# Quick guide to Moq

## 1 - About

### 1.1 - What is it?

The author says: "**Moq (pronounced "Mock-you" or just "Mock") is the only mocking library for .NET developed from scratch to take full advantage of .NET 3.5 (i.e. Linq expression trees) and C# 3.0 features (i.e. lambda expressions) that make it the most productive, type-safe and refactoring-friendly mocking library available. And it supports mocking interfaces as well as classes. Its API is extremely simple and straightforward, and doesn't require any prior knowledge or experience with mocking concepts.**"

In brief, Moq is intended to be:

- Simple to learn and use
- Strongly-typed (no magic strings, and therefore fully compiler-verified and refactoring-friendly!)
- Minimalistic (while still fully functional!)

### 1.2 - Licensing

New BSD License (http://www.opensource.org/licenses/bsd-license.php)

### 1.3 - Homepage

http://code.google.com/p/moq/

### 1.4 - Contributors

There are many developers to maintain this project (http://code.google.com/p/moq/people/list). You can join them if you like (http://code.google.com/p/moq/wiki/HowToContribute).

## 2 - Moq basics

### 2.1 - Concepts

What is a mock object? – A mock object is a simulated object that mimic the behavior of some real object in controlled ways.

When do we need mocks? – In general, a mock is useful whenever we need an object of type Y but we don't have one real implementation or using a real implementation is not so simple as we want. Specifically, we have these typical scenarios:

- When we want to test or use an object of class X, which depends on an object of type Y, and we haven't implemented type Y and implementing Y may take a lot of time => we should create a mock of type Y
- When we want to test an object of class X, which depends on an object of interface Y, and the real implementation used is changeable, configurable or dynamically selected or created at runtime by the framework => we should create a mock of interface Y

*(Note: Besides **mock**, we also hear **stub**, **fake**, etc. They may be different depending on how we define them and they may be useful in other testing frameworks. In our case with Moq, it suffices to use **mock** for all.)*

## 2.2 - First examples

Consider these types:

```
public interface IAnimal
{
    bool CanRun { get; }

    void Run();
}
public class AnimalTamer
{
    /// <summary>
    /// If animal.CanRun returns true, call animal.Run().
    /// </summary>
    /// <param name="animal"></param>
    public void Tame(IAnimal animal)
    {
        if (animal.CanRun)
        {
            animal.Run();
        }
    }
}
```

Suppose we want to test that AnimalTamer.Tame() works as documented. Here is our first test using Moq:

```
[TestClass]
public class AnimalTamerTest
{
    [TestMethod]
    public void Tame_Calls_AnimalRun_If_AnimalCanRun()
    {
        var mock = new Mock<IAnimal>();

        mock.Setup(animal => animal.CanRun).Returns(true);

        IAnimal mockAnimal = mock.Object;
        var tamer = new AnimalTamer();
        tamer.Tame(mockAnimal);
```

```
            mock.Verify(animal => animal.Run());
        }
    }
```

It is easy to see the 4 steps we took:

- First, we create a mock builder for the IAnimal interface. This builder will help us set up the behaviors of our mock.
- Second, we set up what we expect from our mock. In this case, we expect that our mock will always return true for CanRun.
- Third, we use the mock created by the builder. In this case, we pass the mock (i.e. mock.Object) to tamer.Tame().
- Finally, we verify that our expectations have been met. In this case, we verify that Run() has been called at least once.

No more step is required, and steps 2 and 4 are optional depending on your testing purposes. That is to say, in the simplest form, we can create and use a Moq-based mock using a single expression like this:

```
new Mock<IAnimal>.Object
```

Wow, Moq is so simple, isn't it?

Now, let's look at our second test:

```
public void Tame_DoesNotCall_AnimalRun_If_AnimalCanNotRun()
{
    var mock = new Mock<IAnimal>();

    mock.SetupGet(animal => animal.CanRun).Returns(false);

    IAnimal mockAnimal = mock.Object;
    var tamer = new AnimalTamer();
    tamer.Tame(mockAnimal);

    mock.Verify(animal => animal.Run(), Times.Never());
}
```

You can get it, can't you?

## 2.3 - Features at a glance

From the above examples, we can see that:

- In term of syntaxes, Moq supports strong typing, generics, and lambda expressions, which makes our code more readable and free from magic-string issues, and enables unsurpassed VS intellisense integration from setting expectations, to specifying method call arguments, return values, etc.
- In term of APIs, Moq requires at most 4 simple steps to use it: create a mock builder, set up mock behaviors, use the mock object and optionally verify calls to it. Correspondingly, it provides types and members with names readable enough to save us from reading its documentation for the most part. As a result, we can quickly learn Moq.

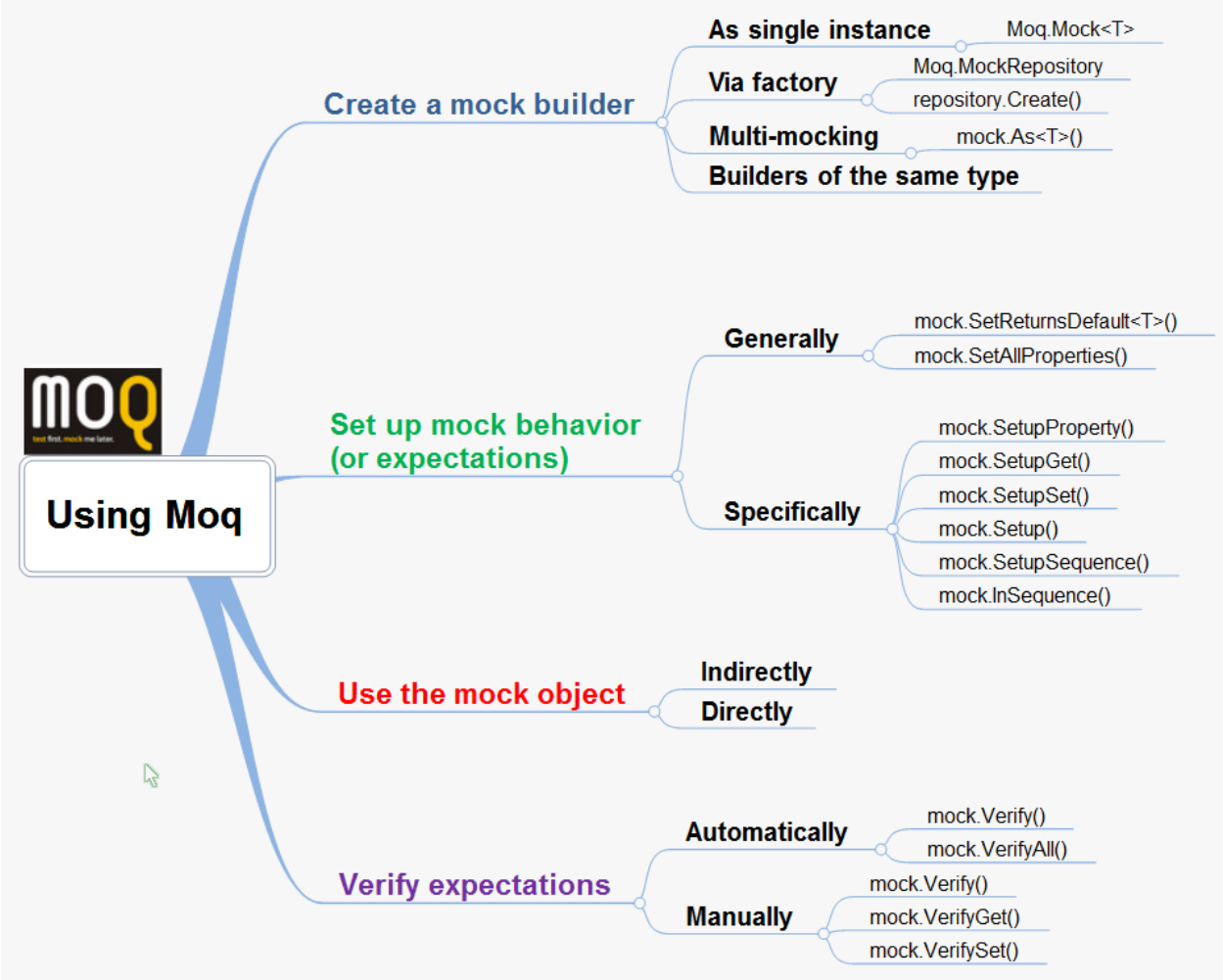
Here are something more to see via the tutorial code:

- In term of concepts, Moq does not require you to distinguish mock, stub, fake, dynamic mock, etc. but simply provides a simple MockBehavior enumeration so that you can granularly control over mock behavior.
- In term of mocking capabilities
    - Moq allows mocking both interfaces (single/multi mocking) and classes (virtual public + virtual protected members). According to the documentation, the interfaces and classes can be public or internal.
    - You can  pass constructor arguments for mocked classes, set constraints to method arguments, intercept and raise events on mocks, set up members with out/ref arguments, set up the sequence of results to return if the same member is called multiple times, or set up a sequence of calls to different members
    - You can also mock members recursively in a very simple manner
    - Moq-based expectations are overridable, i.e. when we have many setups for the same member using the same argument constraints,  only the last of them takes effect. Therefore, we can set default expectations in a fixture setup, and override as needed on tests.

## 2.4 - Moq API summary

This document is associated with a code solution that provides detailed tutorials on Moq API. Readers are expected to read the code and relevant comments for better understanding. Here, only a brief summary is included to show what API is available for each step.

Before we start, let's take a look at this image to have an overview of Moq.




### 2.4.1 - Create a mock builder

We can create a mock builder using the `Mock<T>` class in `Moq` namespace. For example:

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

If the class to mock provides parameterized constructors, we can pass arguments to select the expected constructor. For example:

```
public abstract class Animal
{
    public Animal(string name)
    {
        this.Name = name;
    }
    …
}
…
var mock = new Mock<Animal>("SomeName");
```

We can also specify a mock behavior, which can be:

- `MockBehavior.Strict`: always throws exception for invocations that don't have a corresponding setup
- `MockBehavior.Loose` (default): invocations can have no corresponding setup

For example:

```
var mock = new Mock<IAnimal>(MockBehavior.Strict);
IAnimal theAnimal = mock.Object;
TestUtils.AssertException<MockException>(() => theAnimal.Run());
```

*(Because our mock has strict behavior and we haven't set up any expectation for Run(), calling Run() will throw an exception )*

Other options include:

- If we want our mock to re-use the virtual implementation from the mocked class for all members having no corresponding setup, just set the `CallBase` property to **true**. The default value is **false**.
- If we want the default value of an reference-typed property/field to be non-null, set the `DefaultValue` property to **DefaultValue.Mock**. The default value is **DefaultValue.Empty**.

In brief, a full mock creation may look like this:

```
var mock = new Mock<IAnimal>( MockBehavior.Strict){
   CallBase = true,
   DefaultValue = DefaultValue.Mock
}
```

That was how you can create a single mock builder instance. In case you need to create several mock builders that have the same values for MockBehavior, CallBase, DefaultValue, or that will be verified all together, consider using a repository. In this example, strictMock1 and strictMock2 are created using the behavior set to strictRepository, strictMock3 is created using a different behavior:

```
var strictRepository = new MockRepository(MockBehavior.Strict);
var strictMock1 = strictRepository.Create<IAnimal>();
var strictMock2 = strictRepository.Create<Animal>(1);
var strictMock3 = strictRepository.Create<Animal>(MockBehavior.Loose,
"LooseAnimal");
```

*(**1** and "**LooseAnimal**" are arguments that will be passed to proper parameterized constructors of the `Animal` class)*

Finally, if we want to create a mock that implements several interfaces, we need to create several corresponding mock builders using the As<T>() method. For example:

```
var animalMock = new Mock<IAnimal>();
var disposableAnimalMock = animalMock.As<IDisposable>();
```

Then we have to set up each builder, e.g. `animalMock` and `disposableAnimalMock`, independently. The luck is `animalMock.Object` and `disposableAnimalMock.Object` will be actually the same object, so you can use any of them without hesitation.

### 2.4.2 - Set up mock behavior (or our expectations)

Moq allows us to set up how our mock will behave at 2 levels: generally and specifically.

General behavior set up refers to 'defining expectations that can be applied to several members'. For example: we can expect that:

- Any property or method that returns a **bool** value will return **true** by default.

```
mock.SetReturnsDefault<bool>(true);
```

- All properties must have '*property behavior*', meaning that *setting its value will cause it to be saved and later returned when the property is requested*.

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

As a small practice, try to understand why the following test passes:

```
public interface IAnimal
{
    bool CanRun { get; }
    string Name { get; set; }
    bool CanRunFor(int seconds);
    void Run();
}
…
public void Test01c_GeneralSetup_InCombination()
{
    var mock = new Mock<IAnimal>()
    {
        DefaultValue = DefaultValue.Empty
    };

    mock.SetReturnsDefault<string>("Demo");
    mock.SetReturnsDefault<bool>(true);
    mock.SetupAllProperties();

    IAnimal theAnimal = mock.Object;
    Assert.AreEqual("Demo", theAnimal.Name);

    Assert.IsTrue(theAnimal.CanRunFor(1));

    theAnimal.Name = "Something";
    Assert.AreEqual("Something", theAnimal.Name);
}
```

On the other hand, specific behavior set up refers to 'defining expectations that will be applied to a specific member'. For example: we can expect that:

- The Name property has 'property behavior' (as explained above)

```
mock.SetupProperty(animal => animal.Name);
```

- The get property `CanRun` will always return **true**:

```
mock.SetupGet(animal => animal.CanRun).Returns(true);
```

- Setting the property `Partner` to null will throw `NullException`. Setting it to any non-null value will raise the `Exhausted` event.

```
mock.SetupSet(animal => animal.Partner = null).Throws<NullException>();
mock.SetupSet(animal => animal.Partner = It.Is<IAnimal>(v => v != null))
    .Raises(animal => animal.Exhausted += null, new ExhaustedEventArgs());
```

- If we pass 10 to `CanRunFor()`, it will return false. If we pass a value between 0 and 9 (excluding 0 and 9) or larger than 10, it will return true. For other values, it will return the default value depending on this builder.

```
mock.Setup(animal => animal.CanRunFor(10)).Returns(false);
mock.Setup(animal => animal.CanRunFor(It.IsInRange(0,9,Range.Exclusive)))
    .Returns(true);
mock.Setup(animal => animal.CanRunFor(It.Is<int>(value => value > 10)))
    .Returns(true);
or
mock.Setup(animal => animal.CanRunFor(10)).Returns(false);
mock.Setup(animal => animal.CanRunFor(It.Is<int>(v => (v>0&&v<9)||v>10)))
    .Returns(true);
```

- Property `CanRun` will return the value of a local variable named `localCanRun`. Each time `CanRun` is called and returns, `localCanRun` is changed to the opposite value. So, if `CanRun` is called successively, it will return different values, not only true.

```
bool localCanRun = false;
mock.SetupGet(animal => animal.CanRun)
    .Returns(() => localCanRun)
    .Callback(() => localCanRun = !localCanRun);
```

- Our calls to `CanRun` are verifiable (so that when we call `Verify()` later, it will tell us whether `CanRun` has been called once), without caring about which value they return, but possibly caring about which arguments they are passed.

```
mock.SetupGet(animal => animal.CanRun).Verifiable();
```

(There are more to note on setting up our expections, but please take a look at the associated code project (file ***03_Step02.cs***))

For now, do this small exercise: why do the following tests pass?

```
public void Test03g_SpecificSetup_Expectations_WithRecursion()
{
    var mock = new Mock<IAnimal>();
    var mock2 = new Mock<IAnimal>();

    mock2.SetupGet(animal => animal.Partner.CanRun).Returns(true);
    mock2.SetupGet(animal => animal.Partner.Name).Returns("Partner");

    IAnimal theAnimal = mock.Object;
    Assert.IsNull(theAnimal.Partner);

    IAnimal theAnimal2 = mock2.Object;
    Assert.IsNotNull(theAnimal2.Partner);
    Assert.IsTrue(theAnimal2.Partner.CanRun);
```

```
    Assert.AreEqual("Partner", theAnimal2.Partner.Name);
}

public void Test03h_SpecificSetup_Expectations_WithReturnSequence()
{
    var mock = new Mock<IAnimal>();

    mock.SetupSequence(animal => animal.CanRun)
       .Returns(true)
       .Returns(false)
       .Returns(false)
       .Returns(true);

    IAnimal theAnimal = mock.Object;
    Assert.IsTrue(theAnimal.CanRun);
    Assert.IsFalse(theAnimal.CanRun);
    Assert.IsFalse(theAnimal.CanRun);
    Assert.IsTrue(theAnimal.CanRun);
}
```

So, we have known that we can set up our expectations generally or specifically. You may wonder if we can combine the 2 levels. Yes, we can. But note that:

- Specific setups will override general setups. For example, the following setup will mean “All properties must have "property behavior", but Name always returns "Demo" regardless of what value it is set to, and setting Partner to null always throws NotImplementedException”:

```
mock.SetupAllProperties();

mock.SetupGet(animal => animal.Name).Returns("Demo");
mock.SetupSet(animal => animal.Partner = null).Throws<NotImplException>();
```

- If we use `SetReturnsDefault()` and `SetupAllProperties()` together plus some specific setups, only `SetupAllProperties()` is effective. (This looks like a bug of Moq, rather than the author's intention not wanting us to use both at the same time.)

### 2.4.3 - Use the mock object

This is the simplest step of all. Well, we can use a mock object directly (in few cases) or indirectly (in most cases).

- Using a mock object directly means "Calling its member and check for the changes", as follows:

```
public void Test02_UseMockDirectly()
{
    var mock = new Mock<IAnimal>();

    mock.SetupAllProperties();
    mock.SetupGet(animal => animal.CanRun).Returns(true);

    IAnimal theAnimal = mock.Object;
    theAnimal.Name = "Demo";
    Assert.AreEqual("Demo", theAnimal.Name);
    Assert.IsTrue(theAnimal.CanRun);
```

```
}
```

- Using a mock object indirectly means "Pass it to another object X, call X's member(s) and optionally check for the changes", as follows:

```
public interface IAnimal
{
    bool CanRun { get; }
    string Name { get; set; }
    void Run();
}

public class AnimalTamer
{
    public void Tame(IAnimal animal)
    {
        if (animal.CanRun)
        {
            animal.Run();
            animal.Name = "Tamed " + animal.Name;
        }
    }
}

public void Test01_UseMockInDirectly()
{
    var mock = new Mock<IAnimal>();
    var tamer = new AnimalTamer();

    mock.SetupAllProperties();
    mock.SetupGet(animal => animal.CanRun).Returns(true);

    IAnimal theAnimal = mock.Object;
    theAnimal.Name = "Demo";
    tamer.Tame(theAnimal);
    Assert.AreEqual("Tamed Demo", theAnimal.Name);
}
```

As you can guess, a mock object is mainly used indirectly. It is used directly in some rare cases, such as when we're testing a mocking framework.

### 2.4.4 - Verify that our expectations are met

After creating a mock builder, setting up expectations and using the mock object, we may want to do one optional last thing: verify that our expectations are met, which means 'making sure that they have been called and the number of calls satisfies some criteria.' Moq provides `Verify()` and `VerifyAll()` for that purpose. For example, we can verify that:

- Each verifiable expectation (marked with Verifiable()) must be called at least once.

```
mock.Verify()
```

- All explicit expectations must be called at least once

```
mock.VerifyAll()
```

- A call to CanRun must have been called at least once.

```
mock.VerifyGet(animal => animal.CanRun, Times.AtLeastOnce());
```

- A call to CanRunFor() must never have been called with any argument.

```
mock.Verify(animal => animal.CanRunFor(It.IsAny<int>()), Times.Never());
```

Note:

- If we don't set up any expectation, calling `Verify()` will never throw an exception.
- If we have expectations that are not called, `Verify()` will throw `MockVerificationException`.

(If you want to know more, please look at the associated code project.)

That's all for Moq basics. Before we leave this section, please read this test to see if you understand it thoroughly.

```
[TestMethod]
public void Test03_Verify_AllExpectations()
{
    var mock = new Mock<IAnimal>();
    var mock2 = new Mock<IAnimal>();

    mock.Setup(animal => animal.CanRunFor(100)).Returns(true);
    mock.SetupGet(animal => animal.CanRun).Returns(true).Verifiable();

    mock2.Setup(animal => animal.CanRunFor(100)).Returns(true);
    mock2.SetupGet(animal => animal.CanRun).Returns(true).Verifiable();

    IAnimal theAnimal = mock.Object;
    Assert.IsTrue(theAnimal.CanRun);

    IAnimal theAnimal2 = mock2.Object;
    Assert.IsTrue(theAnimal2.CanRun);

    mock.Verify();
    TestUtils.AssertException(()=>mock2.VerifyAll(), "Moq.MockException");
}
```

I hope you do ☺.

## 3 - Best practices

- From my experience, Moq is not the most powerful mocking framework, but it's good enough to cover most of your needs with mocking. In case where your mocking needs are not achievable with Moq, it is recommended to check whether the needs make sense or whether it's time to update your code, before seeking for another mocking framework.
- Moq allows mocking internal types and protected members, but we should avoid doing that as much as possible. Focus on public class/interface, and public virtual members.

- If you need to use runtime values to set up expectations, use lambda expressions. Otherwise, use normal expressions.
- Depending on your needs, you may want `Returns()` or `Callback()` to be called first, but try to stick to one style.
- You are free to choose the order of assertion arguments, but your expected and actual values should follow the order suggested by the framework you use to write assertions. Moreover, make sure your expressions have the expected values in your assertions by using as fixed and literal values as possible.
- When you write your test, make sure you use `Assert.AreSame()`, not `Assert.ReferenceEquals()` to check whether 2 objects are the same instance. The former is a part of Assert class, while the latter is a common method that all .NET objects have. (This last tip has nothing to do with Moq, but while writing the associated code project, I made a big mistake in using `Assert.ReferenceEquals()` instead of `Assert.AreSame()`, which led me to some wrong comments on Moq. So, I list it out for your reference.)