Mockery Docs Documentation

Release 1.6.6

Pádraic Brady, Dave Marshall, Wouter, Graham Campbell

Dec 10, 2023

Contents

1 Mock Objects **3**

2 Getting Started **5**
2.1 Getting Started 5

3 Reference **13**
3.1 Reference 13

4 Mockery **43**
4.1 Mockery 43

5 Cookbook **49**
5.1 Cookbook 49

Index **61**

Mockery is a simple yet flexible PHP mock object framework for use in unit testing with PHPUnit, PHPSpec or any other testing framework. Its core goal is to offer a test double framework with a succinct API capable of clearly defining all possible object operations and interactions using a human readable Domain Specific Language (DSL). Designed as a drop in alternative to PHPUnit's phpunit-mock-objects library, Mockery is easy to integrate with PHPUnit and can operate alongside phpunit-mock-objects without the World ending.

CHAPTER 1

Mock Objects

In unit tests, mock objects simulate the behaviour of real objects. They are commonly utilised to offer test isolation, to stand in for objects which do not yet exist, or to allow for the exploratory design of class APIs without requiring actual implementation up front.

The benefits of a mock object framework are to allow for the flexible generation of such mock objects (and stubs). They allow the setting of expected method calls and return values using a flexible API which is capable of capturing every possible real object behaviour in way that is stated as close as possible to a natural language description.

CHAPTER 2

Getting Started

Ready to dive into the Mockery framework? Then you can get started by reading the "Getting Started" section!

2.1 Getting Started

2.1.1 Installation

Mockery can be installed using Composer or by cloning it from its GitHub repository. These two options are outlined below.

Composer

You can read more about Composer on getcomposer.org. To install Mockery using Composer, first install Composer for your project using the instructions on the Composer download page. You can then define your development dependency on Mockery using the suggested parameters below. While every effort is made to keep the master branch stable, you may prefer to use the current stable version tag instead (use the `@stable` tag).

```
{
    "require-dev": {
        "mockery/mockery": "dev-master"
    }
}
```

To install, you then may call:

```
php composer.phar update
```

This will install Mockery as a development dependency, meaning it won't be installed when using `php composer.phar update --no-dev` in production.

Other way to install is directly from composer command line, as below.

```
php composer.phar require --dev mockery/mockery
```

Git

The Git repository hosts the development version in its master branch. You can install this using Composer by referencing `dev-master` as your preferred version in your project's `composer.json` file as the earlier example shows.

2.1.2 Upgrading

Upgrading to 1.0.0

Minimum PHP version

As of Mockery 1.0.0 the minimum PHP version required is 5.6.

Using Mockery with PHPUnit

In the "old days", 0.9.x and older, the way Mockery was integrated with PHPUnit was through a PHPUnit listener. That listener would in turn call the `\Mockery::close()` method for us.

As of 1.0.0, PHPUnit test cases where we want to use Mockery, should either use the `\Mockery\Adapter\Phpunit\MockeryPHPUnitIntegration` trait, or extend the `\Mockery\Adapter\Phpunit\MockeryTestCase` test case. This will in turn call the `\Mockery::close()` method for us.

Read the documentation for a detailed overview of "*PHPUnit Integration*".

`\Mockery\Matcher\MustBe` is deprecated

As of 1.0.0 the `\Mockery\Matcher\MustBe` matcher is deprecated and will be removed in Mockery 2.0.0. We recommend instead to use the PHPUnit equivalents of the MustBe matcher.

`allows` and `expects`

As of 1.0.0, Mockery has two new methods to set up expectations: `allows` and `expects`. This means that these methods names are now "reserved" for Mockery, or in other words classes you want to mock with Mockery, can't have methods called `allows` or `expects`.

Read more in the documentation about this "*Alternative shouldReceive Syntax*".

No more implicit regex matching for string arguments

When setting up string arguments in method expectations, Mockery 0.9.x and older, would try to match arguments using a regular expression in a "last attempt" scenario.

As of 1.0.0, Mockery will no longer attempt to do this regex matching, but will only try first the identical operator ===, and failing that, the equals operator ==.

If you want to match an argument using regular expressions, please use the new `\Mockery\Matcher\Pattern` matcher. Read more in the documentation about this pattern matcher in the "*Argument Validation*" section.

andThrow \Throwable

As of 1.0.0, the `andThrow` can now throw any `\Throwable`.

Upgrading to 0.9

The generator was completely rewritten, so any code with a deep integration to mockery will need evaluating.

Upgrading to 0.8

Since the release of 0.8.0 the following behaviours were altered:

1. The `shouldIgnoreMissing()` behaviour optionally applied to mock objects returned an instance of `\Mockery\Undefined` when methods called did not match a known expectation. Since 0.8.0, this behaviour was switched to returning `null` instead. You can restore the 0.7.2 behaviour by using the following:

```
$mock = \Mockery::mock('stdClass')->shouldIgnoreMissing()->asUndefined();
```

2.1.3 Simple Example

Imagine we have a `Temperature` class which samples the temperature of a locale before reporting an average temperature. The data could come from a web service or any other data source, but we do not have such a class at present. We can, however, assume some basic interactions with such a class based on its interaction with the `Temperature` class:

```
class Temperature
{
    private $service;

    public function __construct($service)
    {
        $this->service = $service;
    }

    public function average()
    {
        $total = 0;
        for ($i=0; $i<3; $i++) {
            $total += $this->service->readTemp();
        }
        return $total/3;
    }
}
```

Even without an actual service class, we can see how we expect it to operate. When writing a test for the `Temperature` class, we can now substitute a mock object for the real service which allows us to test the behaviour of the `Temperature` class without actually needing a concrete service instance.

```
use \Mockery;

class TemperatureTest extends \PHPUnit\Framework\TestCase
{
    public function tearDown()
    {
        Mockery::close();
    }

    public function testGetsAverageTemperatureFromThreeServiceReadings()
    {
        $service = Mockery::mock('service');
        $service->shouldReceive('readTemp')
            ->times(3)
            ->andReturn(10, 12, 14);

        $temperature = new Temperature($service);

        $this->assertEquals(12, $temperature->average());
    }
}
```

We create a mock object which our `Temperature` class will use and set some expectations for that mock — that it should receive three calls to the `readTemp` method, and these calls will return 10, 12, and 14 as results.

Note: PHPUnit integration can remove the need for a `tearDown()` method. See "*PHPUnit Integration*" for more information.

2.1.4 Quick Reference

The purpose of this page is to give a quick and short overview of some of the most common Mockery features.

Do read the *Reference* to learn about all the Mockery features.

Integrate Mockery with PHPUnit, either by extending the `MockeryTestCase`:

```
use \Mockery\Adapter\Phpunit\MockeryTestCase;

class MyTest extends MockeryTestCase
{
}
```

or by using the `MockeryPHPUnitIntegration` trait:

```
use \PHPUnit\Framework\TestCase;
use \Mockery\Adapter\Phpunit\MockeryPHPUnitIntegration;

class MyTest extends TestCase
{
    use MockeryPHPUnitIntegration;
}
```

Creating a test double:

```
$testDouble = \Mockery::mock('MyClass');
```

Creating a test double that implements a certain interface:

```
$testDouble = \Mockery::mock('MyClass, MyInterface');
```

Expecting a method to be called on a test double:

```
$testDouble = \Mockery::mock('MyClass');
$testDouble->shouldReceive('foo');
```

Expecting a method to **not** be called on a test double:

```
$testDouble = \Mockery::mock('MyClass');
$testDouble->shouldNotReceive('foo');
```

Expecting a method to be called on a test double, once, with a certain argument, and to return a value:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->once()
    ->with($arg)
    ->andReturn($returnValue);
```

Expecting a method to be called on a test double and to return a different value for each successive call:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->andReturn(1, 2, 3);

$mock->foo(); // int(1);
$mock->foo(); // int(2);
$mock->foo(); // int(3);
$mock->foo(); // int(3);
```

Creating a runtime partial test double:

```
$mock = \Mockery::mock('MyClass')->makePartial();
```

Creating a spy:

```
$spy = \Mockery::spy('MyClass');
```

Expecting that a spy should have received a method call:

```
$spy = \Mockery::spy('MyClass');

$spy->foo();

$spy->shouldHaveReceived()->foo();
```

Not so simple examples

Creating a mock object to return a sequence of values from a set of method calls:

```
use \Mockery\Adapter\Phpunit\MockeryTestCase;

class SimpleTest extends MockeryTestCase
{
    public function testSimpleMock()
    {
        $mock = \Mockery::mock(array('pi' => 3.1416, 'e' => 2.71));
        $this->assertEquals(3.1416, $mock->pi());
        $this->assertEquals(2.71, $mock->e());
    }
}
```

Creating a mock object which returns a self-chaining Undefined object for a method call:

```
use \Mockery\Adapter\Phpunit\MockeryTestCase;

class UndefinedTest extends MockeryTestCase
{
    public function testUndefinedValues()
    {
        $mock = \Mockery::mock('mymock');
        $mock->shouldReceive('divideBy')->with(0)->andReturnUndefined();
        $this->assertTrue($mock->divideBy(0) instanceof \Mockery\Undefined);
    }
}
```

Creating a mock object with multiple query calls and a single update call:

```
use \Mockery\Adapter\Phpunit\MockeryTestCase;

class DbTest extends MockeryTestCase
{
    public function testDbAdapter()
    {
        $mock = \Mockery::mock('db');
        $mock->shouldReceive('query')->andReturn(1, 2, 3);
        $mock->shouldReceive('update')->with(5)->andReturn(NULL)->once();

        // ... test code here using the mock
    }
}
```

Expecting all queries to be executed before any updates:

```
use \Mockery\Adapter\Phpunit\MockeryTestCase;

class DbTest extends MockeryTestCase
{
    public function testQueryAndUpdateOrder()
    {
        $mock = \Mockery::mock('db');
        $mock->shouldReceive('query')->andReturn(1, 2, 3)->ordered();
        $mock->shouldReceive('update')->andReturn(NULL)->once()->ordered();

        // ... test code here using the mock
    }
}
```

Creating a mock object where all queries occur after startup, but before finish, and where queries are expected with several different params:

```
use \Mockery\Adapter\Phpunit\MockeryTestCase;

class DbTest extends MockeryTestCase
{
    public function testOrderedQueries()
    {
        $db = \Mockery::mock('db');
        $db->shouldReceive('startup')->once()->ordered();
        $db->shouldReceive('query')->with('CPWR')->andReturn(12.3)->once()->ordered(
↪'queries');
        $db->shouldReceive('query')->with('MSFT')->andReturn(10.0)->once()->ordered(
↪'queries');
        $db->shouldReceive('query')->with(\Mockery::pattern("/^....$/"))->andReturn(3.
↪3)->atLeast()->once()->ordered('queries');
        $db->shouldReceive('finish')->once()->ordered();

        // ... test code here using the mock
    }
}
```


- *Installation*
- *Upgrading*
- *Simple Example*
- *Quick Reference*

- *Installation*
- *Upgrading*
- *Simple Example*
- *Quick Reference*

CHAPTER 3

Reference

The reference contains a complete overview of all features of the Mockery framework.

3.1 Reference

3.1.1 Creating Test Doubles

Mockery's main goal is to help us create test doubles. It can create stubs, mocks, and spies.

Stubs and mocks are created the same. The difference between the two is that a stub only returns a preset result when called, while a mock needs to have expectations set on the method calls it expects to receive.

Spies are a type of test doubles that keep track of the calls they received, and allow us to inspect these calls after the fact.

When creating a test double object, we can pass in an identifier as a name for our test double. If we pass it no identifier, the test double name will be unknown. Furthermore, the identifier does not have to be a class name. It is a good practice, and our recommendation, to always name the test doubles with the same name as the underlying class we are creating test doubles for.

If the identifier we use for our test double is a name of an existing class, the test double will inherit the type of the class (via inheritance), i.e. the mock object will pass type hints or `instanceof` evaluations for the existing class. This is useful when a test double must be of a specific type, to satisfy the expectations our code has.

Stubs and mocks

Stubs and mocks are created by calling the `\Mockery::mock()` method. The following example shows how to create a stub, or a mock, object named "foo":

```
$mock = \Mockery::mock('foo');
```

The mock object created like this is the loosest form of mocks possible, and is an instance of `\Mockery\MockInterface`.

Note: All test doubles created with Mockery are an instance of `\Mockery\MockInterface`, regardless are they a stub, mock or a spy.

To create a stub or a mock object with no name, we can call the `mock()` method with no parameters:

```
$mock = \Mockery::mock();
```

As we stated earlier, we don't recommend creating stub or mock objects without a name.

Classes, abstracts, interfaces

The recommended way to create a stub or a mock object is by using a name of an existing class we want to create a test double of:

```
$mock = \Mockery::mock('MyClass');
```

This stub or mock object will have the type of `MyClass`, through inheritance.

Stub or mock objects can be based on any concrete class, abstract class or even an interface. The primary purpose is to ensure the mock object inherits a specific type for type hinting.

```
$mock = \Mockery::mock('MyInterface');
```

This stub or mock object will implement the `MyInterface` interface.

Note: Classes marked final, or classes that have methods marked final cannot be mocked fully. Mockery supports creating partial mocks for these cases. Partial mocks will be explained later in the documentation.

Mockery also supports creating stub or mock objects based on a single existing class, which must implement one or more interfaces. We can do this by providing a comma-separated list of the class and interfaces as the first argument to the `\Mockery::mock()` method:

```
$mock = \Mockery::mock('MyClass, MyInterface, OtherInterface');
```

This stub or mock object will now be of type `MyClass` and implement the `MyInterface` and `OtherInterface` interfaces.

Note: The class name doesn't need to be the first member of the list but it's a friendly convention to use for readability.

We can tell a mock to implement the desired interfaces by passing the list of interfaces as the second argument:

```
$mock = \Mockery::mock('MyClass', 'MyInterface, OtherInterface');
```

For all intents and purposes, this is the same as the previous example.

Spies

The third type of test doubles Mockery supports are spies. The main difference between spies and mock objects is that with spies we verify the calls made against our test double after the calls were made. We would use a spy when we don't necessarily care about all of the calls that are going to be made to an object.

A spy will return `null` for all method calls it receives. It is not possible to tell a spy what will be the return value of a method call. If we do that, then we would deal with a mock object, and not with a spy.

We create a spy by calling the `\Mockery::spy()` method:

```
$spy = \Mockery::spy('MyClass');
```

Just as with stubs or mocks, we can tell Mockery to base a spy on any concrete or abstract class, or to implement any number of interfaces:

```
$spy = \Mockery::spy('MyClass, MyInterface, OtherInterface');
```

This spy will now be of type `MyClass` and implement the `MyInterface` and `OtherInterface` interfaces.

Note: The `\Mockery::spy()` method call is actually a shorthand for calling `\Mockery::mock()->shouldIgnoreMissing()`. The `shouldIgnoreMissing` method is a "behaviour modifier". We'll discuss them a bit later.

Mocks vs. Spies

Let's try and illustrate the difference between mocks and spies with the following example:

```
$mock = \Mockery::mock('MyClass');
$spy = \Mockery::spy('MyClass');

$mock->shouldReceive('foo')->andReturn(42);

$mockResult = $mock->foo();
$spyResult = $spy->foo();

$spy->shouldHaveReceived()->foo();

var_dump($mockResult); // int(42)
var_dump($spyResult); // null
```

As we can see from this example, with a mock object we set the call expectations before the call itself, and we get the return result we expect it to return. With a spy object on the other hand, we verify the call has happened after the fact. The return result of a method call against a spy is always `null`.

We also have a dedicated chapter to *Spies* only.

Partial Test Doubles

Partial doubles are useful when we want to stub out, set expectations for, or spy on *some* methods of a class, but run the actual code for other methods.

We differentiate between three types of partial test doubles:

- runtime partial test doubles,
- generated partial test doubles, and
- proxied partial test doubles.

Runtime partial test doubles

What we call a runtime partial, involves creating a test double and then telling it to make itself partial. Any method calls that the double hasn't been told to allow or expect, will act as they would on a normal instance of the object.

```
class Foo {
    function foo() { return 123; }
    function bar() { return $this->foo(); }
}

$foo = mock(Foo::class)->makePartial();
$foo->foo(); // int(123);
```

We can then tell the test double to allow or expect calls as with any other Mockery double.

```
$foo->shouldReceive('foo')->andReturn(456);
$foo->bar(); // int(456)
```

See the cookbook entry on *Big Parent Class* for an example usage of runtime partial test doubles.

Generated partial test doubles

The second type of partial double we can create is what we call a generated partial. With generated partials, we specifically tell Mockery which methods we want to be able to allow or expect calls to. All other methods will run the actual code *directly*, so stubs and expectations on these methods will not work.

```
class Foo {
    function foo() { return 123; }
    function bar() { return $this->foo(); }
}

$foo = mock("Foo[foo]");

$foo->foo(); // error, no expectation set

$foo->shouldReceive('foo')->andReturn(456);
$foo->foo(); // int(456)

// setting an expectation for this has no effect
$foo->shouldReceive('bar')->andReturn(999);
$foo->bar(); // int(456)
```

It's also possible to specify explicitly which methods to run directly using the *!method* syntax:

```
class Foo {
    function foo() { return 123; }
    function bar() { return $this->foo(); }
}

$foo = mock("Foo[!foo]");

$foo->foo(); // int(123)

$foo->bar(); // error, no expectation set
```

Note: Even though we support generated partial test doubles, we do not recommend using them.

One of the reasons why is because a generated partial will call the original constructor of the mocked class. This can have unwanted side-effects during testing application code.

See *Not Calling the Original Constructor* for more details.

Proxied partial test doubles

A proxied partial mock is a partial of last resort. We may encounter a class which is simply not capable of being mocked because it has been marked as final. Similarly, we may find a class with methods marked as final. In such a scenario, we cannot simply extend the class and override methods to mock - we need to get creative.

```
$mock = \Mockery::mock(new MyClass);
```

Yes, the new mock is a Proxy. It intercepts calls and reroutes them to the proxied object (which we construct and pass in) for methods which are not subject to any expectations. Indirectly, this allows us to mock methods marked final since the Proxy is not subject to those limitations. The tradeoff should be obvious - a proxied partial will fail any typehint checks for the class being mocked since it cannot extend that class.

Aliasing

Prefixing the valid name of a class (which is NOT currently loaded) with "alias:" will generate an "alias mock". Alias mocks create a class alias with the given classname to stdClass and are generally used to enable the mocking of public static methods. Expectations set on the new mock object which refer to static methods will be used by all static calls to this class.

```
$mock = \Mockery::mock('alias:MyClass');
```

Note: Even though aliasing classes is supported, we do not recommend it.

Overloading

Prefixing the valid name of a class (which is NOT currently loaded) with "overload:" will generate an alias mock (as with "alias:") except that created new instances of that class will import any expectations set on the origin mock (`$mock`). The origin mock is never verified since it's used an expectation store for new instances. For this purpose we use the term "instance mock" to differentiate it from the simpler "alias mock".

In other words, an instance mock will "intercept" when a new instance of the mocked class is created, then the mock will be used instead. This is useful especially when mocking hard dependencies which will be discussed later.

```
$mock = \Mockery::mock('overload:MyClass');
```

Note: Using alias/instance mocks across more than one test will generate a fatal error since we can't have two classes of the same name. To avoid this, run each test of this kind in a separate PHP process (which is supported out of the box by both PHPUnit and PHPT).

Named Mocks

The `namedMock()` method will generate a class called by the first argument, so in this example `MyClassName`. The rest of the arguments are treated in the same way as the `mock` method:

```
$mock = \Mockery::namedMock('MyClassName', 'DateTime');
```

This example would create a class called `MyClassName` that extends `DateTime`.

Named mocks are quite an edge case, but they can be useful when code depends on the `__CLASS__` magic constant, or when we need two derivatives of an abstract type, that are actually different classes.

See the cookbook entry on *Class Constants* for an example usage of named mocks.

Note: We can only create a named mock once, any subsequent calls to `namedMock`, with different arguments are likely to cause exceptions.

Constructor Arguments

Sometimes the mocked class has required constructor arguments. We can pass these to Mockery as an indexed array, as the 2nd argument:

```
$mock = \Mockery::mock('MyClass', [$constructorArg1, $constructorArg2]);
```

or if we need the `MyClass` to implement an interface as well, as the 3rd argument:

```
$mock = \Mockery::mock('MyClass', 'MyInterface', [$constructorArg1,
↪$constructorArg2]);
```

Mockery now knows to pass in `$constructorArg1` and `$constructorArg2` as arguments to the constructor.

Behavior Modifiers

When creating a mock object, we may wish to use some commonly preferred behaviours that are not the default in Mockery.

The use of the `shouldIgnoreMissing()` behaviour modifier will label this mock object as a Passive Mock:

```
\Mockery::mock('MyClass')->shouldIgnoreMissing();
```

In such a mock object, calls to methods which are not covered by expectations will return `null` instead of the usual error about there being no expectation matching the call.

On PHP >= 7.0.0, methods with missing expectations that have a return type will return either a mock of the object (if return type is a class) or a "falsy" primitive value, e.g. empty string, empty array, zero for ints and floats, false for bools, or empty closures.

On PHP >= 7.1.0, methods with missing expectations and nullable return type will return null.

We can optionally prefer to return an object of type `\Mockery\Undefined` (i.e. a `null` object) (which was the 0.7.2 behaviour) by using an additional modifier:

```
\Mockery::mock('MyClass')->shouldIgnoreMissing()->asUndefined();
```

The returned object is nothing more than a placeholder so if, by some act of fate, it's erroneously used somewhere it shouldn't, it will likely not pass a logic check.

We have encountered the `makePartial()` method before, as it is the method we use to create runtime partial test doubles:

```
\Mockery::mock('MyClass')->makePartial();
```

This form of mock object will defer all methods not subject to an expectation to the parent class of the mock, i.e. `MyClass`. Whereas the previous `shouldIgnoreMissing()` returned `null`, this behaviour simply calls the parent's matching method.

3.1.2 Expectation Declarations

Note: In order for our expectations to work we MUST call `Mockery::close()`, preferably in a callback method such as `tearDown` or `_after` (depending on whether or not we're integrating Mockery with another framework). This static call cleans up the Mockery container used by the current test, and run any verification tasks needed for our expectations.

Once we have created a mock object, we'll often want to start defining how exactly it should behave (and how it should be called). This is where the Mockery expectation declarations take over.

Declaring Method Call Expectations

To tell our test double to expect a call for a method with a given name, we use the `shouldReceive` method:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method');
```

This is the starting expectation upon which all other expectations and constraints are appended.

We can declare more than one method call to be expected:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method_1', 'name_of_method_2');
```

All of these will adopt any chained expectations or constraints.

It is possible to declare the expectations for the method calls, along with their return values:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive([
    'name_of_method_1' => 'return value 1',
    'name_of_method_2' => 'return value 2',
]);
```

There's also a shorthand way of setting up method call expectations and their return values:

```
$mock = \Mockery::mock('MyClass', ['name_of_method_1' => 'return value 1', 'name_of_
↪method_2' => 'return value 2']);
```

All of these will adopt any additional chained expectations or constraints.

We can declare that a test double should not expect a call to the given method name:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldNotReceive('name_of_method');
```

This method is a convenience method for calling `shouldReceive()->never()`.

Declaring Method Argument Expectations

For every method we declare expectation for, we can add constraints that the defined expectations apply only to the method calls that match the expected argument list:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->with($arg1, $arg2, ...);
// or
$mock->shouldReceive('name_of_method')
    ->withArgs([$arg1, $arg2, ...]);
```

We can add a lot more flexibility to argument matching using the built in matcher classes (see later). For example, `\Mockery::any()` matches any argument passed to that position in the `with()` parameter list. Mockery also allows Hamcrest library matchers - for example, the Hamcrest function `anything()` is equivalent to `\Mockery::any()`.

It's important to note that this means all expectations attached only apply to the given method when it is called with these exact arguments:

```
$mock = \Mockery::mock('MyClass');

$mock->shouldReceive('foo')->with('Hello');

$mock->foo('Goodbye'); // throws a NoMatchingExpectationException
```

This allows for setting up differing expectations based on the arguments provided to expected calls.

Argument matching with closures

Instead of providing a built-in matcher for each argument, we can provide a closure that matches all passed arguments at once:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->withArgs(closure);
```

The given closure receives all the arguments passed in the call to the expected method. In this way, this expectation only applies to method calls where passed arguments make the closure evaluate to true:

```
$mock = \Mockery::mock('MyClass');

$mock->shouldReceive('foo')->withArgs(function ($arg) {
    if ($arg % 2 == 0) {
        return true;
    }
    return false;
});
```

(continues on next page)

(continued from previous page)

```
$mock->foo(4); // matches the expectation
$mock->foo(3); // throws a NoMatchingExpectationException
```

Argument matching with some of given values

We can provide expected arguments that match passed arguments when mocked method is called.

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->withSomeOfArgs(arg1, arg2, arg3, ...);
```

The given expected arguments order doesn't matter. Check if expected values are included or not, but type should be matched:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->withSomeOfArgs(1, 2);

$mock->foo(1, 2, 3);  // matches the expectation
$mock->foo(3, 2, 1);  // matches the expectation (passed order doesn't matter)
$mock->foo('1', '2'); // throws a NoMatchingExpectationException (type should be␣
↪matched)
$mock->foo(3);        // throws a NoMatchingExpectationException
```

Any, or no arguments

We can declare that the expectation matches a method call regardless of what arguments are passed:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->withAnyArgs();
```

This is set by default unless otherwise specified.

We can declare that the expectation matches method calls with zero arguments:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->withNoArgs();
```

Declaring Return Value Expectations

For mock objects, we can tell Mockery what return values to return from the expected method calls.

For that we can use the `andReturn()` method:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andReturn($value);
```

This sets a value to be returned from the expected method call.

It is possible to set up expectation for multiple return values. By providing a sequence of return values, we tell Mockery what value to return on every subsequent call to the method:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andReturn($value1, $value2, ...)
```

The first call will return `$value1` and the second call will return `$value2`.

If we call the method more times than the number of return values we declared, Mockery will return the final value for any subsequent method call:

```
$mock = \Mockery::mock('MyClass');

$mock->shouldReceive('foo')->andReturn(1, 2, 3);

$mock->foo(); // int(1)
$mock->foo(); // int(2)
$mock->foo(); // int(3)
$mock->foo(); // int(3)
```

The same can be achieved using the alternative syntax:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andReturnValues([$value1, $value2, ...])
```

It accepts a simple array instead of a list of parameters. The order of return is determined by the numerical index of the given array with the last array member being returned on all calls once previous return values are exhausted.

The following two options are primarily for communication with test readers:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andReturnNull();
// or
$mock->shouldReceive('name_of_method')
    ->andReturn([null]);
```

They mark the mock object method call as returning `null` or nothing.

Sometimes we want to calculate the return results of the method calls, based on the arguments passed to the method. We can do that with the `andReturnUsing()` method which accepts one or more closure:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andReturnUsing(closure, ...);
```

Closures can be queued by passing them as extra parameters as for `andReturn()`.

Occasionally, it can be useful to echo back one of the arguments that a method is called with. In this case we can use the `andReturnArg()` method; the argument to be returned is specified by its index in the arguments list:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andReturnArg(1);
```

This returns the second argument (index #1) from the list of arguments when the method is called.

Note: We cannot currently mix `andReturnUsing()` or `andReturnArg` with `andReturn()`.

If we are mocking fluid interfaces, the following method will be helpful:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andReturnSelf();
```

It sets the return value to the mocked class name.

Throwing Exceptions

We can tell the method of mock objects to throw exceptions:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andThrow(new Exception);
```

It will throw the given `Exception` object when called.

Rather than an object, we can pass in the `Exception` class, message and/or code to use when throwing an `Exception` from the mocked method:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andThrow('exception_name', 'message', 123456789);
```

Setting Public Properties

Used with an expectation so that when a matching method is called, we can cause a mock object's public property to be set to a specified value, by using `andSet()` or `set()`:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->andSet($property, $value);
// or
$mock->shouldReceive('name_of_method')
    ->set($property, $value);
```

In cases where we want to call the real method of the class that was mocked and return its result, the `passthru()` method tells the expectation to bypass a return queue:

```
passthru()
```

It allows expectation matching and call count validation to be applied against real methods while still calling the real class method with the expected arguments.

Declaring Call Count Expectations

Besides setting expectations on the arguments of the method calls, and the return values of those same calls, we can set expectations on how many times should any method be called.

When a call count expectation is not met, a `\Mockery\Expectation\InvalidCountException` will be thrown.

Note: It is absolutely required to call `\Mockery::close()` at the end of our tests, for example in the `tearDown()` method of PHPUnit. Otherwise Mockery will not verify the calls made against our mock objects.

We can declare that the expected method may be called zero or more times:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->zeroOrMoreTimes();
```

This is the default for all methods unless otherwise set.

To tell Mockery to expect an exact number of calls to a method, we can use the following:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->times($n);
```

where `$n` is the number of times the method should be called.

A couple of most common cases got their shorthand methods.

To declare that the expected method must be called one time only:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->once();
```

To declare that the expected method must be called two times:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->twice();
```

To declare that the expected method must never be called:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->never();
```

Call count modifiers

The call count expectations can have modifiers set.

If we want to tell Mockery the minimum number of times a method should be called, we use `atLeast()`:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->atLeast()
    ->times(3);
```

`atLeast()->times(3)` means the call must be called at least three times (given matching method args) but never less than three times.

Similarly, we can tell Mockery the maximum number of times a method should be called, using `atMost()`:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->atMost()
    ->times(3);
```

`atMost()->times(3)` means the call must be called no more than three times. If the method gets no calls at all, the expectation will still be met.

We can also set a range of call counts, using `between()`:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method')
    ->between($min, $max);
```

This is actually identical to using `atLeast()->times($min)->atMost()->times($max)` but is provided as a shorthand. It may be followed by a `times()` call with no parameter to preserve the APIs natural language readability.

Multiple Calls with Different Expectations

If a method is expected to get called multiple times with different arguments and/or return values we can simply repeat the expectations. The same of course also works if we expect multiple calls to different methods.

```
$mock = \Mockery::mock('MyClass');
// Expectations for the 1st call
$mock->shouldReceive('name_of_method')
    ->once()
    ->with('arg1')
    ->andReturn($value1)

    // 2nd call to same method
    ->shouldReceive('name_of_method')
    ->once()
    ->with('arg2')
    ->andReturn($value2)

    // final call to another method
    ->shouldReceive('other_method')
    ->once()
    ->with('other')
    ->andReturn($value_other);
```

Expectation Declaration Utilities

Declares that this method is expected to be called in a specific order in relation to similarly marked methods.

```
ordered()
```

The order is dictated by the order in which this modifier is actually used when setting up mocks.

Declares the method as belonging to an order group (which can be named or numbered). Methods within a group can be called in any order, but the ordered calls from outside the group are ordered in relation to the group:

```
ordered(group)
```

We can set up so that method1 is called before group1 which is in turn called before method2.

When called prior to `ordered()` or `ordered(group)`, it declares this ordering to apply across all mock objects (not just the current mock):

```
globally()
```

This allows for dictating order expectations across multiple mocks.

The `byDefault()` marks an expectation as a default. Default expectations are applied unless a non-default expectation is created:

```
byDefault()
```

These later expectations immediately replace the previously defined default. This is useful so we can setup default mocks in our unit test `setup()` and later tweak them in specific tests as needed.

Returns the current mock object from an expectation chain:

```
getMock()
```

Useful where we prefer to keep mock setups as a single statement, e.g.:

```
$mock = \Mockery::mock('foo')->shouldReceive('foo')->andReturn(1)->getMock();
```

3.1.3 Argument Validation

The arguments passed to the `with()` declaration when setting up an expectation determine the criteria for matching method calls to expectations. Thus, we can setup up many expectations for a single method, each differentiated by the expected arguments. Such argument matching is done on a "best fit" basis. This ensures explicit matches take precedence over generalised matches.

An explicit match is merely where the expected argument and the actual argument are easily equated (i.e. using === or ==). More generalised matches are possible using regular expressions, class hinting and the available generic matchers. The purpose of generalised matchers is to allow arguments be defined in non-explicit terms, e.g. `Mockery::any()` passed to `with()` will match **any** argument in that position.

Mockery's generic matchers do not cover all possibilities but offers optional support for the Hamcrest library of matchers. Hamcrest is a PHP port of the similarly named Java library (which has been ported also to Python, Erlang, etc). By using Hamcrest, Mockery does not need to duplicate Hamcrest's already impressive utility which itself promotes a natural English DSL.

The examples below show Mockery matchers and their Hamcrest equivalent, if there is one. Hamcrest uses functions (no namespacing).

Note: If you don't wish to use the global Hamcrest functions, they are all exposed through the `\Hamcrest\Matchers` class as well, as static methods. Thus, `identicalTo($arg)` is the same as `\Hamcrest\Matchers::identicalTo($arg)`

The most common matcher is the `with()` matcher:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(1):
```

It tells mockery that it should receive a call to the `foo` method with the integer `1` as an argument. In cases like this, Mockery first tries to match the arguments using === (identical) comparison operator. If the argument is a primitive, and if it fails the identical comparison, Mockery does a fallback to the == (equals) comparison operator.

When matching objects as arguments, Mockery only does the strict === comparison, which means only the same `$object` will match:

```
$object = new stdClass();
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive("foo")
    ->with($object);

// Hamcrest equivalent
$mock->shouldReceive("foo")
    ->with(identicalTo($object));
```

A different instance of `stdClass` will **not** match.

Note: The `Mockery\Matcher\MustBe` matcher has been deprecated.

If we need a loose comparison of objects, we can do that using Hamcrest's `equalTo` matcher:

```
$mock->shouldReceive("foo")
    ->with(equalTo(new stdClass));
```

In cases when we don't care about the type, or the value of an argument, just that any argument is present, we use `any()`:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive("foo")
    ->with(\Mockery::any());

// Hamcrest equivalent
$mock->shouldReceive("foo")
    ->with(anything())
```

Anything and everything passed in this argument slot is passed unconstrained.

Validating Types and Resources

The `type()` matcher accepts any string which can be attached to `is_` to form a valid type check.

To match any PHP resource, we could do the following:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive("foo")
    ->with(\Mockery::type('resource'));

// Hamcrest equivalents
$mock->shouldReceive("foo")
    ->with(resourceValue());
$mock->shouldReceive("foo")
    ->with(typeOf('resource'));
```

It will return a `true` from an `is_resource()` call, if the provided argument to the method is a PHP resource. For example, `\Mockery::type('float')` or Hamcrest's `floatValue()` and `typeOf('float')` checks use

`is_float()`, and `\Mockery::type('callable')` or Hamcrest's `callable()` uses `is_callable()`.

The `type()` matcher also accepts a class or interface name to be used in an `instanceof` evaluation of the actual argument. Hamcrest uses `anInstanceOf()`.

A full list of the type checkers is available at php.net or browse Hamcrest's function list in the Hamcrest code.

Complex Argument Validation

If we want to perform a complex argument validation, the `on()` matcher is invaluable. It accepts a closure (anonymous function) to which the actual argument will be passed.

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive("foo")
    ->with(\Mockery::on(closure));
```

If the closure evaluates to (i.e. returns) boolean `true` then the argument is assumed to have matched the expectation.

```
$mock = \Mockery::mock('MyClass');

$mock->shouldReceive('foo')
    ->with(\Mockery::on(function ($argument) {
        if ($argument % 2 == 0) {
            return true;
        }
        return false;
    }));

$mock->foo(4); // matches the expectation
$mock->foo(3); // throws a NoMatchingExpectationException
```

Note: There is no Hamcrest version of the `on()` matcher.

We can also perform argument validation by passing a closure to `withArgs()` method. The closure will receive all arguments passed in the call to the expected method and if it evaluates (i.e. returns) to boolean `true`, then the list of arguments is assumed to have matched the expectation:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive("foo")
    ->withArgs(closure);
```

The closure can also handle optional parameters, so if an optional parameter is missing in the call to the expected method, it doesn't necessary means that the list of arguments doesn't match the expectation.

```
$closure = function ($odd, $even, $sum = null) {
    $result = ($odd % 2 != 0) && ($even % 2 == 0);
    if (!is_null($sum)) {
        return $result && ($odd + $even == $sum);
    }
    return $result;
};

$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')->withArgs($closure);
```

(continues on next page)

(continued from previous page)

```
$mock->foo(1, 2); // It matches the expectation: the optional argument is not needed
$mock->foo(1, 2, 3); // It also matches the expectation: the optional argument pass␣
↪the validation
$mock->foo(1, 2, 4); // It doesn't match the expectation: the optional doesn't pass␣
↪the validation
```

Note: In previous versions, Mockery's `with()` would attempt to do a pattern matching against the arguments, attempting to use the argument as a regular expression. Over time this proved to be not such a great idea, so we removed this functionality, and have introduced `Mockery::pattern()` instead.

If we would like to match an argument against a regular expression, we can use the `\Mockery::pattern()`:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::pattern('/^foo/'));

// Hamcrest equivalent
$mock->shouldReceive('foo')
    ->with(matchesPattern('/^foo/'));
```

The `ducktype()` matcher is an alternative to matching by class type:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::ducktype('foo', 'bar'));
```

It matches any argument which is an object containing the provided list of methods to call.

Note: There is no Hamcrest version of the `ducktype()` matcher.

Capturing Arguments

If we want to perform multiple validations on a single argument, the `capture` matcher provides a streamlined alternative to using the `on()` matcher. It accepts a variable which the actual argument will be assigned.

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive("foo")
    ->with(\Mockery::capture($bar));
```

This will assign *any* argument passed to `foo` to the local `$bar` variable to then perform additional validation using assertions.

Note: The `capture` matcher always evaluates to `true`. As such, we should always perform additional argument validation.

Additional Argument Matchers

The `not()` matcher matches any argument which is not equal or identical to the matcher's parameter:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::not(2));

// Hamcrest equivalent
$mock->shouldReceive('foo')
    ->with(not(2));
```

`anyOf()` matches any argument which equals any one of the given parameters:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::anyOf(1, 2));

// Hamcrest equivalent
$mock->shouldReceive('foo')
    ->with(anyOf(1,2));
```

`notAnyOf()` matches any argument which is not equal or identical to any of the given parameters:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::notAnyOf(1, 2));
```

Note: There is no Hamcrest version of the `notAnyOf()` matcher.

`subset()` matches any argument which is any array containing the given array subset:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::subset(array(0 => 'foo')));
```

This enforces both key naming and values, i.e. both the key and value of each actual element is compared.

Note: There is no Hamcrest version of this functionality, though Hamcrest can check a single entry using `hasEntry()` or `hasKeyValuePair()`.

`contains()` matches any argument which is an array containing the listed values:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::contains(value1, value2));
```

The naming of keys is ignored.

`hasKey()` matches any argument which is an array containing the given key name:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::hasKey(key));
```

`hasValue()` matches any argument which is an array containing the given value:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('foo')
    ->with(\Mockery::hasValue(value));
```

3.1.4 Alternative shouldReceive Syntax

As of Mockery 1.0.0, we support calling methods as we would call any PHP method, and not as string arguments to Mockery `should*` methods.

The two Mockery methods that enable this are `allows()` and `expects()`.

Allows

We use `allows()` when we create stubs for methods that return a predefined return value, but for these method stubs we don't care how many times, or if at all, were they called.

```
$mock = \Mockery::mock('MyClass');
$mock->allows([
    'name_of_method_1' => 'return value',
    'name_of_method_2' => 'return value',
]);
```

This is equivalent with the following `shouldReceive` syntax:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive([
    'name_of_method_1' => 'return value',
    'name_of_method_2' => 'return value',
]);
```

Note that with this format, we also tell Mockery that we don't care about the arguments to the stubbed methods.

If we do care about the arguments, we would do it like so:

```
$mock = \Mockery::mock('MyClass');
$mock->allows()
    ->name_of_method_1($arg1)
    ->andReturn('return value');
```

This is equivalent with the following `shouldReceive` syntax:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method_1')
    ->with($arg1)
    ->andReturn('return value');
```

Expects

We use `expects()` when we want to verify that a particular method was called:

```
$mock = \Mockery::mock('MyClass');
$mock->expects()
    ->name_of_method_1($arg1)
    ->andReturn('return value');
```

This is equivalent with the following `shouldReceive` syntax:

```
$mock = \Mockery::mock('MyClass');
$mock->shouldReceive('name_of_method_1')
    ->once()
    ->with($arg1)
    ->andReturn('return value');
```

By default `expects()` sets up an expectation that the method should be called once and once only. If we expect more than one call to the method, we can change that expectation:

```
$mock = \Mockery::mock('MyClass');
$mock->expects()
    ->name_of_method_1($arg1)
    ->twice()
    ->andReturn('return value');
```

3.1.5 Spies

Spies are a type of test doubles, but they differ from stubs or mocks in that, that the spies record any interaction between the spy and the System Under Test (SUT), and allow us to make assertions against those interactions after the fact.

Creating a spy means we don't have to set up expectations for every method call the double might receive during the test, some of which may not be relevant to the current test. A spy allows us to make assertions about the calls we care about for this test only, reducing the chances of over-specification and making our tests more clear.

Spies also allow us to follow the more familiar Arrange-Act-Assert or Given-When-Then style within our tests. With mocks, we have to follow a less familiar style, something along the lines of Arrange-Expect-Act-Assert, where we have to tell our mocks what to expect before we act on the SUT, then assert that those expectations were met:

```
// arrange
$mock = \Mockery::mock('MyDependency');
$sut = new MyClass($mock);

// expect
$mock->shouldReceive('foo')
    ->once()
    ->with('bar');

// act
$sut->callFoo();

// assert
\Mockery::close();
```

Spies allow us to skip the expect part and move the assertion to after we have acted on the SUT, usually making our tests more readable:

```
// arrange
$spy = \Mockery::spy('MyDependency');
$sut = new MyClass($spy);

// act
$sut->callFoo();
```

(continues on next page)

(continued from previous page)

```
// assert
$spy->shouldHaveReceived()
    ->foo()
    ->with('bar');
```

On the other hand, spies are far less restrictive than mocks, meaning tests are usually less precise, as they let us get away with more. This is usually a good thing, they should only be as precise as they need to be, but while spies make our tests more intent-revealing, they do tend to reveal less about the design of the SUT. If we're having to setup lots of expectations for a mock, in lots of different tests, our tests are trying to tell us something - the SUT is doing too much and probably should be refactored. We don't get this with spies, they simply ignore the calls that aren't relevant to them.

Another downside to using spies is debugging. When a mock receives a call that it wasn't expecting, it immediately throws an exception (failing fast), giving us a nice stack trace or possibly even invoking our debugger. With spies, we're simply asserting calls were made after the fact, so if the wrong calls were made, we don't have quite the same just in time context we have with the mocks.

Finally, if we need to define a return value for our test double, we can't do that with a spy, only with a mock object.

Note: This documentation page is an adaption of the blog post titled "Mockery Spies", published by Dave Marshall on his blog. Dave is the original author of spies in Mockery.

Spies Reference

To verify that a method was called on a spy, we use the `shouldHaveReceived()` method:

```
$spy->shouldHaveReceived('foo');
```

To verify that a method was **not** called on a spy, we use the `shouldNotHaveReceived()` method:

```
$spy->shouldNotHaveReceived('foo');
```

We can also do argument matching with spies:

```
$spy->shouldHaveReceived('foo')
    ->with('bar');
```

Argument matching is also possible by passing in an array of arguments to match:

```
$spy->shouldHaveReceived('foo', ['bar']);
```

Although when verifying a method was not called, the argument matching can only be done by supplying the array of arguments as the 2nd argument to the `shouldNotHaveReceived()` method:

```
$spy->shouldNotHaveReceived('foo', ['bar']);
```

This is due to Mockery's internals.

Finally, when expecting calls that should have been received, we can also verify the number of calls:

```
$spy->shouldHaveReceived('foo')
    ->with('bar')
    ->twice();
```

Alternative shouldReceive syntax

As of Mockery 1.0.0, we support calling methods as we would call any PHP method, and not as string arguments to Mockery `should*` methods.

In cases of spies, this only applies to the `shouldHaveReceived()` method:

```
$spy->shouldHaveReceived()
    ->foo('bar');
```

We can set expectation on number of calls as well:

```
$spy->shouldHaveReceived()
    ->foo('bar')
    ->twice();
```

Unfortunately, due to limitations we can't support the same syntax for the `shouldNotHaveReceived()` method.

3.1.6 Instance Mocking

Instance mocking means that a statement like:

```
$obj = new \MyNamespace\Foo;
```

...will actually generate a mock object. This is done by replacing the real class with an instance mock (similar to an alias mock), as with mocking public methods. The alias will import its expectations from the original mock of that type (note that the original is never verified and should be ignored after its expectations are setup). This lets you intercept instantiation where you can't simply inject a replacement object.

As before, this does not prevent a require statement from including the real class and triggering a fatal PHP error. It's intended for use where autoloading is the primary class loading mechanism.

3.1.7 Creating Partial Mocks

Partial mocks are useful when we only need to mock several methods of an object leaving the remainder free to respond to calls normally (i.e. as implemented). Mockery implements three distinct strategies for creating partials. Each has specific advantages and disadvantages so which strategy we use will depend on our own preferences and the source code in need of mocking.

We have previously talked a bit about *Partial Test Doubles*, but we'd like to expand on the subject a bit here.

1. Runtime partial test doubles
2. Generated partial test doubles
3. Proxied Partial Mock

Runtime partial test doubles

A runtime partial test double, also known as a passive partial mock, is a kind of a default state of being for a mocked object.

```
$mock = \Mockery::mock('MyClass')->makePartial();
```

With a runtime partial, we assume that all methods will simply defer to the parent class (`MyClass`) original methods unless a method call matches a known expectation. If we have no matching expectation for a specific method call, that call is deferred to the class being mocked. Since the division between mocked and unmocked calls depends entirely on the expectations we define, there is no need to define which methods to mock in advance.

See the cookbook entry on *Big Parent Class* for an example usage of runtime partial test doubles.

Generated Partial Test Doubles

A generated partial test double, also known as a traditional partial mock, defines ahead of time which methods of a class are to be mocked and which are to be left unmocked (i.e. callable as normal). The syntax for creating traditional mocks is:

```
$mock = \Mockery::mock('MyClass[foo,bar]');
```

In the above example, the `foo()` and `bar()` methods of MyClass will be mocked but no other MyClass methods are touched. We will need to define expectations for the `foo()` and `bar()` methods to dictate their mocked behaviour.

Don't forget that we can pass in constructor arguments since unmocked methods may rely on those!

```
$mock = \Mockery::mock('MyNamespace\MyClass[foo]', array($arg1, $arg2));
```

See the *Constructor Arguments* section to read up on them.

Note: Even though we support generated partial test doubles, we do not recommend using them.

Proxied Partial Mock

A proxied partial mock is a partial of last resort. We may encounter a class which is simply not capable of being mocked because it has been marked as final. Similarly, we may find a class with methods marked as final. In such a scenario, we cannot simply extend the class and override methods to mock - we need to get creative.

```
$mock = \Mockery::mock(new MyClass);
```

Yes, the new mock is a Proxy. It intercepts calls and reroutes them to the proxied object (which we construct and pass in) for methods which are not subject to any expectations. Indirectly, this allows us to mock methods marked final since the Proxy is not subject to those limitations. The tradeoff should be obvious - a proxied partial will fail any typehint checks for the class being mocked since it cannot extend that class.

Special Internal Cases

All mock objects, with the exception of Proxied Partials, allows us to make any expectation call to the underlying real class method using the `passthru()` expectation call. This will return values from the real call and bypass any mocked return queue (which can simply be omitted obviously).

There is a fourth kind of partial mock reserved for internal use. This is automatically generated when we attempt to mock a class containing methods marked final. Since we cannot override such methods, they are simply left unmocked. Typically, we don't need to worry about this but if we really really must mock a final method, the only possible means is through a Proxied Partial Mock. SplFileInfo, for example, is a common class subject to this form of automatic internal partial since it contains public final methods used internally.

3.1.8 Mocking Protected Methods

By default, Mockery does not allow mocking protected methods. We do not recommend mocking protected methods, but there are cases when there is no other solution.

For those cases we have the `shouldAllowMockingProtectedMethods()` method. It instructs Mockery to specifically allow mocking of protected methods, for that one class only:

```
class MyClass
{
    protected function foo()
    {
    }
}

$mock = \Mockery::mock('MyClass')
    ->shouldAllowMockingProtectedMethods();
$mock->shouldReceive('foo');
```

3.1.9 Mocking Public Properties

Mockery allows us to mock properties in several ways. One way is that we can set a public property and its value on any mock object. The second is that we can use the expectation methods `set()` and `andSet()` to set property values if that expectation is ever met.

You can read more about *Setting Public Properties*.

Note: In general, Mockery does not support mocking any magic methods since these are generally not considered a public API (and besides it is a bit difficult to differentiate them when you badly need them for mocking!). So please mock virtual properties (those relying on `__get()` and `__set()`) as if they were actually declared on the class.

3.1.10 Mocking Public Static Methods

Static methods are not called on real objects, so normal mock objects can't mock them. Mockery supports class aliased mocks, mocks representing a class name which would normally be loaded (via autoloading or a require statement) in the system under test. These aliases block that loading (unless via a require statement - so please use autoloading!) and allow Mockery to intercept static method calls and add expectations for them.

See the *Aliasing* section for more information on creating aliased mocks, for the purpose of mocking public static methods.

3.1.11 Preserving Pass-By-Reference Method Parameter Behaviour

PHP Class method may accept parameters by reference. In this case, changes made to the parameter (a reference to the original variable passed to the method) are reflected in the original variable. An example:

```
class Foo
{

    public function bar(&$a)
    {
        $a++;
```

(continues on next page)

(continued from previous page)

```
    }

}

$baz = 1;
$foo = new Foo;
$foo->bar($baz);

echo $baz; // will echo the integer 2
```

In the example above, the variable `$baz` is passed by reference to `Foo::bar()` (notice the `&` symbol in front of the parameter?). Any change `bar()` makes to the parameter reference is reflected in the original variable, `$baz`.

Mockery handles references correctly for all methods where it can analyse the parameter (using `Reflection`) to see if it is passed by reference. To mock how a reference is manipulated by the class method, we can use a closure argument matcher to manipulate it, i.e. `\Mockery::on()` - see the *Complex Argument Validation* chapter.

There is an exception for internal PHP classes where Mockery cannot analyse method parameters using `Reflection` (a limitation in PHP). To work around this, we can explicitly declare method parameters for an internal class using `\Mockery\Configuration::setInternalClassMethodParamMap()`.

Here's an example using `MongoCollection::insert()`. `MongoCollection` is an internal class offered by the mongo extension from PECL. Its `insert()` method accepts an array of data as the first parameter, and an optional options array as the second parameter. The original data array is updated (i.e. when a `insert()` pass-by-reference parameter) to include a new `_id` field. We can mock this behaviour using a configured parameter map (to tell Mockery to expect a pass by reference parameter) and a `Closure` attached to the expected method parameter to be updated.

Here's a PHPUnit unit test verifying that this pass-by-reference behaviour is preserved:

```
public function testCanOverrideExpectedParametersOfInternalPHPClassesToPreserveRefs()
{
    \Mockery::getConfiguration()->setInternalClassMethodParamMap(
        'MongoCollection',
        'insert',
        array('&$data', '$options = array()')
    );
    $m = \Mockery::mock('MongoCollection');
    $m->shouldReceive('insert')->with(
        \Mockery::on(function(&$data) {
            if (!is_array($data)) return false;
            $data['_id'] = 123;
            return true;
        }),
        \Mockery::any()
    );

    $data = array('a'=>1,'b'=>2);
    $m->insert($data);

    $this->assertTrue(isset($data['_id']));
    $this->assertEquals(123, $data['_id']);

    \Mockery::resetContainer();
}
```

Protected Methods

When dealing with protected methods, and trying to preserve pass by reference behavior for them, a different approach is required.

```
class Model
{
    public function test(&$data)
    {
        return $this->doTest($data);
    }

    protected function doTest(&$data)
    {
        $data['something'] = 'wrong';
        return $this;
    }
}

class Test extends \PHPUnit\Framework\TestCase
{
    public function testModel()
    {
        $mock = \Mockery::mock('Model[test]')->shouldAllowMockingProtectedMethods();

        $mock->shouldReceive('test')
            ->with(\Mockery::on(function(&$data) {
                $data['something'] = 'wrong';
                return true;
            }));

        $data = array('foo' => 'bar');

        $mock->test($data);
        $this->assertTrue(isset($data['something']));
        $this->assertEquals('wrong', $data['something']);
    }
}
```

This is quite an edge case, so we need to change the original code a little bit, by creating a public method that will call our protected method, and then mock that, instead of the protected method. This new public method will act as a proxy to our protected method.

3.1.12 Mocking Demeter Chains And Fluent Interfaces

Both of these terms refer to the growing practice of invoking statements similar to:

```
$object->foo()->bar()->zebra()->alpha()->selfDestruct();
```

The long chain of method calls isn't necessarily a bad thing, assuming they each link back to a local object the calling class knows. As a fun example, Mockery's long chains (after the first `shouldReceive()` method) all call to the same instance of `\Mockery\Expectation`. However, sometimes this is not the case and the chain is constantly crossing object boundaries.

In either case, mocking such a chain can be a horrible task. To make it easier Mockery supports demeter chain mocking. Essentially, we shortcut through the chain and return a defined value from the final call. For example, let's

assume `selfDestruct()` returns the string "Ten!" to $object (an instance of `CaptainsConsole`). Here's how we could mock it.

```
$mock = \Mockery::mock('CaptainsConsole');
$mock->shouldReceive('foo->bar->zebra->alpha->selfDestruct')->andReturn('Ten!');
```

The above expectation can follow any previously seen format or expectation, except that the method name is simply the string of all expected chain calls separated by `->`. Mockery will automatically setup the chain of expected calls with its final return values, regardless of whatever intermediary object might be used in the real implementation.

Arguments to all members of the chain (except the final call) are ignored in this process.

3.1.13 Dealing with Final Classes/Methods

One of the primary restrictions of mock objects in PHP, is that mocking classes or methods marked final is hard. The final keyword prevents methods so marked from being replaced in subclasses (subclassing is how mock objects can inherit the type of the class or object being mocked).

The simplest solution is to implement an interface in your final class and typehint against / mock this.

However this may not be possible in some third party libraries. Mockery does allow creating "proxy mocks" from classes marked final, or from classes with methods marked final. This offers all the usual mock object goodness but the resulting mock will not inherit the class type of the object being mocked, i.e. it will not pass any instanceof comparison. Methods marked as final will be proxied to the original method, i.e., final methods can't be mocked.

We can create a proxy mock by passing the instantiated object we wish to mock into `\Mockery::mock()`, i.e. Mockery will then generate a Proxy to the real object and selectively intercept method calls for the purposes of setting and meeting expectations.

See the *Partial Test Doubles* chapter, the subsection about proxied partial test doubles.

3.1.14 PHP Magic Methods

PHP magic methods which are prefixed with a double underscore, e.g. `__set()`, pose a particular problem in mocking and unit testing in general. It is strongly recommended that unit tests and mock objects do not directly refer to magic methods. Instead, refer only to the virtual methods and properties these magic methods simulate.

Following this piece of advice will ensure we are testing the real API of classes and also ensures there is no conflict should Mockery override these magic methods, which it will inevitably do in order to support its role in intercepting method calls and properties.

3.1.15 PHPUnit Integration

Mockery was designed as a simple-to-use *standalone* mock object framework, so its need for integration with any testing framework is entirely optional. To integrate Mockery, we need to define a `tearDown()` method for our tests containing the following (we may use a shorter `\Mockery` namespace alias):

```
public function tearDown() {
    \Mockery::close();
}
```

This static call cleans up the Mockery container used by the current test, and run any verification tasks needed for our expectations.

For some added brevity when it comes to using Mockery, we can also explicitly use the Mockery namespace with a shorter alias. For example:

```
use \Mockery as m;

class SimpleTest extends \PHPUnit\Framework\TestCase
{
    public function testSimpleMock() {
        $mock = m::mock('simplemock');
        $mock->shouldReceive('foo')->with(5, m::any())->once()->andReturn(10);

        $this->assertEquals(10, $mock->foo(5));
    }

    public function tearDown() {
        m::close();
    }
}
```

Mockery ships with an autoloader so we don't need to litter our tests with `require_once()` calls. To use it, ensure Mockery is on our `include_path` and add the following to our test suite's `Bootstrap.php` or `TestHelper.php` file:

```
require_once 'Mockery/Loader.php';
require_once 'Hamcrest/Hamcrest.php';

$loader = new \Mockery\Loader;
$loader->register();
```

If we are using Composer, we can simplify this to including the Composer generated autoloader file:

```
require __DIR__ . '/../vendor/autoload.php'; // assuming vendor is one directory up
```

Caution: Prior to Hamcrest 1.0.0, the `Hamcrest.php` file name had a small "h" (i.e. `hamcrest.php`). If upgrading Hamcrest to 1.0.0 remember to check the file name is updated for all your projects.)

To integrate Mockery into PHPUnit and avoid having to call the close method and have Mockery remove itself from code coverage reports, have your test case extends the `\Mockery\Adapter\Phpunit\MockeryTestCase`:

```
class MyTest extends \Mockery\Adapter\Phpunit\MockeryTestCase
{

}
```

An alternative is to use the supplied trait:

```
class MyTest extends \PHPUnit\Framework\TestCase
{
    use \Mockery\Adapter\Phpunit\MockeryPHPUnitIntegration;
}
```

Extending `MockeryTestCase` or using the `MockeryPHPUnitIntegration` trait is **the recommended way** of integrating Mockery with PHPUnit, since Mockery 1.0.0.

PHPUnit listener

Before the 1.0.0 release, Mockery provided a PHPUnit listener that would call `Mockery::close()` for us at the end of a test. This has changed significantly since the 1.0.0 version.

Now, Mockery provides a PHPUnit listener that makes tests fail if `Mockery::close()` has not been called. It can help identify tests where we've forgotten to include the trait or extend the `MockeryTestCase`.

If we are using PHPUnit's XML configuration approach, we can include the following to load the `TestListener`:

```
<listeners>
    <listener class="\Mockery\Adapter\Phpunit\TestListener"></listener>
</listeners>
```

Make sure Composer's or Mockery's autoloader is present in the bootstrap file or we will need to also define a "file" attribute pointing to the file of the `TestListener` class.

If we are creating the test suite programmatically we may add the listener like this:

```
// Create the suite.
$suite = new PHPUnit\Framework\TestSuite();

// Create the listener and add it to the suite.
$result = new PHPUnit\Framework\TestResult();
$result->addListener(new \Mockery\Adapter\Phpunit\TestListener());

// Run the tests.
$suite->run($result);
```

Caution: PHPUnit provides a functionality that allows tests to run in a separated process, to ensure better isolation. Mockery verifies the mocks expectations using the `Mockery::close()` method, and provides a PHPUnit listener, that automatically calls this method for us after every test.

However, this listener is not called in the right process when using PHPUnit's process isolation, resulting in expectations that might not be respected, but without raising any `Mockery\Exception`. To avoid this, we cannot rely on the supplied Mockery PHPUnit `TestListener`, and we need to explicitly call `Mockery::close`. The easiest solution to include this call in the `tearDown()` method, as explained previously.

- *Creating Test Doubles*
- *Expectation Declarations*
- *Argument Validation*
- *Alternative shouldReceive Syntax*
- *Spies*
- *Creating Partial Mocks*
- *Mocking Protected Methods*
- *Mocking Public Properties*
- *Mocking Public Static Methods*
- *Preserving Pass-By-Reference Method Parameter Behaviour*
- *Mocking Demeter Chains And Fluent Interfaces*
- *Dealing with Final Classes/Methods*

- *PHP Magic Methods*
- *PHPUnit Integration*
- *Creating Test Doubles*
- *Expectation Declarations*
- *Argument Validation*
- *Alternative shouldReceive Syntax*
- *Spies*
- *Creating Partial Mocks*
- *Mocking Protected Methods*
- *Mocking Public Properties*
- *Mocking Public Static Methods*
- *Preserving Pass-By-Reference Method Parameter Behaviour*
- *Mocking Demeter Chains And Fluent Interfaces*
- *Dealing with Final Classes/Methods*
- *PHP Magic Methods*
- *PHPUnit Integration*

CHAPTER 4

Mockery

Learn about Mockery's configuration, reserved method names, exceptions...

4.1 Mockery

4.1.1 Mockery Global Configuration

To allow for a degree of fine-tuning, Mockery utilises a singleton configuration object to store a small subset of core behaviours. The three currently present include:

- Option to allow/disallow the mocking of methods which do not actually exist fulfilled (i.e. unused)
- Setter/Getter for added a parameter map for internal PHP class methods (`Reflection` cannot detect these automatically)
- Option to drive if quick definitions should define a stub or a mock with an 'at least once' expectation.

By default, the first behaviour is enabled. Of course, there are situations where this can lead to unintended consequences. The mocking of non-existent methods may allow mocks based on real classes/objects to fall out of sync with the actual implementations, especially when some degree of integration testing (testing of object wiring) is not being performed.

You may allow or disallow this behaviour (whether for whole test suites or just select tests) by using the following call:

```
\Mockery::getConfiguration()->allowMockingNonExistentMethods(bool);
```

Passing a true allows the behaviour, false disallows it. It takes effect immediately until switched back. If the behaviour is detected when not allowed, it will result in an Exception being thrown at that point. Note that disallowing this behaviour should be carefully considered since it necessarily removes at least some of Mockery's flexibility.

The other two methods are:

```
\Mockery::getConfiguration()->setInternalClassMethodParamMap($class, $method, array
↪$paramMap)
\Mockery::getConfiguration()->getInternalClassMethodParamMap($class, $method)
```

These are used to define parameters (i.e. the signature string of each) for the methods of internal PHP classes (e.g. SPL, or PECL extension classes like ext/mongo's MongoCollection. Reflection cannot analyse the parameters of internal classes. Most of the time, you never need to do this. It's mainly needed where an internal class method uses pass-by-reference for a parameter - you MUST in such cases ensure the parameter signature includes the `&` symbol correctly as Mockery won't correctly add it automatically for internal classes. Note that internal class parameter overriding is not available in PHP 8. This is because incompatible signatures have been reclassified as fatal errors.

Finally there is the possibility to change what a quick definition produces. By default quick definitions create stubs but you can change this behaviour by asking Mockery to use 'at least once' expectations.

```
\Mockery::getConfiguration()->getQuickDefinitions()->shouldBeCalledAtLeastOnce(bool)
```

Passing a true allows the behaviour, false disallows it. It takes effect immediately until switched back. By doing so you can avoid the proliferating of quick definitions that accumulate overtime in your code since the test would fail in case the 'at least once' expectation is not fulfilled.

Disabling reflection caching

Mockery heavily uses "reflection" to do it's job. To speed up things, Mockery caches internally the information it gathers via reflection. In some cases, this caching can cause problems.

The **only** known situation when this occurs is when PHPUnit's `--static-backup` option is used. If you use `--static-backup` and you get an error that looks like the following:

```
Error: Internal error: Failed to retrieve the reflection object
```

We suggest turning off the reflection cache as so:

```
\Mockery::getConfiguration()->disableReflectionCache();
```

Turning it back on can be done like so:

```
\Mockery::getConfiguration()->enableReflectionCache();
```

In no other situation should you be required turn this reflection cache off.

4.1.2 Mockery Exceptions

Mockery throws three types of exceptions when it cannot verify a mock object.

1. `\Mockery\Exception\InvalidCountException`
2. `\Mockery\Exception\InvalidOrderException`
3. `\Mockery\Exception\NoMatchingExpectationException`

You can capture any of these exceptions in a try…catch block to query them for specific information which is also passed along in the exception message but is provided separately from getters should they be useful when logging or reformatting output.

MockeryExceptionInvalidCountException

The exception class is used when a method is called too many (or too few) times and offers the following methods:

- `getMock()` - return actual mock object
- `getMockName()` - return the name of the mock object
- `getMethodName()` - return the name of the method the failing expectation is attached to
- `getExpectedCount()` - return expected calls
- `getExpectedCountComparative()` - returns a string, e.g. <= used to compare to actual count
- `getActualCount()` - return actual calls made with given argument constraints

MockeryExceptionInvalidOrderException

The exception class is used when a method is called outside the expected order set using the `ordered()` and `globally()` expectation modifiers. It offers the following methods:

- `getMock()` - return actual mock object
- `getMockName()` - return the name of the mock object
- `getMethodName()` - return the name of the method the failing expectation is attached to
- `getExpectedOrder()` - returns an integer represented the expected index for which this call was expected
- `getActualOrder()` - return the actual index at which this method call occurred.

MockeryExceptionNoMatchingExpectationException

The exception class is used when a method call does not match any known expectation. All expectations are uniquely identified in a mock object by the method name and the list of expected arguments. You can disable this exception and opt for returning NULL from all unexpected method calls by using the earlier mentioned shouldIgnoreMissing() behaviour modifier. This exception class offers the following methods:

- `getMock()` - return actual mock object
- `getMockName()` - return the name of the mock object
- `getMethodName()` - return the name of the method the failing expectation is attached to
- `getActualArguments()` - return actual arguments used to search for a matching expectation

4.1.3 Reserved Method Names

As you may have noticed, Mockery uses a number of methods called directly on all mock objects, for example `shouldReceive()`. Such methods are necessary in order to setup expectations on the given mock, and so they cannot be implemented on the classes or objects being mocked without creating a method name collision (reported as a PHP fatal error). The methods reserved by Mockery are:

- `shouldReceive()`
- `shouldNotReceive()`
- `allows()`
- `expects()`
- `shouldAllowMockingMethod()`

- `shouldIgnoreMissing()`
- `asUndefined()`
- `shouldAllowMockingProtectedMethods()`
- `makePartial()`
- `byDefault()`
- `shouldHaveReceived()`
- `shouldHaveBeenCalled()`
- `shouldNotHaveReceived()`
- `shouldNotHaveBeenCalled()`

In addition, all mocks utilise a set of added methods and protected properties which cannot exist on the class or object being mocked. These are far less likely to cause collisions. All properties are prefixed with `_mockery` and all method names with `mockery_`.

4.1.4 Gotchas!

Mocking objects in PHP has its limitations and gotchas. Some functionality can't be mocked or can't be mocked YET! If you locate such a circumstance, please please (pretty please with sugar on top) create a new issue on GitHub so it can be documented and resolved where possible. Here is a list to note:

1. Classes containing public `__wakeup()` methods can be mocked but the mocked `__wakeup()` method will perform no actions and cannot have expectations set for it. This is necessary since Mockery must serialize and unserialize objects to avoid some `__construct()` insanity and attempting to mock a `__wakeup()` method as normal leads to a `BadMethodCallException` being thrown.
2. Mockery has two scenarios where real classes are replaced: Instance mocks and alias mocks. Both will generate PHP fatal errors if the real class is loaded, usually via a require or include statement. Only use these two mock types where autoloading is in place and where classes are not explicitly loaded on a per-file basis using `require()`, `require_once()`, etc.
3. Internal PHP classes are not entirely capable of being fully analysed using `Reflection`. For example, `Reflection` cannot reveal details of expected parameters to the methods of such internal classes. As a result, there will be problems where a method parameter is defined to accept a value by reference (Mockery cannot detect this condition and will assume a pass by value on scalars and arrays). If references as internal class method parameters are needed, you should use the `\Mockery\Configuration::setInternalClassMethodParamMap()` method. Note, however that internal class parameter overriding is not available in PHP 8 since incompatible signatures have been reclassified as fatal errors.
4. Creating a mock implementing a certain interface with incorrect case in the interface name, and then creating a second mock implementing the same interface, but this time with the correct case, will have undefined behavior due to PHP's `class_exists` and related functions being case insensitive. Using the `::class` keyword in PHP can help you avoid these mistakes.

The gotchas noted above are largely down to PHP's architecture and are assumed to be unavoidable. But - if you figure out a solution (or a better one than what may exist), let us know!

- *Mockery Global Configuration*
- *Mockery Exceptions*
- *Reserved Method Names*
- *Gotchas!*

- *Mockery Global Configuration*
- *Mockery Exceptions*
- *Reserved Method Names*
- *Gotchas!*

CHAPTER 5

Cookbook

Want to learn some easy tips and tricks? Take a look at the cookbook articles!

5.1 Cookbook

5.1.1 Default Mock Expectations

Often in unit testing, we end up with sets of tests which use the same object dependency over and over again. Rather than mocking this class/object within every single unit test (requiring a mountain of duplicate code), we can instead define reusable default mocks within the test case's `setup()` method. This even works where unit tests use varying expectations on the same or similar mock object.

How this works, is that you can define mocks with default expectations. Then, in a later unit test, you can add or fine-tune expectations for that specific test. Any expectation can be set as a default using the `byDefault()` declaration.

5.1.2 Detecting Mock Objects

Users may find it useful to check whether a given object is a real object or a simulated Mock Object. All Mockery mocks implement the `\Mockery\MockInterface` interface which can be used in a type check.

```
assert($mightBeMocked instanceof \Mockery\MockInterface);
```

5.1.3 Not Calling the Original Constructor

When creating generated partial test doubles, Mockery mocks out only the method which we specifically told it to. This means that the original constructor of the class we are mocking will be called.

In some cases this is not a desired behavior, as the constructor might issue calls to other methods, or other object collaborators, and as such, can create undesired side-effects in the application's environment when running the tests.

If this happens, we need to use runtime partial test doubles, as they don't call the original constructor.

```
class MyClass
{
    public function __construct()
    {
        echo "Original constructor called." . PHP_EOL;
        // Other side-effects can happen...
    }
}

// This will print "Original constructor called."
$mock = \Mockery::mock('MyClass[foo]');
```

A better approach is to use runtime partial doubles:

```
class MyClass
{
    public function __construct()
    {
        echo "Original constructor called." . PHP_EOL;
        // Other side-effects can happen...
    }
}

// This will not print anything
$mock = \Mockery::mock('MyClass')->makePartial();
$mock->shouldReceive('foo');
```

This is one of the reason why we don't recommend using generated partial test doubles, but if possible, always use the runtime partials.

Read more about *Partial Test Doubles*.

Note: The way generated partial test doubles work, is a BC break. If you use a really old version of Mockery, it might behave in a way that the constructor is not being called for these generated partials. In the case if you upgrade to a more recent version of Mockery, you'll probably have to change your tests to use runtime partials, instead of generated ones.

This change was introduced in early 2013, so it is highly unlikely that you are using a Mockery from before that, so this should not be an issue.

5.1.4 Mocking Hard Dependencies (new Keyword)

One prerequisite to mock hard dependencies is that the code we are trying to test uses autoloading.

Let's take the following code for an example:

```
<?php
namespace App;
class Service
{
    function callExternalService($param)
    {
        $externalService = new Service\External($version = 5);
        $externalService->sendSomething($param);
        return $externalService->getSomething();
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The way we can test this without doing any changes to the code itself is by creating *instance mocks* by using the `overload` prefix.

```
<?php
namespace AppTest;
use Mockery as m;
class ServiceTest extends \PHPUnit_Framework_TestCase
{
    public function testCallingExternalService()
    {
        $param = 'Testing';

        $externalMock = m::mock('overload:App\Service\External');
        $externalMock->shouldReceive('sendSomething')
            ->once()
            ->with($param);
        $externalMock->shouldReceive('getSomething')
            ->once()
            ->andReturn('Tested!');

        $service = new \App\Service();

        $result = $service->callExternalService($param);

        $this->assertSame('Tested!', $result);
    }
}
```

If we run this test now, it should pass. Mockery does its job and our `App\Service` will use the mocked external service instead of the real one.

The problem with this is when we want to, for example, test the `App\Service\External` itself, or if we use that class somewhere else in our tests.

When Mockery overloads a class, because of how PHP works with files, that overloaded class file must not be included otherwise Mockery will throw a "class already exists" exception. This is where autoloading kicks in and makes our job a lot easier.

To make this possible, we'll tell PHPUnit to run the tests that have overloaded classes in separate processes and to not preserve global state. That way we'll avoid having the overloaded class included more than once. Of course this has its downsides as these tests will run slower.

Our test example from above now becomes:

```
<?php
namespace AppTest;
use Mockery as m;
/**
 * @runTestsInSeparateProcesses
 * @preserveGlobalState disabled
 */
class ServiceTest extends \PHPUnit_Framework_TestCase
{
    public function testCallingExternalService()
```

(continues on next page)

(continued from previous page)

```
    {
        $param = 'Testing';

        $externalMock = m::mock('overload:App\Service\External');
        $externalMock->shouldReceive('sendSomething')
            ->once()
            ->with($param);
        $externalMock->shouldReceive('getSomething')
            ->once()
            ->andReturn('Tested!');

        $service = new \App\Service();

        $result = $service->callExternalService($param);

        $this->assertSame('Tested!', $result);
    }
}
```

Testing the constructor arguments of hard Dependencies

Sometimes we might want to ensure that the hard dependency is instantiated with particular arguments. With overloaded mocks, we can set up expectations on the constructor.

```
<?php
namespace AppTest;
use Mockery as m;
/**
 * @runTestsInSeparateProcesses
 * @preserveGlobalState disabled
 */
class ServiceTest extends \PHPUnit_Framework_TestCase
{
    public function testCallingExternalService()
    {
        $externalMock = m::mock('overload:App\Service\External');
        $externalMock->allows('sendSomething');
        $externalMock->shouldReceive('__construct')
            ->once()
            ->with(5);

        $service = new \App\Service();
        $result = $service->callExternalService($param);
    }
}
```

Note: For more straightforward and single-process tests oriented way check *Mocking class within class*.

Note: This cookbook entry is an adaption of the blog post titled "Mocking hard dependencies with Mockery", published by Robert Basic on his blog.

5.1.5 Class Constants

When creating a test double for a class, Mockery does not create stubs out of any class constants defined in the class we are mocking. Sometimes though, the non-existence of these class constants, setup of the test, and the application code itself, it can lead to undesired behavior, and even a PHP error: `PHP Fatal error: Uncaught Error: Undefined class constant 'FOO' in ...`

While supporting class constants in Mockery would be possible, it does require an awful lot of work, for a small number of use cases.

Named Mocks

We can, however, deal with these constants in a way supported by Mockery - by using *Named Mocks*.

A named mock is a test double that has a name of the class we want to mock, but under it is a stubbed out class that mimics the real class with canned responses.

Lets look at the following made up, but not impossible scenario:

```
class Fetcher
{
    const SUCCESS = 0;
    const FAILURE = 1;

    public static function fetch()
    {
        // Fetcher gets something for us from somewhere...
        return self::SUCCESS;
    }
}

class MyClass
{
    public function doFetching()
    {
        $response = Fetcher::fetch();

        if ($response == Fetcher::SUCCESS) {
            echo "Thanks!" . PHP_EOL;
        } else {
            echo "Try again!" . PHP_EOL;
        }
    }
}
```

Our `MyClass` calls a `Fetcher` that fetches some resource from somewhere - maybe it downloads a file from a remote web service. Our `MyClass` prints out a response message depending on the response from the `Fetcher::fetch()` call.

When testing `MyClass` we don't really want `Fetcher` to go and download random stuff from the internet every time we run our test suite. So we mock it out:

```
// Using alias: because fetch is called statically!
\Mockery::mock('alias:Fetcher')
    ->shouldReceive('fetch')
    ->andReturn(0);
```

(continues on next page)

(continued from previous page)

```
$myClass = new MyClass();
$myClass->doFetching();
```

If we run this, our test will error out with a nasty `PHP Fatal error: Uncaught Error: Undefined class constant 'SUCCESS' in ...`

Here's how a `namedMock()` can help us in a situation like this.

We create a stub for the `Fetcher` class, stubbing out the class constants, and then use `namedMock()` to create a mock named `Fetcher` based on our stub:

```
class FetcherStub
{
    const SUCCESS = 0;
    const FAILURE = 1;
}

\Mockery::namedMock('Fetcher', 'FetcherStub')
    ->shouldReceive('fetch')
    ->andReturn(0);

$myClass = new MyClass();
$myClass->doFetching();
```

This works because under the hood, Mockery creates a class called `Fetcher` that extends `FetcherStub`.

The same approach will work even if `Fetcher::fetch()` is not a static dependency:

```
class Fetcher
{
    const SUCCESS = 0;
    const FAILURE = 1;

    public function fetch()
    {
        // Fetcher gets something for us from somewhere...
        return self::SUCCESS;
    }
}

class MyClass
{
    public function doFetching($fetcher)
    {
        $response = $fetcher->fetch();

        if ($response == Fetcher::SUCCESS) {
            echo "Thanks!" . PHP_EOL;
        } else {
            echo "Try again!" . PHP_EOL;
        }
    }
}
```

And the test will have something like this:

```
class FetcherStub
{
    const SUCCESS = 0;
    const FAILURE = 1;
}

$mock = \Mockery::mock('Fetcher', 'FetcherStub')
$mock->shouldReceive('fetch')
    ->andReturn(0);

$myClass = new MyClass();
$myClass->doFetching($mock);
```

Constants Map

Another way of mocking class constants can be with the use of the constants map configuration.

Given a class with constants:

```
class Fetcher
{
    const SUCCESS = 0;
    const FAILURE = 1;

    public function fetch()
    {
        // Fetcher gets something for us from somewhere...
        return self::SUCCESS;
    }
}
```

It can be mocked with:

```
\Mockery::getConfiguration()->setConstantsMap([
    'Fetcher' => [
        'SUCCESS' => 'success',
        'FAILURE' => 'fail',
    ]
]);

$mock = \Mockery::mock('Fetcher');
var_dump($mock::SUCCESS); // (string) 'success'
var_dump($mock::FAILURE); // (string) 'fail'
```

5.1.6 Big Parent Class

In some application code, especially older legacy code, we can come across some classes that extend a "big parent class" - a parent class that knows and does too much:

```
class BigParentClass
{
    public function doesEverything()
    {
        // sets up database connections
```

(continues on next page)

(continued from previous page)

```
        // writes to log files
    }
}

class ChildClass extends BigParentClass
{
    public function doesOneThing()
    {
        // but calls on BigParentClass methods
        $result = $this->doesEverything();
        // does something with $result
        return $result;
    }
}
```

We want to test our `ChildClass` and its `doesOneThing` method, but the problem is that it calls on `BigParentClass::doesEverything()`. One way to handle this would be to mock out **all** of the dependencies `BigParentClass` has and needs, and then finally actually test our `doesOneThing` method. It's an awful lot of work to do that.

What we can do, is to do something... unconventional. We can create a runtime partial test double of the `ChildClass` itself and mock only the parent's `doesEverything()` method:

```
$childClass = \Mockery::mock('ChildClass')->makePartial();
$childClass->shouldReceive('doesEverything')
    ->andReturn('some result from parent');

$childClass->doesOneThing(); // string("some result from parent");
```

With this approach we mock out only the `doesEverything()` method, and all the unmocked methods are called on the actual `ChildClass` instance.

5.1.7 Complex Argument Matching With Mockery::on

When we need to do a more complex argument matching for an expected method call, the `\Mockery::on()` matcher comes in really handy. It accepts a closure as an argument and that closure in turn receives the argument passed in to the method, when called. If the closure returns `true`, Mockery will consider that the argument has passed the expectation. If the closure returns `false`, or a "falsey" value, the expectation will not pass.

The `\Mockery::on()` matcher can be used in various scenarios — validating an array argument based on multiple keys and values, complex string matching...

Say, for example, we have the following code. It doesn't do much; publishes a post by setting the `published` flag in the database to `1` and sets the `published_at` to the current date and time:

```
<?php
namespace Service;
class Post
{
    public function __construct($model)
    {
        $this->model = $model;
    }

    public function publishPost($id)
```

(continues on next page)

(continued from previous page)

```
    {
        $saveData = [
            'post_id' => $id,
            'published' => 1,
            'published_at' => gmdate('Y-m-d H:i:s'),
        ];
        $this->model->save($saveData);
    }
}
```

In a test we would mock the model and set some expectations on the call of the `save()` method:

```
<?php
$postId = 42;

$modelMock = \Mockery::mock('Model');
$modelMock->shouldReceive('save')
    ->once()
    ->with(\Mockery::on(function ($argument) use ($postId) {
        $postIdIsSet = isset($argument['post_id']) && $argument['post_id'] ===
↪$postId;
        $publishedFlagIsSet = isset($argument['published']) && $argument['published']␣
↪=== 1;
        $publishedAtIsSet = isset($argument['published_at']);

        return $postIdIsSet && $publishedFlagIsSet && $publishedAtIsSet;
    }));

$service = new \Service\Post($modelMock);
$service->publishPost($postId);

\Mockery::close();
```

The important part of the example is inside the closure we pass to the `\Mockery::on()` matcher. The `$argument` is actually the `$saveData` argument the `save()` method gets when it is called. We check for a couple of things in this argument:

- the post ID is set, and is same as the post ID we passed in to the `publishPost()` method,
- the `published` flag is set, and is `1`, and
- the `published_at` key is present.

If any of these requirements is not satisfied, the closure will return `false`, the method call expectation will not be met, and Mockery will throw a `NoMatchingExpectationException`.

Note: This cookbook entry is an adaption of the blog post titled "Complex argument matching in Mockery", published by Robert Basic on his blog.

5.1.8 Mocking class within class

Imagine a case where you need to create an instance of a class and use it within the same method:

```
// Point.php
<?php
namespace App;

class Point {
    public function setPoint($x, $y) {
        echo "Point (" . $x . ", " . $y . ")" . PHP_EOL;
    }
}

// Rectangle.php
<?php
namespace App;
use App\Point;

class Rectangle {
    public function create($x1, $y1, $x2, $y2) {
        $a = new Point();
        $a->setPoint($x1, $y1);

        $b = new Point();
        $b->setPoint($x2, $y1);

        $c = new Point();
        $c->setPoint($x2, $y2);

        $d = new Point();
        $d->setPoint($x1, $y2);

        $this->draw([$a, $b, $c, $d]);
    }

    public function draw($points) {
        echo "Do something with the points";
    }
}
```

And that you want to test that a logic in `Rectangle->create()` calls properly each used thing - in this case calls `Point->setPoint()`, but `Rectangle->draw()` does some graphical stuff that you want to avoid calling.

You set the mocks for `App\Point` and `App\Rectangle`:

```
<?php
class MyTest extends PHPUnit\Framework\TestCase {
    public function testCreate() {
        $point = Mockery::mock("App\Point");
        // check if our mock is called
        $point->shouldReceive("setPoint")->andThrow(Exception::class);

        $rect = Mockery::mock("App\Rectangle")->makePartial();
        $rect->shouldReceive("draw");

        $rect->create(0, 0, 100, 100);  // does not throw exception
        Mockery::close();
    }
}
```

and the test does not work. Why? The mocking relies on the class not being present yet, but the class is autoloaded

therefore the mock alone for `App\Point` is useless which you can see with `echo` being executed.

Mocks however work for the first class in the order of loading i.e. `App\Rectangle`, which loads the `App\Point` class. In more complex example that would be a single point that initiates the whole loading (`use Class`) such as:

```
A        // main loading initiator
|- B     // another loading initiator
|  |-E
|  +-G
|
|- C     // another loading initiator
|  +-F
|
+- D
```

That basically means that the loading prevents mocking and for each such a loading initiator there needs to be implemented a workaround. Overloading is one approach, however it pollutes the global state. In this case we try to completely avoid the global state pollution with custom `new Class()` behavior per loading initiator and that can be mocked easily in few critical places.

That being said, although we can't stop loading, we can return mocks. Let's look at `Rectangle->create()` method:

```
class Rectangle {
    public function newPoint() {
        return new Point();
    }

    public function create($x1, $y1, $x2, $y2) {
        $a = $this->newPoint();
        $a->setPoint($x1, $y1);
        ...
    }
    ...
}
```

We create a custom function to encapsulate `new` keyword that would otherwise just use the autoloaded class `App\Point` and in our test we mock that function so that it returns our mock:

```
<?php
class MyTest extends PHPUnit\Framework\TestCase {
    public function testCreate() {
        $point = Mockery::mock("App\Point");
        // check if our mock is called
        $point->shouldReceive("setPoint")->andThrow(Exception::class);

        $rect = Mockery::mock("App\Rectangle")->makePartial();
        $rect->shouldReceive("draw");

        // pass the App\Point mock into App\Rectangle as an alternative
        // to using new App\Point() in-place.
        $rect->shouldReceive("newPoint")->andReturn($point);

        $this->expectException(Exception::class);
        $rect->create(0, 0, 100, 100);
        Mockery::close();
    }
}
```

If we run this test now, it should pass. For more complex cases we'd find the next loader in the program flow and proceed with wrapping and passing mock instances with predefined behavior into already existing classes.

- *Default Mock Expectations*
- *Detecting Mock Objects*
- *Not Calling the Original Constructor*
- *Mocking Hard Dependencies (new Keyword)*
- *Class Constants*
- *Big Parent Class*
- *Complex Argument Matching With Mockery::on*
- *Default Mock Expectations*
- *Detecting Mock Objects*
- *Not Calling the Original Constructor*
- *Mocking Hard Dependencies (new Keyword)*
- *Class Constants*
- *Big Parent Class*
- *Complex Argument Matching With Mockery::on*

Index

A

Alternative shouldReceive Syntax, 31
Argument Validation, 26

C

Cookbook
    Big Parent Class, 55
    Class Constants, 52
    Complex Argument Matching With Mockery::on, 56
    Default Mock Expectations, 49
    Detecting Mock Objects, 49
    Mocking class within class, 57
    Mocking Hard Dependencies, 50
    Not Calling the Original Constructor, 49

E

Expectations, 19

G

Getting Started
    Simple Example, 7

I

Installation, 5

M

Mockery
    Configuration, 43
    Exceptions, 44
    Gotchas, 46
Mocking
    Demeter Chains, 38
    Final Classes/Methods, 39
    Instance, 34
    Magic Methods, 39
    Partial Mocks, 34
    Protected Methods, 35
    Public Properties, 36
    Public Static Methods, 36

P

Pass-By-Reference Method Parameter Behaviour, 36
PHPUnit Integration, 39

Q

Quick Reference, 8

R

Reference
    Creating Test Doubles, 13
    Spies, 32
Reserved Method Names, 45

U

Upgrading, 6