

# Быстрый старт в Moq.

[leave a comment »](#)

Moq разработан чтобы быть простым, строго типизированным, небольшим, но в тоже время реализовывать большую функциональность.

## Методы

```
var mock = new Mock<IFoo>();
mock.Setup(foo => foo.DoSomething(``"ping"``)).Returns(``true``);`
`// out аргументы`
var `outString = ` "ack" ``;`
`// TryParse возвратит true, а out аргумент - "ack"`
mock.Setup(foo => foo.TryParse(``"ping"``, `out`
`outString)).Returns(``true``);`
`// ref аргумент`
var `instance = ` new ` Bar();`
`// Совпадает только если ref аргумент того же экземпляра`
mock.Setup(foo => foo.Submit(``ref` `instance)).Returns(``true``);`
`// доступные аргументы для вызова когда возвращается значение`
mock.Setup(x => x.DoSomething(It.IsAny<`string``>()))`
`.Returns((`string` `s) => s.ToLower());`
`// выбрасывание исключения при вызове`
mock.Setup(foo =>
foo.DoSomething(``"reset"``)).Throws<InvalidOperationException>();`
mock.Setup(foo => foo.DoSomething(``""``)).Throws(``new`
`ArgumentException(``"command"``));`
`// ленивое вычисление возвращаемого значения`
mock.Setup(foo => foo.GetCount()).Returns(() => count);`
`// возвращение различных значений при каждом вызове`
var `mock = ` new ` Mock<IFoo>();`
var `calls = 0;`
mock.Setup(foo => foo.GetCountThing())`
`.Returns(() => calls)`
`.Callback(() => calls++);`
`// возвращает 0 при первом вызове, 1 при втором и т.д.`
Console.WriteLine(mock.Object.GetCountThing());`
```

## Методы

```
1 var mock = new Mock<IFoo>();
2 mock.Setup(foo => foo.DoSomething("ping")).Returns(true);
3
4 // out аргументы
5 var outString = "ack";
6 // TryParse возвратит true, а out аргумент - "ack"
7 mock.Setup(foo => foo.TryParse("ping", out outString)).Returns(true);
8
9 // ref аргумент
10 var instance = new Bar();
11 // Совпадает только если ref аргумент того же экземпляра
12 mock.Setup(foo => foo.Submit(ref instance)).Returns(true);
13
14 // доступные аргументы для вызова когда возвращается значение
15 mock.Setup(x => x.DoSomething(It.IsAny<string>()))
16     .Returns((string s) => s.ToLower());
17
18 // выбрасывание исключения при вызове
19 mock.Setup(foo => foo.DoSomething("reset")).Throws<InvalidOperationException>();
20 mock.Setup(foo => foo.DoSomething("")).Throws(new ArgumentException("command"));
21
22 // ленивое вычисление возвращаемого значения
23 mock.Setup(foo => foo.GetCount()).Returns(() => count);
24
25 // возвращение различных значений при каждом вызове
26 var mock = new Mock<IFoo>();
27 var calls = 0;
28 mock.Setup(foo => foo.GetCountThing())
29     .Returns(() => calls)
30     .Callback(() => calls++);
31 // возвращает 0 при первом вызове, 1 при втором и т.д.
32 Console.WriteLine(mock.Object.GetCountThing());
```

## Совпадающие аргументы

```
`// любое значение`
`mock.Setup(foo => foo.DoSomething(It.IsAny<`string`>
  ())).Returns(`true`);`
`// соответствие Func<int>, ленивый вызов`
`mock.Setup(foo => foo.Add(It.Is<`int`>(i => i % 2 ==
  0))).Returns(`true`);`
`// соответствие диапазонов`
`mock.Setup(foo => foo.Add(It.IsInRange<`int`>(0, 10,
  Range.Inclusive))).Returns(`true`);`
`// соответствие регулярному выражению`
`mock.Setup(x => x.DoSomething(It.IsRegex(`"[a-d]+"`,
  RegexOptions.IgnoreCase))).Returns(`"foo"`);`
```

## Совпадающие аргументы

```
1 // любое значение
2 mock.Setup(foo => foo.DoSomething(It.IsAny<string>())).Returns(true);
3
4 // соответствие Func<int>, ленивый вызов
5 mock.Setup(foo => foo.Add(It.Is<int>(i => i % 2 == 0))).Returns(true);
6
7 // соответствие диапазонов
8 mock.Setup(foo => foo.Add(It.IsInRange<int>(0, 10, Range.Inclusive))).Returns(true);
9
10 // соответствие регулярному выражению
11 mock.Setup(x => x.DoSomething(It.IsRegex("[a-d]+", RegexOptions.IgnoreCase))).Returns("foo");
```

# Свойства

```
`mock.Setup(foo => foo.Name).Returns(``"bar"``);`  
`// иерархия свойств (для рекурсивных заглушек)`  
`mock.Setup(foo => foo.Bar.Baz.Name).Returns(``"baz"``);`  
`// ожидание вызова установки значения "foo"`  
`mock.SetupSet(foo => foo.Name = ``"foo"``);`  
`// или прямо проверить установщик`  
`mock.VerifySet(foo => foo.Name = ``"foo"``);`
```

- Установка заглушек для свойств (также известное как "Stub")

```
`// запускаем "отслеживание" sets/gets для этого свойства`  
`mock.SetupProperty(f => f.Name);`  
`// или устанавливаем значение по умолчанию для`  
`"заглушенного" свойства`  
`mock.SetupProperty(f => f.Name, ``"foo"``);`  
`// теперь мы можем сделать:`  
`IFoo foo = mock.Object;`  
`// первоначальное значение было установлено`  
`Assert.Equal(``"foo"``, foo.Name);`  
`// устанавливаем новое значение свойства`  
`foo.Name = ``"bar"``;`  
`Assert.Equal(``"bar"``, foo.Name);`
```

- Заглушаем все свойства в mock-объекте (не доступно в Silverlight):

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

## Свойства

```
1 mock.Setup(foo => foo.Name).Returns("bar");
2
3 // иерархия свойств (для рекурсивные заглушки)
4 mock.Setup(foo => foo.Bar.Baz.Name).Returns("baz");
5
6 // ожидание вызова установки значения "foo"
7 mock.SetupSet(foo => foo.Name = "foo");
8
9 // или прямо проверить установщик
10 mock.VerifySet(foo => foo.Name = "foo");
```

- Установка заглушек для свойств (также известное как "Stub")

```
1 // запускаем "отслеживание" sets/gets для этого свойства
2 mock.SetupProperty(f => f.Name);
3
4 // или устанавливаем значение по умолчанию для "заглушенного" свойства
5 mock.SetupProperty(f => f.Name, "foo");
6
7 // теперь мы можем сделать:
8 IFoo foo = mock.Object;
9 // первоначальное значение было установлено
10 Assert.Equal("foo", foo.Name);
11
12 // устанавливаем новое значение свойства
13 foo.Name = "bar";
14 Assert.Equal("bar", foo.Name);
```

- Заглушаем все свойства в mock-объекте (не доступно в Silverlight):

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

## События

```
`// вызов события над mock-объектом`
`mock.Raise(m => m.FooEvent += ` `null``, ` `new`
`FooEventArgs(fooValue));`

`// вызов события по нисходящей иерархии`
`mock.Raise(m => m.Child.First.FooEvent += ` `null``, `
`new` `FooEventArgs(fooValue));`

`// Автоматическое порождение события при вызове метода
Submit`

`mock.Setup(foo => foo.Submit()).Raises(f => f.Sent += `
`null``, EventArgs.Empty);`

`// вызов пользовательского события не придерживающегося
паттерна EventHandler`

`public` `delegate` `void` `MyEventHandler(``int` `i,`
`bool` `b);`

`public` `interface` `IFoo`
`{`
`    `event` `MyEventHandler MyEvent;`
`}`

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

`// вызываем наше событие и передаем ему нужные
аргументы`

`mock.Raise(foo => foo.MyEvent += ` `null``, 25,`
`true`);
```

## События

```
1 // вызов события над mock-объектом
2 mock.Raise(m => m.FooEvent += null, new FooEventArgs(fooValue));
3
4 // вызов события по нисходящей иерархии
5 mock.Raise(m => m.Child.First.FooEvent += null, new FooEventArgs(fooValue));
6
7 // Автоматическое порождение события при вызове метода Submit
8 mock.Setup(foo => foo.Submit()).Raises(f => f.Sent += null, EventArgs.Empty);
9
10 // вызов пользовательского события не придерживающегося паттерна EventHandler
11 public delegate void MyEventHandler(int i, bool b);
12 public interface IFoo
13 {
14     event MyEventHandler MyEvent;
15 }
16
17 var mock = new Mock<IFoo>();
18 ...
19 // вызываем наше событие и передаем ему нужные аргументы
20 mock.Raise(foo => foo.MyEvent += null, 25, true);
```

### ## Обратные вызовы (Callbacks)

```
`var` `mock` = `new` `Mock<IFoo>();`
`mock.Setup(foo => foo.Execute(``"ping"``))`
`.Returns(``true``)`
`.Callback(() => calls++);`
`// доступ к аргументам вызова`
`mock.Setup(foo => foo.Execute(It.IsAny<``string``>()))`
`.Returns(``true``)`
`.Callback((``string` `s` => calls.Add(s));`
`// альтернативный метод использую синтаксис generic`
`mock.Setup(foo => foo.Execute(Is.IsAny<``string``>()))`
`.Returns(``true``)`
`.Callback<``string``>(s => calls.Add(s));`
`// доступ к аргументам метода со множеством параметров`
`mock.Setup(foo => foo.Execute(It.IsAny<``int``>(),`
It.IsAny<``string``>()))`
`.Returns(``true``)`
`.Callback<``int`,``string``>((i, s) =>`
calls.Add(s));`
`// callback-и могут быть заданы до и после вызова`
`mock.Setup(foo => foo.Execute(``"ping"``))`
`.Callback(() => Console.WriteLine(``"Before`
returns"``))`
`.Returns(``true``)`
`.Callback(() => Console.WriteLine(``"After`
returns"``));`
```

## Обратные вызовы (Callbacks)

```
1 var mock = new Mock<IFoo>();
2 mock.Setup(foo => foo.Execute("ping"))
3     .Returns(true)
4     .Callback(() => calls++);
5
6 // доступ к аргументам вызова
7 mock.Setup(foo => foo.Execute(It.IsAny<string>()))
8     .Returns(true)
9     .Callback((string s) => calls.Add(s));
10
11 // альтернативный метод использую синтаксис generic методов
12 mock.Setup(foo => foo.Execute(Is.IsAny<string>()))
13     .Returns(true)
14     .Callback<string>(s => calls.Add(s));
15
16 // доступ к аргументам метода со множеством параметров
17 mock.Setup(foo => foo.Execute(It.IsAny<int>(), It.IsAny<string>()))
18     .Returns(true)
19     .Callback<int, string>((i, s) => calls.Add(s));
20
21 // callback-и могут быть заданы до и после вызова
22 mock.Setup(foo => foo.Execute("ping"))
23     .Callback(() => Console.WriteLine("Before returns"))
24     .Returns(true)
25     .Callback(() => Console.WriteLine("After returns"));
```

### ## Верификация

```
`mock.Verify(foo => foo.Execute(``"ping"``));`
`// проверка с выводом собственного сообщения об ошибке`
`mock.Verify(foo => foo.Execute(``"ping"``), ` ` "Когда`
выполняем операцию X сервис должен быть всегда пингуем" `);`
`// метод никогда не должен быть вызван`
`mock.Verify(foo => foo.Execute(``"ping"``),`
Times.Never());`
`// вызван хотя бы раз`
`mock.Verify(foo => foo.Execute(``"ping"``),`
Times.AtLeastOnce());`
`mock.VerifyGet(foo => foo.Name);`
`// проверка setter вызова, вне зависимости от`
значения.`
`mock.VerifySet(foo => foo.Name);`
`// проверка, что setter вызван с определенным`
значением`
`mock.VerifySet(foo => foo.Name = ` ` "foo" `);`
`// проверка setter на совпадение с аргументом`
`mock.VerifySet(foo => foo.Value = It.IsInRange(1, 5,`
Range.Inclusive));`
```

## Верификация

```
1 mock.Verify(foo => foo.Execute("ping"));
2
3 // проверка с выводом собственного сообщения об ошибке
4 mock.Verify(foo => foo.Execute("ping"), "Когда выполняем операцию X сервис должен быть всег
5
6 // метод никогда не должен быть вызван
7 mock.Verify(foo => foo.Execute("ping"), Times.Never());
8
9 // вызван хотя бы раз
10 mock.Verify(foo => foo.Execute("ping"), Times.AtLeastOnce());
11
12 mock.VerifyGet(foo => foo.Name);
13
14 // проверка setter вызова, вне зависимости от значения.
15 mock.VerifySet(foo => foo.Name);
16
17 // проверка, что setter вызван с определенным значением
18 mock.VerifySet(foo => foo.Name = "foo");
19
20 // проверка setter на совпадение с аргументом
21 mock.VerifySet(foo => foo.Value = It.IsInRange(1, 5, Range.Inclusive));
```

## ## Настраиваемое поведение mock-объектов

- Создавайте mock-объекты в духе “чистого Mock” – выбрасывайте исключения для всего, что не соответствует вашим ожиданиям, на сленге это называется “сильный Mock”. Поведение по умолчанию это “свободный” mock-объект, никогда не возбуждающий исключений и не возвращающий значения по умолчанию или пустые массивы, списки перечисления и т.д. если для него не заданы никакие ожидания.

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

- Вызывайте реализацию базового класса, если не заданы ожидания переопределяющие какой-либо член класса (для “частичные mock-и” в Rhino Mocks): по умолчанию false. (Это необходимо если вы “подделываете” Web/Html элементы управления в System.Web!)

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

- Создавайте автоматически генерируемые рекурсивные mock-объекты: mock-объект возвращающий новый mock-объект для каждого члена для которого не задано поведение и его возвращаемое значение может быть подделано.

```
`var` `mock` = ` `new` `Mock<IFoo> { DefaultValue = DefaultValue.Mock
};`
    `// значение по умолчанию DefaultValue.Empty`
    `// это свойство будет возвращать новый mock-объект IBar'a`
```

```

    `IBar value = mock.Object.Bar;`
    `// полученный mock-объект повторно используется, т.ч.
    дальнейший доступ к этому свойству возвратит`
    `// тот же самый mock-объект. Это позволяет нам использовать
    этот объект для установления дальнейших`
    `// ожиданий. Конечно, если мы это захотим.`
    `var` `barMock = Mock.Get(value);`
    `barMock.Setup(b => b.Submit()).Returns(``true``);`

```

- Централизуйте создание и управление mock-объектами: вы можете создать и проверить все mock-объекты в единственном месте используя MockFactory, которое позволяет устанавливать MockBehavior, а также его CallBase и DefaultValue соответственно.

```

    `var` `factory =` `new` `MockFactory(MockBehavior.Strict) {
    DefaultValue = DefaultValue.Mock };`
    `// создаем mock-объект используя настройки фабрики`
    `var` `fooMock = factory.Create<IFoo>();`
    `// создаем mock-объект переопределяющий настройки фабрики`
    `var` `barMock = factory.Create<IBar>(MockBehavior.Loose);`
    `// проверка всех ожиданий у всех объектов созданных с
    помощью фабрики`
    `factory.Verify();`

```



## Настраиваемое поведение mock-объектов

- Создавайте mock-объекты в духе "чистого Mock" – выбрасывайте исключения для всего, что не соответствует вашим ожиданиям, на сленге это называется "сильный Mock". Поведение по умолчанию это "свободный" mock-объект, никогда не возбуждающие исключений и не возвращающий значения по умолчанию или пустые массивы, списки перечисления и т.д. если для него не заданы никакие ожидания.

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

- Вызывайте реализацию базового класса, если не заданы ожидания переопределяющие какой-либо член класса (для "частичные mock-и" в Rhino Mocks): по умолчанию false. (Это необходимо если вы "подделываете" Web/Html элементы управления в System.Web!)

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

- Создавайте автоматически генерирующиеся рекурсивные mock-объекты: mock-объект возвращающий новый mock-объект для каждого члена для которого не задано поведение и его возвращающее значение может быть подделанно.

```
1 | var mock = new Mock<IFoo> { DefaultValue = DefaultValue.Mock };
2 | // значение по умолчанию DefaultValue.Empty
3 |
4 | // это свойство будет возвращать новый mock-объект IBar'a
5 | IBar value = mock.Object.Bar;
6 |
7 | // полученный mock-объект повторно используется, т.ч. дальнейший доступ к этому свойству
8 | // тот же самый mock-объект. Это позволяет нам использовать этот объект для установления
9 | // ожиданий. Конечно, если мы это захотим.
10 | var barMock = Mock.Get(value);
11 | barMock.Setup(b => b.Submit()).Returns(true);
```

- Централизируйте создание и управление mock-объектами: вы можете создать и проверить все mock-объекты в единственном месте используя MockFactory, которое позволяет устанавливать MockBehavior, а также его CallBase и DefaultValue соответственно.

```
1 | var factory = new MockFactory(MockBehavior.Strict) { DefaultValue = DefaultValue.Mock }
2 |
3 | // создаем mock-объект используя настройки фабрики
4 | var fooMock = factory.Create<IFoo>();
5 |
6 | // создаем mock-объект переопределяющий настройки фабрики
7 | var barMock = factory.Create<IBar>(MockBehavior.Loose);
8 |
9 | // проверка всех ожиданий у всех объектов созданных с помощью фабрики
10 | factory.Verify();
```

## Прочее

Установка ожиданий над защищенными членами класса (вы не сможете использовать IntelliSense для этого, т.ч. они будут доступны как строки содержащие их названия):

```
`// добавим в начале файла с тестами`
`using` `Moq.Protected`;
`// в тесте`
`var` `mock` = `new` `Mock<CommandBase>();`
`mock.Protected()`
`.Setup<`int`>(`"Execute"`)`
`.Return(5);`
`// если вам нужно совпадение аргументов вы ДОЛЖНЫ`
`использовать ItExpr вместо It`
`mock.Protected()`
`.Setup<`string`>(`"Execute"`,
ItExpr.IsAny<`string`>())`
`.Returns(`true`);`
```

## Дополнительный возможности

```
`// получение mock-объекта из уже подделанного объекта`
```

```

    `IFoo foo =` `// получаем подделанный объект каким-либо
способом`

    `var` `fooMock = Mock.Get(foo);`

    `fooMock.Setup(f => f.Submit()).Returns(``true``);`

    `// реализация множества интерфейсов в mock-объекте`

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

    `var` `disposableFoo = foo.As<IDisposable>();`

    `// теперь IFoo mock-объект также реализует IDisposable `

    `disposableFoo.Setup(df => df.Dispose());`

    `// пользовательские совпадения`

    `mock.Setup(foo =>
foo.Submit(IsLarge())).Throws<ArgumentException>());`

    `...`

    `public` `string` `IsLarge()`

    `{`

        `return` `Match<``string``>.Create(s =>
!String.IsNullOrEmpty(s) && s.Length > 100);`

    `}`

```

- Подделка внутренних (internal) типов в других проектах: добавьте следующий атрибут сборки (обычно в AssemblyInfo.cs) в проект содержащий внутренние типы:  
// это динамическая сборка созданная по умолчанию Castle DynamicProxy  
используемая Moq. Вставьте в одну строчку.`

```

[assembly:InternalsVisibleTo(``"DynamicProxyGenAssembly2,PublicKey=

```

```
0024000004800000940000000602000000240000525341310004000001000100c547
cac37abd99c8db225ef2f6c8a3602f3b3606cc9891605d02baa56104f4cfc0734aa3
9b93bf7852f7d9266654753cc297e7d2edfe0bac1cdcf9f717241550e0a7b191195b
7667bb4f64bcb8e2121380fd1d9d46ad2d92d2d15605093924cceaaf74c4861eff62a
bf69b9291ed0a340e113be11e6a7d3113e92484cf7045cc7"``)J`
```

## Прочее

Установка ожиданий над защищенными членами класса (вы не сможете использовать IntelliSense для этого, т.ч. они будут доступны как строки содержащие их названия):

```
1 // добавим в начале файла с тестами
2 using Moq.Protected;
3
4 // в тесте
5 var mock = new Mock<CommandBase>();
6 mock.Protected()
7     .Setup<int>("Execute")
8     .Return(5);
9
10 // если вам нужно совпадение аргументов вы ДОЛЖНЫ использовать ItExpr вместо It
11 mock.Protected()
12     .Setup<string>("Execute", ItExpr.IsAny<string>())
13     .Returns(true);
```

## Дополнительный возможности

```
1 // получение mock-объекта из уже подделанного объекта
2 IFoo foo = // получаем подделанный объект каким-либо способом
3 var fooMock = Mock.Get(foo);
4 fooMock.Setup(f => f.Submit()).Returns(true);
5
6 // реализация множества интерфейсов в mock-объекте
7 var foo = new Mock<IFoo>();
8 var disposableFoo = foo.As<IDisposable>();
9 // теперь IFoo mock-объект также реализует IDisposable <img draggable="false" role="img" class="disposable" />
10 disposableFoo.Setup(df => df.Dispose());
11
12 // пользовательские совпадения
13 mock.Setup(foo => foo.Submit(IsLarge())).Throws<ArgumentException>();
14 ...
15 public string IsLarge()
16 {
17     return Match<string>.Create(s => !String.IsNullOrEmpty(s) && s.Length > 100);
18 }
```

- Подделка внутренних (internal) типов в других проектах: добавьте следующий атрибут сборки (обычно в AssemblyInfo.cs) в проект содержащий внутренние типы:

```
// это динамическая сборка созданная по умолчанию Castle DynamicProxy используемая Moq.  
[assembly:InternalsVisibleTo("DynamicProxyGenAssembly2,PublicKey=0024000004800000940000")]
```