ABOUT HUGH HAMILL

Hugh is a Senior Software Engineer and Certified Scrum Master based in Galway, Ireland. He achieved his B.Sc. in Applied Computing from Waterford Institute of Technology in 2002 and has been working in industry since then. He has worked for a several large blue chip software companies listed on both the NASDAQ and NYSE.

Getting started with Mockito

Posted by: Hugh Hamill in Core Java November 15th, 2015

This article is part of our Academy Course titled Testing with Mockito.

In this course, you will dive into the magic of Mockito. You will learn about Mocks, Spies and Partial Mocks, and their corresponding Stubbing behaviour. You will also see the process of Verification with Test Doubles and Object Matchers. Finally, Test Driven Development (TDD) with Mockito is discussed in order to see how this library fits in the concept of TDD. Check it out here!

Table of Contents

1. Why Mock?
2. Introduction to the Mockito Framework
2.1. A Quick Example of Stubbing
2.2. A Quick Example of Verification
3. Mixing up a Mockito
4. Using Mockito with JUnit
5. Mockito Best Practices

In this tutorial we are going to look at the Mockito Mocking Framework and prepare an Eclipse project to use it by adding it to the classpath.

1. Why Mock?

All of the code we write has a network of interdependencies, it may call into the methods of several other classes which in turn may call yet other methods; indeed this is the intent and power of object oriented programming. Usually at the same time as writing our feature code we

When we unit test our code we want to test it in isolation and we want to test it fast. For the purposes of the unit test we only care about verifying our own code, in the current class under test. Generally we also want to execute our unit tests very regularly, perhaps more than several times per hour when we are refactoring and we are working in our continuous integration environment.

This is when all our interdependencies become an issue. We might end up executing code in another class that has a bug that causes our unit test to fail. Imagine a class which we use to read user details from a database, what happens if there's no database present when we want to run our unit tests? Imagine a class which calls several remote web services, what if they're down or take a long time to respond? Our unit tests could fail due to our dependencies and not because of some issue with the behaviour of our code. This is undesirable.

In addition to this, it might be very difficult to force a specific event or error condition that we want to ensure our code handles correctly. What if we want to test that some class which deserializes an object handles a possible ObjectStreamException properly? What if we want to test all boundary return values from a collaborator? What about ensuring that some calculated value is passed correctly to a collaborator? It might take a lot of coding and a long time to replicate the conditions for our tests, if it is even possible at all.

All these issues simply disappear if we use mocks. Mocks act like a substitute for the classes with which we are collaborating, they take their place and behave exactly how we tell them to behave. Mocks let us pretend that our real collaborators are there, even though they aren't. More importantly mocks can be programmed to return whatever values we want and confirm whatever values are passed to them. Mocks execute instantly and don't require any external resources. Mocks will return what we tell them to return, throw whatever exceptions we want them to throw and will do these things over and over, on demand. They let us test only the behaviour of our own code, to ensure that our class works, regardless of the behaviour of its collaborators.

There are several mocking frameworks available for Java, each have their own syntax, their own strengths, their own weaknesses. In this tutorial we will be using the Mockito framework, which is one of the more popular mocking frameworks available.

2. Introduction to the Mockito Framework

Mockito is a Mocking Framework that makes it very easy to create mocks for the classes and interfaces with which your class under test interacts. Mockito provides a very simple API for creating mocks and assigning their behaviour. It allows you to very quickly specify expected behaviour and verify interactions with your mocks.

Mockito has essentially two phases, one or both of which are executed as part of unit tests:

- Stubbing
- Verification

Stubbing is the process of specifying the behaviour of our mocks. It is how we tell Mockito what we want to happen when we interact with our mocks. Stubbing lets us address some of the concerns which we outlined in the first section – it makes it simple to create all the possible conditions for our tests. It let's us control the responses of our mocks, including forcing them to return any value we want, or throw any exception we want. It allows us to code different behaviours under different conditions. Stubbing lets us control exactly what the mock will do.

Verification is the process of verifying interactions with our mocks. It lets us determine how our mocks were called, and how many times. It lets us look at the arguments of our mocks to make sure they are as expected. Verification lets us address the other concerns mentioned in the first section – it lets us ensure that exactly the values we expect are passed to our collaborators, and that nothing unexpected happens. Verification lets us determine exactly what happened to the mock.

By tying together these two simple phases we can build extremely flexible and powerful unit tests, encoding complex mock behaviour and complex mock interaction verification with the very simple Mockito API.

Mockito does have some limitations, however, including

- You can't mock final classes
- You can't mock static methods
- You can't mock final methods
- You can't mock equals() or hashCode()

2.1. A Quick Example of Stubbing

Imagine you are writing a class which calls the API of a physical temperature sensor. You want to call the

```
double getDegreesC()
```

method and return one of the following Strings – "Hot", "Mild", "Cold" – based on the value returned from the sensor. It would be very difficult, to say the least, to have your unit tests control the environmental temperature of the room in order to test your functionality. But what if we use Mockito to create a mock which we substitute for the sensor?

Now we can write code like this in our unit test:

```java
when(sensor.getDegreesC()).thenReturn(15.0);
```

This tells Mockito that when the mock sensor receives a call to

```
getDegreesC()
```

it should then return the value 15.0.

want to make sure that the observer's
```
notify()
```
method was called once as part of your method execution. You could set some boolean in the observer and check it from your unit test, but this means altering some production code, code you might not even own. What about Mockito, what if the observer is a mock?

Now we can write code like this in our unit test:

```
verify(observer).notify();
```

This tells Mockito that the
```
notify()
```
method must be called once and only once, otherwise the unit test should fail.

3. Mixing up a Mockito

Now we've learned a bit about the framework let's use it in a project.

If you use Maven then adding Mockito to your project is as simple as adding the following dependency:

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
   </dependency>
```

If you use Gradle then just add the following

```
dependencies {
    testCompile "org.mockito:mockito-all:1.9.5"
}
```

To add Mockito to the classpath of an Eclipse project the old fashioned way grab the latest jar from the Mockito download page (take mockito-all-1.9.5.jar) and download it to your hard drive.

Right click on your eclipse project and select 'Properties' and then select 'Java Build Path' in the left pane and 'Libraries' on the right.

On the 'Libraries' tab click the 'Add External Jars' button and navigate to the mockito-all jar you previously downloaded. Select the jar and it is now added to your project and available to use.

As of the time of writing the latest version of Mockito is 1.9.5 but you should check for updates before you add it to your project.

4. Using Mockito with JUnit

To integrate Mockito into your JUnit test class you can use the provided Test Runner
```
MockitoJUnitRunner
```
. Just annotate your test class with:

```
@RunWith(MockitoJUnitRunner.class)
```

This tells Mockito to take any annotated mocks within the test class and initialise them for mocking. You can then simply annotate any instance variable with
```
@Mock
```
to use it as a mock. Note that you should import
```
org.mockito.Mock
```
and not
```
org.mockito.mockitoannotations.Mock
```
, which is deprecated.

As with anything a full example makes things clearer, we will create a new Test class and within it we'll use Mockito to mock a
```
java.util.List
```
:

```
import java.util.List;
import org.junit.runner.RunWith;
import org.mockito.Mock;
```

```

    @Mock
    private List<String> mockList;

}
```

The

```
@Mock
```

annotation tells Mockito that

```
mockList
```

is to be treated as a mock and the

```
@RunWith(MockitoJUnitRunner.class)
```

tells Mockito to go through all the

```
@Mock
```

annotated members of

```
MyTest
```

and initialize them for Mocking. You don't have to assign any new instance to mockList, this is done under the hood for you by Mockito. With this simple code above mockList is ready to be used as a mock.

Try adding the following imports:

```
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;
import org.junit.Test;
```

And then the following simple test case:

```
@Test
public void test() {
 String expected = "Hello, World!";
 when(mockList.get(0)).thenReturn(expected);

 String actual = mockList.get(0);
 assertEquals(expected, actual);
 System.out.println(actual);
}
```

Here we see that we set up an expectation – that we have a String

```
"Hello, World!"
```

and then we go on to stub the

```
List.get()
```

method of our mock List to return our expected String when the first element of the list is requested.

We then call

```
mockList.get(0)
```

to get our actual value for testing and assert that our actual value is equal to our expected value, and output it to the console for good measure.

At no point have we created a real list, or inserted "Hello, World!" into a list. It is just a mock list and the only functionality it has or knows about is the get() method with an input of 0.

Try changing

```
String actual = mockList.get(0);
```

to

```
String actual = mockList.get(1);
```

and running the test. You will see that

```
actual
```

is now null. The reason is that the only functionality we have stubbed is for calling .get() with an input of 0 – Mockito doesn't know what to do with an input of 1 so it just returns null. In fact any other method of List we call will return null, and any method which returns nothing will effectively act as a no-op. This is powerful control, in a couple of lines of code we have created an implementation of List that does exactly what we want, when we want it every time it's called.

- Mockito does not have a provision for mocking static methods because Mockito encourages object oriented design and dependency injection over procedural code.
- Mockito does not have a provision for mocking private methods because public methods should be black boxes and from the standpoint of testing private methods don't exist.
- Mockito packages and encourages the usage of Hamcrest Matchers, which will be covered in subsequent tutorials.
- Mockito encourages adherence to the Law of Demeter and does not encourage mocking chained methods.
- You should not stub or verify on a mock which is shared between different threads. You may call the methods of a shared mock, however.
- You can't verify the
  ```
  toString()
  ```
  method of a mock, due to the fact that it may be called by the test environment itself, making verification impossible.
- If your test cases use the Given When Then notation you can use the stubbing methods from
  ```
  org.mockito.BDDMockito
  ```
  so that
  ```
  when(mock.method()).thenReturn(something)
  ```
  becomes
  ```
  given(mock.method()).willReturn(something)
  ```
  as it will read nicely in your test format.
- It is possible to use Mockito without the Mockito annotations, however it is much easier and neater to use the annotations so that is what we will do in these tutorials.
- You can 'spy' on any class, including the class under test if your testing requires that you modify the behaviour of a particular method of the class for the purposes of the test. Mockito explicitly recommends that spies should be only used carefully and occasionally, for instance when constrained by dealing with legacy code. This will be covered in a future tutorial.
- In the event that the real call into a spied method could generate an error condition or cannot be called for some other reason Mockito recommends using the do* family of methods for stubbing. This will be covered in a future tutorial.
- Mockito will allow you to use argument matchers in place of real arguments with the limitation that if one argument uses a matcher, all must use matchers. Argument matchers will be covered in a later tutorial but should probably be used sparingly.
- Mockito provides a
  ```
  verifyNoMoreInteractions()
  ```
  method to verify that a particular mock has no more interactions but recommends that it is used very sparingly and only when appropriate.
- Mockito provides the
  ```
  Answer
  ```
  interface to allow for stubbing with callbacks, however it recommends against using it and encourages you to do simple stubbing using the
  ```
  thenReturn()
  ```
  and
  ```
  doThrow()
  ```
  methods. We will cover Answers in a later tutorial.
- If using
  ```
  ArgumentCaptor
  ```
  for argument validation you should use it only in the verification phase, and not the stubbing phase.
  ```
  ArgumentCaptor
  ```
  will be covered in a future tutorial.
- Mockito recommends to use Partial Mocks very carefully, mainly when dealing with legacy code. Well designed code should not require the use of partial mocks.
- Mockito provides a
  ```
  reset()
  ```
  method for resetting your mock in the middle of a test method, however it recommends against using it as it is a code smell that your test may be overly long and complex.

There are more features and practices, but these are the main ones which Mockito tells you to watch out for. We will cover all of the above and more in depth in the coming tutorials.