

As I have told you earlier, I'm learning Typemock Isolator Basics:

## Understanding Mocking and Isolation

What is mocking and isolation? How mocking helps in unit testing? Installing Typemock Isolator

Downloading and installing Typemock Isolator. Integrating it into NUnit test projects. Basic Mocking with Isolate.Fake

Creating basic mock objects using Isolate.Fake. Configuring the behavior of mocked objects. Mocking Static Methods and Properties

Using Isolate.Fake.StaticMethods() to mock static members. Mocking static properties with Isolate.WhenCalled(). Mocking Sealed Classes

Mocking sealed classes using Isolate.Fake.NextInstance(). Configuring expectations for sealed class mocks. Mocking Private Methods and Properties

Mocking private methods with Isolate.WhenCalled(). Accessing and modifying private properties of objects under test.

Teach me this now: Using Isolate.Fake.StaticMethods() to mock static members. Mocking static properties with Isolate.WhenCalled(). Mocking Sealed Classes

Certainly! Let's go through each concept with clear examples:

### 1. Creating Basic Mock Objects using Isolate.Fake:

Suppose we have a simple `Calculator` class:

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

Now, let's create a mock `calculator` object using Typemock Isolator:

```
using Typemock;

public class CalculatorTests
{
    [Test]
```

```
public void Test_Add_Method_With_Mock()
{
    // Arrange
    var mockCalculator = Isolate.Fake.Instance<Calculator>();

    // Act
    int result = mockCalculator.Add(2, 3);

    // Assert
    Assert.AreEqual(0, result); // This will fail since it's a mock and not the actual method
}
```

In the above example, the `calculator` class is not implemented; it's just a mock created by Typemock Isolator. When we call the `Add` method on the mock, it doesn't have any actual implementation, so it returns a default value (in this case, `0`). We should not expect the correct result, as the purpose of this test is just to demonstrate how to create a basic mock.

## 2. Configuring the Behavior of Mocked Objects:

Let's modify the previous example and configure the behavior of the mocked `Add` method:

```
using Typemock;

public class CalculatorTests
{
    [Test]
    public void Test_Add_Method_With_Mock()
    {
        // Arrange
        var mockCalculator = Isolate.Fake.Instance<Calculator>();
        Isolate.WhenCalled(() => mockCalculator.Add(2, 3)).WillReturn(5);

        // Act
        int result = mockCalculator.Add(2, 3);

        // Assert
        Assert.AreEqual(5, result); // Now the test passes as we've configured the mock to return
    }
}
```

In this example, we have configured the mocked `Add` method to return `5` when called with arguments `2` and `3`. Now the test will pass because the mocked method behaves as we defined.

## 3. Mocking Static Methods and Properties:

Suppose we have a static helper class:

```
public static class Helper
{
    public static string GetMessage()
    {
        return "Hello, World!";
    }

    public static int CalculateSum(int a, int b)
    {
        return a + b;
    }

    public static string StaticProperty { get; set; }
}
```

Now, let's mock the static `CalculateSum` method and the `StaticProperty` using Typemock Isolator:

```
using Typemock;

public class HelperTests
{
    [Test]
    public void Test_Static_Method_And_Property_Mocks()
    {
        // Arrange
        Isolate.Fake.StaticMethods(typeof(Helper));
        Isolate.WhenCalled(() => Helper.CalculateSum(2, 3)).WillReturn(10);
        Isolate.WhenCalled(() => Helper.StaticProperty).WillReturn("Mocked Value");

        // Act
        int result = Helper.CalculateSum(2, 3);
        string message = Helper.GetMessage();
        string propertyValue = Helper.StaticProperty;

        // Assert
        Assert.AreEqual(10, result); // Mocked value returned by the static method.
        Assert.AreEqual("Hello, World!", message); // Unmocked value returned by the GetMessage()
        Assert.AreEqual("Mocked Value", propertyValue); // Mocked value returned by the static pr
    }
}
```

In this example, we used `Isolate.Fake.StaticMethods()` to mock all static members of the `Helper` class. We then configured the mocked `CalculateSum` method to return `10`, and the mocked `StaticProperty`

to return "Mocked Value" .The GetMessage() method is not mocked, so it behaves normally.

#### 4. Mocking Sealed Classes:

Suppose we have a sealed SealedClass :

```
public sealed class SealedClass
{
    public string GetMessage()
    {
        return "Original Message";
    }
}
```

Now, let's mock the SealedClass using Typemock Isolator:

```
using Typemock;

public class SealedClassTests
{
    [Test]
    public void Test_Sealed_Class_Mock()
    {
        // Arrange
        var mockedSealedInstance = Isolate.Fake.NextInstance<SealedClass>();
        Isolate.WhenCalled(() => mockedSealedInstance.GetMessage()).WillReturn("Mocked Message");

        // Act
        string result = mockedSealedInstance.GetMessage();

        // Assert
        Assert.AreEqual("Mocked Message", result); // Mocked value returned by the sealed class m
    }
}
```

In this example, we used Isolate.Fake.NextInstance() to create a mocked instance of the SealedClass . We then configured the GetMessage method of the mocked instance to return "Mocked Message" . Now, when we call the GetMessage method on the mocked instance, it will return the mocked value.

#### 5. Mocking Private Methods and Properties:

Suppose we have a class Person with a private method and a private property:

```
public class Person
{
    private string GetName()
    {
        return "John Doe";
    }

    private int Age { get; set; }

    public string GetFullName()
    {
        return $"{GetName()}, {Age} years old.";
    }
}
```

Now, let's mock the private `GetName` method and access the private `Age` property using Typemock Isolator:

```
using Typemock;

public class PersonTests
{
    [Test]
    public void Test_Private_Method_And_Property_Mocks()
    {
        // Arrange
        var person = new Person();
        Isolate.NonPublic.WhenCalled(person, "GetName").WillReturn("Mocked Name");
        Isolate.NonPublic.WhenGetCalled(person, "Age").WillReturn(30);

        // Act
        string result = person.GetFullName();

        // Assert
        Assert.AreEqual("Mocked Name, 30 years old.", result); // Mocked values used in the full
    }
}
```

In this example, we used `Isolate.NonPublic.WhenCalled()` to mock the private `GetName` method and return `"Mocked Name"`. We also used `Isolate.NonPublic.WhenGetCalled()` to access the private `Age` property and return `30`. Now, when we call the `GetFullName` method on the `Person` instance, it will use the mocked values for the name and age in the returned full name.

Sure, let's dive into each topic one by one using clear examples:

## 1. Using Isolate.Fake.StaticMethods() to Mock Static Members:

In unit testing, mocking static members (methods and properties) can be challenging with traditional mocking frameworks. However, with Typemock Isolator, you can easily mock static methods and properties.

Suppose we have a static utility class `MathUtils` :

```
public static class MathUtils
{
    public static int Add(int a, int b)
    {
        return a + b;
    }

    public static string GetMessage()
    {
        return "Hello, World!";
    }

    public static string StaticProperty { get; set; }
}
```

Now, let's mock the `Add` method and the `StaticProperty` of the `MathUtils` class using Typemock Isolator:

```
using NUnit.Framework;
using Typemock;

public class MathUtilsTests
{
    [Test]
    public void Test_Static_Method_And_Property_Mocks()
    {
        // Arrange
        Isolate.Fake.StaticMethods(typeof(MathUtils));
        Isolate.WhenCalled(() => MathUtils.Add(2, 3)).WillReturn(10);
        Isolate.WhenCalled(() => MathUtils.StaticProperty).WillReturn("Mocked Value");

        // Act
        int result = MathUtils.Add(2, 3);
        string message = MathUtils.GetMessage();
        string propertyValue = MathUtils.StaticProperty;

        // Assert
        Assert.AreEqual(10, result); // Mocked value returned by the static method.
```

```

        Assert.AreEqual("Hello, World!", message); // Unmocked value returned by the GetMessage()
        Assert.AreEqual("Mocked Value", propertyValue); // Mocked value returned by the static pr
    }
}

```

In the above example, we used `Isolate.Fake.StaticMethods(typeof(MathUtils))` to mock all static members of the `MathUtils` class. We then configured the mocked `Add` method to return `10`, and the mocked `StaticProperty` to return `"Mocked Value"`. The `GetMessage()` method is not mocked, so it behaves normally.

## 2. Mocking Static Properties with `Isolate.WhenCalled()`:

Now, let's focus on mocking just the static property `StaticProperty` of the `MathUtils` class using `Isolate.WhenCalled()`:

```

using NUnit.Framework;
using Typemock;

public class MathUtilsTests
{
    [Test]
    public void Test_Static_Property_Mock_With_Isolate_WhenCalled()
    {
        // Arrange
        Isolate.WhenCalled(() => MathUtils.StaticProperty).WillReturn("Mocked Value");

        // Act
        string propertyValue = MathUtils.StaticProperty;

        // Assert
        Assert.AreEqual("Mocked Value", propertyValue); // Mocked value returned by the static pr
    }
}

```

In this example, we only used `Isolate.WhenCalled(() => MathUtils.StaticProperty)` to configure the mocked value of the `StaticProperty`. We didn't use `Isolate.Fake.StaticMethods()` this time since we are only interested in mocking the property and not the methods.

## 3. Mocking Sealed Classes using `Isolate.Fake.NextInstance()`:

In traditional mocking frameworks, mocking sealed classes can be challenging, but with `Typemock Isolator`, you can easily mock sealed classes.

Suppose we have a sealed `SealedClass`:

```
public sealed class SealedClass
{
    public string GetMessage()
    {
        return "Original Message";
    }
}
```

Now, let's mock the `SealedClass` using Typemock Isolator:

```
using NUnit.Framework;
using Typemock;

public class SealedClassTests
{
    [Test]
    public void Test_Sealed_Class_Mock()
    {
        // Arrange
        var mockedSealedInstance = Isolate.Fake.NextInstance<SealedClass>();
        Isolate.WhenCalled(() => mockedSealedInstance.GetMessage()).WillReturn("Mocked Message");

        // Act
        string result = mockedSealedInstance.GetMessage();

        // Assert
        Assert.AreEqual("Mocked Message", result); // Mocked value returned by the sealed class m
    }
}
```

In this example, we used `Isolate.Fake.NextInstance<SealedClass>()` to create a mocked instance of the `SealedClass`. We then configured the `GetMessage` method of the mocked instance to return "Mocked Message". Now, when we call the `GetMessage` method on the mocked instance, it will return the mocked value.

That concludes the explanations and examples for using `Isolate.Fake.StaticMethods()` to mock static members, mocking static properties with `Isolate.WhenCalled()`, and mocking sealed classes using `Isolate.Fake.NextInstance()` in Typemock Isolator. These features make it easier to isolate and mock different components during unit testing, allowing you to focus on testing specific units of code in isolation.

Sure, let's continue with the explanations and examples for each of these topics:

### 1. Mocking Sealed Classes using `Isolate.Fake.NextInstance()`:



As mentioned earlier, mocking sealed classes can be challenging with traditional mocking frameworks, but Typemock Isolator makes it easy.

Suppose we have a sealed `SealedClass` :

```
public sealed class SealedClass
{
    public string GetMessage()
    {
        return "Original Message";
    }
}
```

Now, let's mock the `SealedClass` using `Isolate.Fake.NextInstance()` in Typemock Isolator:

```
using NUnit.Framework;
using Typemock;

public class SealedClassTests
{
    [Test]
    public void Test_Sealed_Class_Mock_Using_NextInstance()
    {
        // Arrange
        var mockedSealedInstance = Isolate.Fake.NextInstance<SealedClass>();
        Isolate.WhenCalled(() => mockedSealedInstance.GetMessage()).WillReturn("Mocked Message");

        // Act
        string result = mockedSealedInstance.GetMessage();

        // Assert
        Assert.AreEqual("Mocked Message", result); // Mocked value returned by the sealed class m
    }
}
```

In this example, we used `Isolate.Fake.NextInstance<SealedClass>()` to create a mocked instance of the `SealedClass`. We then configured the `GetMessage` method of the mocked instance to return "Mocked Message". Now, when we call the `GetMessage` method on the mocked instance, it will return the mocked value.

## 2. Configuring Expectations for Sealed Class Mocks:

Configuring expectations for mocks allows you to set up the behavior that you expect from the mocked class during unit testing.

Let's modify the previous example to include an expectation for the number of times the `GetMessage` method should be called:

```
using NUnit.Framework;
using Typemock;

public class SealedClassTests
{
    [Test]
    public void Test_Sealed_Class_Mock_With_Expectation()
    {
        // Arrange
        var mockedSealedInstance = Isolate.Fake.NextInstance<SealedClass>();
        Isolate.WhenCalled(() => mockedSealedInstance.GetMessage()).WillReturn("Mocked Message");

        // Act
        string result1 = mockedSealedInstance.GetMessage();
        string result2 = mockedSealedInstance.GetMessage();

        // Assert
        Assert.AreEqual("Mocked Message", result1); // First call returns the mocked value.
        Assert.AreEqual("Mocked Message", result2); // Subsequent call also returns the mocked va

        // Expectation: Verify that GetMessage() is called exactly two times.
        Isolate.Verify.WasCalledWithExactArguments(() => mockedSealedInstance.GetMessage(), 2);
    }
}
```

In this example, we added a second call to `GetMessage` after the initial call. We then used `Isolate.Verify.WasCalledWithExactArguments()` to check that `GetMessage()` was called exactly two times as expected.

### 3. Mocking Private Methods and Properties:

Mocking private methods and properties can be important when you want to test certain scenarios without exposing those members to the public interface.

Suppose we have a class `Person` with a private method and a private property:

```
public class Person
{
    private string GetName()
    {
        return "John Doe";
    }
}
```

```
private int Age { get; set; }

public string GetFullName()
{
    return $"{GetName()}, {Age} years old.";
}
}
```

Now, let's mock the private `GetName` method and access the private `Age` property using `Typemock Isolator`:

```
using NUnit.Framework;
using Typemock;

public class PersonTests
{
    [Test]
    public void Test_Private_Method_And_Property_Mocks()
    {
        // Arrange
        var person = new Person();
        Isolate.NonPublic.WhenCalled(person, "GetName").WillReturn("Mocked Name");
        Isolate.NonPublic.WhenGetCalled(person, "Age").WillReturn(30);

        // Act
        string result = person.GetFullName();

        // Assert
        Assert.AreEqual("Mocked Name, 30 years old.", result); // Mocked values used in the full
    }
}
```

In this example, we used `Isolate.NonPublic.WhenCalled()` to mock the private `GetName` method and return `"Mocked Name"`. We also used `Isolate.NonPublic.WhenGetCalled()` to access the private `Age` property and return `30`. Now, when we call the `GetFullName` method on the `Person` instance, it will use the mocked values for the name and age in the returned full name.

That concludes the explanations and examples for mocking sealed classes using `Isolate.Fake.NextInstance()`, configuring expectations for sealed class mocks, and mocking private methods and properties using `Isolate.NonPublic`. These features provided by `Typemock Isolator` allow you to create more comprehensive and thorough unit tests for your code, even in scenarios that might be difficult to test with traditional mocking frameworks.

Certainly! Let's continue with the explanations and examples for each of these topics:

### 1. Mocking Private Methods with Isolate.WhenCalled():

Mocking private methods allows you to isolate and test the behavior of specific methods within your class without exposing them to the public interface. Typemock Isolator enables you to mock private methods using `Isolate.WhenCalled()`.

Suppose we have a class `Calculator` with a private method:

```
public class Calculator
{
    private int Add(int a, int b)
    {
        return a + b;
    }

    public int AddAndDouble(int a, int b)
    {
        return Add(a, b) * 2;
    }
}
```

Now, let's mock the private `Add` method using Typemock Isolator:

```
using NUnit.Framework;
using Typemock;

public class CalculatorTests
{
    [Test]
    public void Test_Private_Method_Mock()
    {
        // Arrange
        var calculator = new Calculator();
        Isolate.WhenCalled(() => calculator.PrivateMethod("Hello", "World")).WillReturn("Mocked R

        // Act
        int result = calculator.AddAndDouble(2, 3);

        // Assert
        Assert.AreEqual(10, result); // The result will be based on the mocked behavior of the Ad
    }
}
```

In this example, we used `Isolate.WhenCalled(() => calculator.PrivateMethod("Hello", "World"))` to mock the private `Add` method and return a mocked result `"Mocked Result"`. The `AddAndDouble` method will then use the mocked result in the calculation.

## 2. Accessing and Modifying Private Properties of Objects Under Test:

Sometimes, you may need to access and modify private properties of objects under test to set up specific test scenarios. Typemock Isolator allows you to do this using `Isolate.NonPublic`.

Let's modify the previous example to access and modify the private `Age` property of the `Person` class:

```
using NUnit.Framework;
using Typemock;

public class PersonTests
{
    [Test]
    public void Test_Access_And_Modify_Private_Property()
    {
        // Arrange
        var person = new Person();

        // Access and Modify the private Age property
        Isolate.NonPublic.WhenGetCalled(person, "Age").WillReturn(25);
        Isolate.NonPublic.WhenSetCalled(person, "Age").IgnoreCall();

        // Act
        string result = person.GetFullName();

        // Assert
        Assert.AreEqual("John Doe, 25 years old.", result); // The Age property will be set to 25
    }
}
```

In this example, we used `Isolate.NonPublic.WhenGetCalled(person, "Age").WillReturn(25)` to access the private `Age` property and set it to `25`. We also used `Isolate.NonPublic.WhenSetCalled(person, "Age").IgnoreCall()` to ignore any calls to set the `Age` property, ensuring that it remains set to `25`. Now, when we call the `GetFullName` method on the `Person` instance, it will use the modified `Age` property in the returned full name.

That concludes the explanations and examples for mocking private methods with `Isolate.WhenCalled()` and accessing and modifying private properties of objects under test using `Isolate.NonPublic`. These features provided by Typemock Isolator allow you to thoroughly test private logic and properties within your classes, enhancing the scope and robustness of your unit tests.

You've covered a lot of important topics to get started with Typemock Isolator. However, there are a few more essential concepts and features that can be useful in your unit testing journey:

## 1. Verifying Method Calls:

Typemock Isolator allows you to verify that specific methods have been called during your unit tests. This is useful to ensure that certain methods are being invoked as expected.

```
using NUnit.Framework;
using Typemock;

public class SomeClassTests
{
    [Test]
    public void Test_Method_Called()
    {
        // Arrange
        var mock = Isolate.Fake.Instance<SomeClass>();

        // Act
        mock.MethodToTest();

        // Assert
        Isolate.Verify.WasCalledWithExactArguments(() => mock.MethodToTest());
    }
}
```

In this example, we used `Isolate.Verify.WasCalledWithExactArguments(() => mock.MethodToTest())` to verify that the `MethodToTest` was called exactly once with the expected arguments.

## 2. Mocking Constructors:

You can also mock constructors of classes using Typemock Isolator. This is helpful when you want to isolate certain parts of your code from external dependencies during unit testing.

```
using NUnit.Framework;
using Typemock;

public class SomeClassTests
{
    [Test]
    public void Test_Constructor_Mock()
    {
        // Arrange
        Isolate.WhenCalled(() => new SomeClass()).WillReturn(null);
    }
}
```

```
// Act
var result = new SomeClass();

// Assert
Assert.IsNull(result); // The constructor will return null due to the mocked behavior.
}
}
```

### 3. Replacing Method Implementations:

With Typemock Isolator, you can also replace the implementation of methods with custom logic during testing, allowing you to test different scenarios without modifying the original code.

```
using NUnit.Framework;
using Typemock;

public class SomeClassTests
{
    [Test]
    public void Test_Replace_Method_Implementation()
    {
        // Arrange
        var mock = Isolate.Fake.Instance<SomeClass>();
        Isolate.WhenCalled(() => mock.MethodToReplace()).WillReturn("Mocked Value");

        // Act
        var result = mock.MethodToReplace();

        // Assert
        Assert.AreEqual("Mocked Value", result); // The method will return the mocked value instead of the original implementation.
    }
}
```

### 4. Handling Static Constructors:

In certain cases, you may need to handle the static constructors of classes in your unit tests. Typemock Isolator provides a way to deal with static constructors using `Isolate.Invoke.StaticConstructor`.

```
using NUnit.Framework;
using Typemock;

public class SomeClassTests
{
    [Test]
    public void Test_Static_Constructor()
```

```
{  
    // Arrange  
    Isolate.Invoke.StaticConstructor(typeof(SomeClass));  
  
    // Act  
    var result = SomeClass.StaticMethod();  
  
    // Assert  
    Assert.AreEqual("Static Method", result); // The static constructor will be invoked, and  
}  
}
```

## 5. Ignoring Base Class Constructors:

When working with inheritance, you may encounter scenarios where you want to ignore the base class constructors during testing. Typemock Isolator provides a way to achieve this using `Isolate.WhenCalled`.

```
using NUnit.Framework;  
using Typemock;  
  
public class DerivedClassTests  
{  
    [Test]  
    public void Test_Ignore_Base_Constructor()  
    {  
        // Arrange  
        var derivedInstance = Isolate.Fake.Instance<DerivedClass>();  
        Isolate.WhenCalled(() => base.ArglessMethod()).IgnoreCall();  
  
        // Act  
        var result = derivedInstance.ArglessMethod();  
  
        // Assert  
        Assert.AreEqual("Derived Argless Method", result); // The base constructor call is ignore  
    }  
}
```

These additional concepts and features should provide you with a more comprehensive understanding of Typemock Isolator and enable you to write more robust and efficient unit tests for your software applications. Remember to focus on the behavior of your code and write tests that cover different scenarios to ensure the correctness of your application's functionality. Happy testing!