(42);


// implementing multiple interfaces in mock
var mock = new Mock<IFoo>();
var disposableFoo = mock.As<IDisposable>();
// now the IFoo mock also implements IDisposable :)
disposableFoo.Setup(disposable => disposable.Dispose());

// implementing multiple interfaces in single mock
var mock = new Mock<IFoo>();
mock.Setup(foo => foo.Name).Returns("Fred");
mock.As<IDisposable>().Setup(disposable => disposable.Dispose());


// custom matchers
mock.Setup(foo => foo.DoSomething(IsLarge())).Throws<ArgumentException>
();
...
[Matcher]
public string IsLarge()
{
  return Match.Create<string>(s => !String.IsNullOrEmpty(s) && s.Length
> 100);
}
```

- Mocking internal types: Add either of the following custom attributes (typically in `AssemblyInfo.cs`) to the project containing the internal types — which one you need depends on whether your own project is strong-named or not:

```
// This assembly is the default dynamic assembly generated by Castle
DynamicProxy,
// used by Moq. If your assembly is strong-named, paste the
following in a single line:
[assembly:InternalsVisibleTo("DynamicProxyGenAssembly2,PublicKey=002
400000480000094000000060200000024000052534131000400000100010 0c547cac
37abd99c8db225ef2f6c8a3602f3b3606cc9891605d02baa56104f4cfc0734aa39b9
3bf7852f7d9266654753cc297e7d2edfe0bac1cdcf9f717241550e0a7b191195b766
7bb4f64bcb8e2121380fd1d9d46ad2d92d2d15605093924cceaf74c4861eff62abf6
9b9291ed0a340e113be11e6a7d3113e92484cf7045cc7")]

// Or, if your own assembly is not strong-named, omit the public
key:
[assembly:InternalsVisibleTo("DynamicProxyGenAssembly2")]
```

- Starting in Moq 4.8, you can create your own custom default value generation strategies (besides `DefaultValue.Empty` and `DefaultValue.Mock`) by subclassing `DefaultValueProvider`, or, if you want some more convenience, `LookupOrFallbackDefaultValueProvider`:

```
class MyEmptyDefaultValueProvider :
LookupOrFallbackDefaultValueProvider
{
    public MyEmptyDefaultValueProvider()
    {
        base.Register(typeof(string), (type, mock) ⇒ "?");
        base.Register(typeof(List<>), (type, mock) ⇒
Activator.CreateInstance(type));
    }
}

var mock = new Mock<IFoo> { DefaultValueProvider = new
MyEmptyDefaultValueProvider() };
var name = mock.Object.Name;  // ⇒ "?"
```

---

**Note**: When you pass the mock for consumption, you must pass `mock.Object`, not `mock` itself.

## Matching Generic Type Arguments

```
public interface IFoo
{
    bool M1<T>();
    bool M2<T>(T arg);
}

var mock = new Mock<IFoo>();
```

Generic arguments are matched using the usual type polymorphism rules, so if you want to match any type, you can simply use `object` as type argument in many cases:

```
mock.Setup(m => m.M1<object>()).Returns(true);
```

### Type matchers (Moq 4.13+)

In some situations, that might not work. Starting with Moq 4.13, you can use explicit type matchers, e.g. `It.IsAnyType` which is essentially a placeholder for a type (just like `It.IsAny<T>()` is a placeholder for any value):

```
// matches any type argument:
mock.Setup(m => m.M1<It.IsAnyType>()).Returns(true);

// matches only type arguments that are subtypes of / implement T:
mock.Setup(m => m.M1<It.IsSubtype<T>>()).Returns(true);

// use of type matchers is allowed in the argument list:
mock.Setup(m => m.M2(It.IsAny<It.IsAnyType>())).Returns(true);
mock.Setup(m => m.M2(It.IsAny<It.IsSubtype<T>>())).Returns(true);
```

## LINQ to Mocks

Moq is the one and only mocking framework that allows specifying mock behavior via declarative specification queries. You can think of LINQ to Mocks as:

> from the universe of mocks, get me one/those that behave like this (by Fernando Simonazzi)

Keep that query form in mind when reading the specifications:

```
var services = Mock.Of<IServiceProvider>(sp =>
    sp.GetService(typeof(IRepository)) == Mock.Of<IRepository>(r =>
r.IsAuthenticated == true) &&
    sp.GetService(typeof(IAuthentication)) == Mock.Of<IAuthentication>(a
=> a.AuthenticationType == "OAuth"));

// Multiple setups on a single mock and its recursive mocks
ControllerContext context = Mock.Of<ControllerContext>(ctx =>
```

```
      ctx.HttpContext.User.Identity.Name == "kzu" &&
      ctx.HttpContext.Request.IsAuthenticated == true &&
      ctx.HttpContext.Request.Url == new Uri("http://moqthis.com") &&
      ctx.HttpContext.Response.ContentType == "application/xml");

// Setting up multiple chained mocks:
var context = Mock.Of<ControllerContext>(ctx =>
      ctx.HttpContext.Request.Url == new Uri("http://moqthis.me") &&
      ctx.HttpContext.Response.ContentType == "application/xml" &&
      // Chained mock specification
      ctx.HttpContext.GetSection("server") == Mock.Of<ServerSection>
(config =>
          config.Server.ServerUrl == new Uri("http://moqthis.com/api")));
```

LINQ to Mocks is great for quickly stubbing out dependencies that typically don't need further verification. If you do need to verify later some invocation on those mocks, you can easily retrieve them with `Mock.Get(instance)`.