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.

Mockito Verification

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. What is Verification?
2. Using verify()
2.1. Using the built in Verification Modes
2.2. Creating a custom Verification Mode
2.3. Verification with Parameters
2.4. Verification with Timeout
3. Verifying No Interactions and No More Interactions
4. Verification in Order
5. Argument Captors
6. Conclusion
7. Download the Source Code

1. What is Verification?

behaviour is verification and there are a number of tools which Mockito provides to allow us to do it.

2. Using verify()

The main tool for performing verification in the Mockito toolbox is the

```
org.mockito.Mockito.verify()
```

method. The verify method takes the Mock object as a parameter and returns an instance of the same Class as the Mock, allowing you to call the methods of the Class, which Mockito interprets as a request to verify that there was some interaction with that method.

Let's look again at our printer interface from the previous tutorial.

```
1  public interface Printer {
2
3      void printTestPage();
4
5  }
```

We can create a simple unit test to demonstrate verify using a Mock Printer

```
01 import static org.mockito.Mockito.verify;
02
03 import org.junit.Test;
04 import org.junit.runner.RunWith;
05 import org.mockito.Mock;
06 import org.mockito.runners.MockitoJUnitRunner;
07
08 @RunWith(MockitoJUnitRunner.class)
09 public class PrinterTest {
10
11     @Mock
12     private Printer printer;
13
14     @Test
15     public void simple_interaction_verification() {
16         // Given
17
18         // When
19         printer.printTestPage();
20
21         // Then
22         verify(printer).printTestPage();
23     }
24 }
```

We can see that our unit test firstly calls

```
printer.printTestPage()
```

. This is simulating a possible interaction within a class under test, but to keep things simple we are doing it out in the unit test class. The next call is the call to

```
verify(printer).printTestPage()
```

. This instructs Mockito to check if there has been a single call to the

```
printTestPage()
```

method of the Mock Printer.

Note carefully the syntax of the call, the parameter of

```
verify()
```

is the Mock object, not the method call. If we had put

```
verify(printer.printTestPage())
```

we would have generated a compile error. Contrast this to the given/when syntax in stubbing which takes the form

```
when(mockObject.someMethod()).thenReturn(...)
```

.

If we hadn't called

```
printTestPage()
```

above this call to verify Mockito would have generated a verification error informing us that there was no invocation of

```
printTestPage()
```

, which would look like this:

```
1  Wanted but not invoked:
```

Additionally if we had made a second call to

```
printTestPage()
```

Mockito would have generated a verification error informing us that there were too many invocations of

```
printTestPage()
```

. This error would look like this:

```
1 org.mockito.exceptions.verification.TooManyActualInvocations:
2 printer.printTestPage();
3 Wanted 1 time:
4 -> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification(PrinterTe
5 But was 2 times. Undesired invocation:
6 -> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification(PrinterTe
```

Usefully the error informs us which line of code contained the superfluous invocation – in this case line 22 of

```
PrinterTest.java
```

.

But what if we want multiple interactions with our Mock? Does Mockito support this? Unsurprisingly the answer is Yes!

The

```
verify()
```

method takes a second parameter of type

```
org.mockito.verification.VerificationMode
```

which can be used to provide additional details about the desired interactions with the mock.

2.1. Using the built in Verification Modes

As usual Mockito provides a number of convenient static methods in org.mockito.Mockito for creating VerificationModes, such as:

```
times(int)
```

This will verify that the method was called the input number of times.

```
01 @Test
02 public void simple_interaction_verification_times_1() {
03     // Given
04
05     // When
06     printer.printTestPage();
07
08     // Then
09     verify(printer, times(1)).printTestPage();
10 }
```

Note that

```
verify(mock)
```

is an alias of

```
verify(mock, times(1))
```

.

Of course we can verify multiple interactions using

```
times()
```

```
01 @Test
02 public void simple_interaction_verification_times_3() {
03     // Given
04
05     // When
06     printer.printTestPage();
07     printer.printTestPage();
08     printer.printTestPage();
09
10     // Then
11     verify(printer, times(3)).printTestPage();
12 }
```

This

will generate useful errors when the actual number of invocations doesn't match the expected number.

Not enough invocations:

```
org.mockito.exceptions.verification.TooLittleActualInvocations:
printer.printTestPage();
Wanted 3 times:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_times_3(P
But was 2 times:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_times_3(P
```

Too many invocations:

```
org.mockito.exceptions.verification.TooManyActualInvocations:
printer.printTestPage();
Wanted 3 times:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_times_3(P
But was 4 times. Undesired invocation:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_times_3(P
```

`atLeastOnce()`

,

`atLeast(int)`

This will verify that the method was called at least the given number of times.

```
@Test
public void simple_interaction_verification_atleastonce() {
    // Given

    // When
    printer.printTestPage();
    printer.printTestPage();

    // Then
    verify(printer, atLeastOnce()).printTestPage();
}
```

```
@Test
public void simple_interaction_verification_atleast_2() {
    // Given

    // When
    printer.printTestPage();
    printer.printTestPage();
    printer.printTestPage();

    // Then
    verify(printer, atLeast(2)).printTestPage();
}
```

As usual we get comprehensive error reporting:

```
org.mockito.exceptions.verification.TooLittleActualInvocations:
printer.printTestPage();
Wanted *at least* 2 times:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_atleast_2
But was 1 time:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_atleast_2
```

`atMost(int)`

This will verify that the method was called at most the given number of times.

```
@Test
public void simple_interaction_verification_atmost_3() {
    // Given

    // When
    printer.printTestPage();
    printer.printTestPage();

    // Then
    verify(printer, atMost(3)).printTestPage();
}
```

And the error condition:

```
org.mockito.exceptions.base.MockitoAssertionError:
Wanted at most 3 times but was 4
    at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_atmost_3
```

This will verify that the method was not called.

```
@Test
public void simple_interaction_verification_never() {
    // Given

    // When

    // Then
    verify(printer, never()).printTestPage();
}
```

And the error condition:

```
org.mockito.exceptions.verification.NeverWantedButInvoked:
printer.printTestPage();
Never wanted here:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_never(Pri
But invoked here:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_never(Pri
```

`only()`

This will verify that the method being verified was the only method of the mock called.

```
@Test
public void simple_interaction_verification_only() {
    // Given

    // When
    printer.printTestPage();

    // Then
    verify(printer, only()).printTestPage();
}
```

We can produce an error by adding the following method to our printer interface:

```
void turnOff();
```

And calling it in our test

```
@Test
public void simple_interaction_verification_only_fails() {
    // Given

    // When
    printer.printTestPage();
    printer.turnOff();

    // Then
    verify(printer, only()).printTestPage();
}
```

to give the following error:

```
org.mockito.exceptions.verification.NoInteractionsWanted:
No interactions wanted here:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_only_fail
But found this interaction:
-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_only_fail
***
For your reference, here is the list of all invocations ([?] - means unverified).
1. [?]-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_onl
2. [?]-> at com.javacodegeeks.hughwphamill.mockito.stubbing.PrinterTest.simple_interaction_verification_onl
```

2.2. Creating a custom Verification Mode

You can create your own custom verification mode by implementing the

`org.mockito.verification.VerificationMode`

interface. Note that this is using some classes that don't form part of the public API of Mockito. There are plans to promote them to public API as of the time of writing, but this feature should be used with caution in case the implementation changes before that happens.

`VerificationMode`

exposes a single

`void verify(VerificationData data)`

method which is used to verify that the mock invocations we are interested in happend correctly.

:

`InvocationsFinder`

will return a list of all invocations with the mock of interest.

`InvocationMarker`

can be used to mark the mock invocation as verified.

`Reporter`

exposes a number of shortcut methods for throwing various

`VerificationFailure`

errors.

`InvocationMatcher`

is used in conjunction with

`InvocationsMarker`

to find the desired Invocations if they happened.

We are going to create a

`VerificationMode`

called

`First`

which will verify that a given method was the first invocation on the Mock. We will create a class which implements

`VerificationMode`

and in the verify method we will find all matching invocations and verify two things:

1. The invocation we wanted actually happened, if it did not we will use Reporter to throw a "wanted but not invoked" error.
2. The invocation we wanted was the first invocation on the Mock, if it was not we will throw a new exception with an appropriate message detailing the expected invocation and the actual one.

Lastly we will expose the creating of

`First`

through a static factory method to be consistent with the Mockito syntax.

The class

`First`

looks like this:

```
01 package com.javacodegeeks.hughwphamill.mockito.verification;
02
03 import java.util.Arrays;
04 import java.util.List;
05
06 import org.mockito.exceptions.Reporter;
07 import org.mockito.exceptions.verification.VerificationInOrderFailure;
08 import org.mockito.internal.debugging.LocationImpl;
09 import org.mockito.internal.invocation.InvocationMarker;
10 import org.mockito.internal.invocation.InvocationMatcher;
11 import org.mockito.internal.invocation.InvocationsFinder;
12 import org.mockito.internal.verification.api.VerificationData;
13 import org.mockito.invocation.Invocation;
14 import org.mockito.verification.VerificationMode;
15
16 public class First implements VerificationMode {
17
18     private final InvocationsFinder finder = new InvocationsFinder();
19     private final InvocationMarker marker = new InvocationMarker();
20     private final Reporter reporter = new Reporter();
21
22     public static VerificationMode first() {
23         return new First();
24     }
25
26     @Override
27     public void verify(VerificationData data) {
28         List<Invocation> invocations = data.getAllInvocations();
29         InvocationMatcher matcher = data.getWanted();
30
```

```
        } else if (!sameInvocation(invocations.get(0), chunk.get(0))) {
            reportNotFirst(chunk.get(0), invocations.get(0));
        }

        marker.markVerified(chunk.get(0), matcher);
    }

    private boolean sameInvocation(Invocation left, Invocation right) {
        if (left == right) {
            return true;
        }
        return left.getMock().equals(right.getMock()) && left.getMethod().equals(right.getMethod()) && Arra
    }

    private void reportNotFirst(Invocation wanted, Invocation unwanted) {
        StringBuilder message = new StringBuilder();
        message.append("\\nWanted first:\\n").append(wanted).append("\\n").append(new LocationImpl());
        message.append("\\nInstead got:\\n").append(unwanted).append("\\n").append(unwanted.getLocation()).
        throw new VerificationInOrderFailure(message.toString());

    }
}
```

We can use it in a test case like this:

```
@Test
public void simple_interaction_verification_first() {
    // Given

    // When
    printer.printTestPage();
    printer.turnOff();

    // Then
    verify(printer, first()).printTestPage();
}
```

Or to catch some unexpected behaviour:

```
@Test
public void simple_interaction_verification_first_fails() {
    // Given

    // When
    printer.turnOff();
    printer.printTestPage();

    // Then
    verify(printer, first()).printTestPage();
}
```

Which generates the following error:

```
org.mockito.exceptions.verification.VerificationInOrderFailure:
Wanted first:
printer.printTestPage();
-> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.simple_interaction_verification_first
Instead got:
printer.turnOff();
-> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.simple_interaction_verification_first
```

2.3. Verification with Parameters

We are going to examine verification of methods which take parameters so let's update our

`Printer`

interface to add a new method. This method will simulate printing a String of text and will contain the following parameters:

- String text – The text to be printed.
- Integer copies – The number of copies to be made.
- Boolean collate – True to collate copies.

```
public interface Printer {

    void printTestPage();

    void turnOff();

    void print(String text, Integer copies, Boolean collate);

}
```

Verification with Parameters lets us verify that not only was there an interaction with a

`Mock`

, but what parameters were passed to the Mock. To perform verification with parameters you simply pass the parameters of interest into the Mocked method on the verify call on the Mock.

```
    String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
            + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
    Integer copies = 3;
    Boolean collate = true;

    // When
    printer.print(text, copies, collate);

    // Then
    verify(printer).print(text, copies, collate);
}
```

Note carefully again the syntax of

`verify()`

we are calling

`print()`

on the object returned from the verify method, not directly on the Mock. You can see that simply passing in the values to

`print()`

is enough to perform verification using parameters.

The following test will fail:

```
@Test
public void verificatin_with_actual_parameters_fails() {
    // Given
    String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
            + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
    String text2 = "Ut enim ad minim veniam, quis nostrud exercitation ullamco "
            + "laboris nisi ut aliquip ex ea commodo consequat.";
    Integer copies = 3;
    Boolean collate = true;

    // When
    printer.print(text2, copies, collate);

    // Then
    verify(printer).print(text, copies, collate);
}
```

with the following output:

```
Argument(s) are different! Wanted:
printer.print(
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et
    3,
    true
);
-> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verificatin_with_actual_parameters_fa
Actual invocation has different arguments:
printer.print(
    "Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo conse
    3,
    true
);
```

You can see that Mockito usefully gives you the expected arguments and the actual arguments in the error trace, making it very easy to debug your failing test.

As with simple verification we can use a VerificationMode to do more specific verification when using Parameters. The crucial difference is that the VerificationMode we specify applies only to invocations with the stated parameters. So if we use a verification mode of

`never()`

, for instance, we are stating that the method is never called with the stated parameters, not that it is never called at all.

The following test passes because even though the

`print()`

method is invoked, it is never invoked with the specified parameters.

```
@Test
public void verification_with_actual_parameters_and_verification_mode() {
    // Given
    String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
            + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
    String text2 = "Ut enim ad minim veniam, quis nostrud exercitation ullamco "
            + "laboris nisi ut aliquip ex ea commodo consequat.";
    Integer copies = 3;
    Boolean collate = true;

    // When
    printer.print(text, copies, collate);

    // Then
```

```
Mock
```

we can verify each one individually using multiple calls to

```
verify
```

```
01 @Test
02 public void multiple_verification_with_actual_parameters() {
03     // Given
04     String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
05             + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
06     String text2 = "Ut enim ad minim veniam, quis nostrud exercitation ullamco "
07             + "laboris nisi ut aliquip ex ea commodo consequat.";
08     Integer copies = 3;
09     Boolean collate = true;
10
11     // When
12     printer.print(text, copies, collate);
13     printer.print(text2, copies, collate);
14
15     // Then
16     verify(printer).print(text, copies, collate);
17     verify(printer).print(text2, copies, collate);
18 }
```

A lot of the time we aren't interested or don't know what the actual parameters of the interaction will be, in these instances, just as in the Stubbing phase we can use Argument Matchers to verify interactions.

Look at the following test

```
01 @Test
02 public void verification_with_matchers() {
03     // Given
04     String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
05             + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
06     String text2 = "Ut enim ad minim veniam, quis nostrud exercitation ullamco "
07             + "laboris nisi ut aliquip ex ea commodo consequat.";
08     Integer copies3 = 3;
09     Integer copies4 = 4;
10     Boolean doCollate = true;
11     Boolean doNotCollate = false;
12
13     // When
14     printer.print(text, copies3, doCollate);
15     printer.print(text2, copies4, doNotCollate);
16
17     // Then
18     verify(printer, times(2)).print(anyString(), anyInt(), anyBoolean());
19 }
```

Note that we call

```
printer.print()
```

twice, with completely different parameters each time. We verify both interactions with the Mock on the last line using Argument Matchers. Remember that

```
verify(printer).print()
```

implicitly means we want to verify one and only one interaction with the

```
print()
```

method on the Mock, so we must include the

```
times(2)
```

VerificationMode to ensure we are verifying both interactions with the Mock.

The Argument Matchers used with Verification are the same Argument Matchers used during the Stubbing phase. Please revisit the Stubbing tutorial for a longer list of available Matchers.

As with the Stubbing phase we cannot mix and match Argument Matchers with real values, but what if we didn't care about the text that was passed into the Printer for printing, we were only interested in verifying that there should be 5 collated copies?

In this case, as with Stubbing, we can use the

```
eq()
```

Matcher to verify the real values we are interested, while using

```
anyString()
```

for the text.

```
01 @Test
02 public void verification_with_mixed_matchers() {
```

```
            + "laboris nisi ut aliquip ex ea commodo consequat.";
    Integer copies = 5;
    Boolean collate = true;

    // When
    printer.print(text, copies, collate);
    printer.print(text2, copies, collate);

    // Then
    verify(printer, times(2)).print(anyString(), eq(copies), eq(collate));
}
```

This passes, while the following test will fail

```
@Test
public void verification_with_mixed_matchers_fails() {
    // Given
    String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
            + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
    Integer copies5 = 5;
    Integer copies10 = 10;
    Boolean collate = true;

    // When
    printer.print(text, copies10, collate);

    // Then
    verify(printer).print(anyString(), eq(copies5), eq(collate));
}
```

2.4. Verification with Timeout

Sometimes, when testing multithreaded applications, we want to ensure that certain mock interactions happen within a given timeout period. Mockito provides a

`timeout()`

method to help us achieve this.

Please note that while this feature is available, the Mockito documentation cautions against using it:

"It feels this feature should be used rarely – figure out a better way of testing your multi-threaded system."

So with the health warning out of the way let's look at a couple of examples.

Let's say we have some thread that's going to execute a

`printTestPage()`

and we want to verify that this happens within 100 milliseconds. We can use

`timeout(100)`

to achieve this. It can be passed as the second parameter to

`verify()`

and it returns a VerificationWithTimout which is an extension of VerificationMode.

The following test demonstrates it's usage:

```
@Test
public void verification_with_timeout() {
    // Given

    // When
    Executors.newFixedThreadPool(1).execute(() -> printer.printTestPage());

    // Then
    verify(printer, timeout(100)).printTestPage();
}
```

Here we create a new

`ExecutorService`

using

`Executor`

s which can execute Runnables. We take advantage of Java 8 Lambda expressions to build a new

`Runnable`

on the fly which will execute a call to

`printTestPage()`

. We then call

passing in a timeout of 100ms.

We can look at a failing test now. This time we will use a method reference to generate the Runnable, this is because the body of our Runnable is a bit more complex – it introduces a sleep for 200ms.

```
@Test
public void verification_with_timeout_fails() throws InterruptedException {
    // Given

    // When
    Executors.newFixedThreadPool(1).execute(this::printTestWithSleep);

    // Then
    verify(printer, timeout(100)).printTestPage();
}

private void printTestWithSleep() {
    try {
        Thread.sleep(200L);
        printer.printTestPage();
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

The test fails with a simple 'wanted but not invoked' message.

It is also possible to add VerificationModes to

```
timeout()
```

using methods exposed by the returned

```
VerificationWithTimeout
```

.

```
@Test
public void verification_with_timeout_with_verification_mode() {
    // Given
    int poolsize = 5;

    // When
    ExecutorService service = Executors.newFixedThreadPool(poolsize);
    service.execute(this::printTestWithSleep);
    service.execute(this::printTestWithSleep);
    service.execute(this::printTestWithSleep);

    // Then
    verify(printer, timeout(500).times(3)).printTestPage();
}
```

Here we use the test with sleep to execute

```
printTestPage()
```

3 times, we use an

```
ExecutorService
```

that can run 5 parallel threads so the sleeps happen at the same time, allowing all 3 invocations to occur within the 500ms limit.

We can make the test fail by reducing the number of available threads to 1, forcing the printTestWithSleep calls to execute sequentially and going over the 500ms timeout.

```
@Test
public void verification_with_timeout_with_verification_mode_fails() {
    // Given
    int poolsize = 1;

    // When
    ExecutorService service = Executors.newFixedThreadPool(poolsize);
    service.execute(this::printTestWithSleep);
    service.execute(this::printTestWithSleep);
    service.execute(this::printTestWithSleep);

    // Then
    verify(printer, timeout(500).times(3)).printTestPage();
}
```

The first 2 calls happen within 400ms while the last one will happen after 600ms, causing the timeout of 500ms to fail with the following output:

```
org.mockito.exceptions.verification.TooLittleActualInvocations:
printer.printTestPage();
Wanted 3 times:
-> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verification_with_timeout_with_verifi
But was 2 times:
-> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.printTestWithSleep(PrinterTest.java:3
```

```
never()
```

VerificationMode to ensure that a particular method of a Mock is not invoked, but what about verifying that there are no interactions on a Mock at all?

Mockito provides us with the

```
verifyZeroInteractions()
```

method to do just that. This method uses varargs to allow us to verify no interactions with several mocks in one line of code.

Let's add some other Mock to our test class:

```
1 @Mock
2 private List<String> list;
```

Now we can write the following simplistic test to verify no interactions with the Printer or the List

```
1 @Test
2 public void verify_zero_interactions() {
3     // Given
4
5     // When
6
7     // Then
8     verifyZeroInteractions(printer, list);
9 }
```

As usual, the following test will fail

```
01 @Test
02 public void verify_zero_interactions_fails() {
03     // Given
04
05     // When
06     printer.printTestPage();
07
08     // Then
09     verifyZeroInteractions(printer, list);
10 }
```

With the following output

```
1 org.mockito.exceptions.verification.NoInteractionsWanted:
2 No interactions wanted here:
3 -> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_zero_interactions_fails(Printe
4 But found this interaction:
5 -> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_zero_interactions_fails(Printe
6 Actually, above is the only interaction with this mock.
7     at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_zero_interactions_fails(Print
```

We can also verify that once a certain number of invocations have been verified there are no more interactions with the Mock, using the

```
verifyNoMoreInteractions()
```

method.

```
01 @Test
02 public void verify_no_more_interactions() {
03     // Given
04     String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
05             + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
06     Integer copies = 3;
07     Boolean collate = true;
08
09     // When
10     printer.print(text, copies, collate);
11
12     // Then
13     verify(printer).print(text, copies, collate);
14     verifyNoMoreInteractions(printer);
15 }
```

You can see above that we verify the call to

```
print()
```

and then verify that there are no more interactions with the Mock.

The following test will fail because there was an additional interaction with the mock after the verified call to

```
print()
```

```
01 @Test
02 public void verify_no_more_interactions_fails() {
03     // Given
```

```
08
09      // When
10      printer.print(text, copies, collate);
11      printer.turnOff();
12
13      // Then
14      verify(printer).print(text, copies, collate);
15      verifyNoMoreInteractions(printer);
16  }
```

The failing test generates the following message:

```
1  org.mockito.exceptions.verification.NoInteractionsWanted:
2  No interactions wanted here:
3  -> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_no_more_interactions_fails(Pri
4  But found this interaction:
5  -> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_no_more_interactions_fails(Pri
6  ***
7  For your reference, here is the list of all invocations ([?] - means unverified).
8  1. -> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_no_more_interactions_fails(
9  2. [?]-> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_no_more_interactions_fai
```

4. Verification in Order

Sometimes we want to verify that interactions with our Mocks happened in a particular order. Mockito provides a class called

```
InOrder
```

to help us achieve this.

The first thing we need to do is register the mocks that we want to confirm the invocation order on with the InOrder object. We then execute methods on the Mock object, and then call the

```
verify()
```

method of the

```
InOrder
```

object for each mock method for which we want to confirm the ordered executions, in the order in which we want to verify they happened.

The

```
InOrder.verify()
```

method behaves almost like the standard

```
verify()
```

method, allowing you to pass in VerificationModes, however you can't do Verification With Timeout with InOrder.

Here is an example of verification in order in action:

```
01  @Test
02  public void verify_in_order() {
03      // Given
04      InOrder inOrder = Mockito.inOrder(printer);
05
06      // When
07      printer.printTestPage();
08      printer.turnOff();
09
10      // Then
11      inOrder.verify(printer).printTestPage();
12      inOrder.verify(printer).turnOff();
13  }
```

And the converse failing test:

```
01  @Test
02  public void verify_in_order_fails() {
03      // Given
04      InOrder inOrder = Mockito.inOrder(printer);
05
06      // When
07      printer.turnOff();
08      printer.printTestPage();
09
10      // Then
11      inOrder.verify(printer).printTestPage();
12      inOrder.verify(printer).turnOff();
13  }
```

Which fails with the following error message:

```
1  org.mockito.exceptions.verification.VerificationInOrderFailure:
2  Verification in order failure
3  Wanted but not invoked:
```

```
8 -> at com.javacodegeeks.hughwphamill.mockito.verification.PrinterTest.verify_in_order_fails(PrinterTest.jav
```

You can also Verify In Order across multiple Mocks:

```
01 @Test
02 public void verify_in_order_multiple() {
03     // Given
04     InOrder inOrder = Mockito.inOrder(printer, list);
05
06     // When
07     printer.printTestPage();
08     list.clear();
09     printer.turnOff();
10
11     // Then
12     inOrder.verify(printer).printTestPage();
13     inOrder.verify(list).clear();
14     inOrder.verify(printer).turnOff();
15 }
```

5. Argument Captors

We have looked at using Argument Matchers for verifying invocations with particular parameters but Mockito lets us go further than this, capturing the parameters which were passed into the invocation and performing asserts directly on them. This is very useful for verifying log in your class which is performed on objects which will be passed to collaborators. The facility for doing this is a class called

```
ArgumentCaptor
```

and an annotation called

```
@Captor
```

.

Let's make a new class in our model called

```
PrinterDiagnostics
```

. It will contain a Printer and expose a method called

```
diagnosticPrint
```

, which will have the same parameters as

```
Printer.print()
```

and add some diagnostic information to the text being printed.

```
01 package com.javacodegeeks.hughwphamill.mockito.verification;
02
03 public class PrinterDiagnostics {
04
05     private Printer printer;
06
07     public PrinterDiagnostics(Printer printer) {
08         this.printer = printer;
09     }
10
11     public void diagnosticPrint(String text, Integer copies, Boolean collate) {
12         StringBuilder diagnostic = new StringBuilder();
13         diagnostic.append("** Diagnostic Print **\\n");
14         diagnostic.append("*** Copies: ").append(copies).append(" ***\\n");
15         diagnostic.append("*** Collate: ").append(collate).append(" ***\\n");
16         diagnostic.append("**********************\\n\\n");
17
18         printer.print(new StringBuilder().append(diagnostic).append(text).toString(), copies, collate);
19     }
20 }
```

We'll create a new JUnit test to test this class, using a Mock

```
Printer
```

and an

```
ArgumentCaptor
```

which we'll use to verify the input to the Printer.

Here's the skeleton of the JUnit test:

```
01 package com.javacodegeeks.hughwphamill.mockito.verification;
02
03 import org.junit.Before;
04 import org.junit.runner.RunWith;
05 import org.mockito.ArgumentCaptor;
06 import org.mockito.Captor;
07 import org.mockito.Mock;
```

```
12
13      private PrinterDiagnostics diagnostics;
14      @Mock
15      private Printer printer;
16      @Captor
17      private ArgumentCaptor<String> textCaptor;
18
19      @Before
20      public void setUp() throws Exception {
21          diagnostics = new PrinterDiagnostics(printer);
22      }
23  }
```

Here we see that we create an instance of the class under test, diagnostics, a Mock to represent the printer, printer, and an ArgumentCaptor for String arguments to capture the text input to the printer called textCaptor. You can see that we annotated the ArgumentCaptor with the

```
@Captor
```

annotation; Mockito will automatically instantiate the ArgumentCaptor for us because we used the annotation.

You can also see that ArgumentCaptor is a Generic type, we create an ArgumentCaptor with Type Argument String in this case because we are going to be capturing the text argument, which is a String. If we were capturing the collate parameter we might have created

```
ArgumentCaptor collateCaptor
```

.

In our

```
@Before
```

method we simply create a new

```
PrinterDiagnostics
```

., injecting our mock Printer through its constructor.

Now let's create our test. We want to ensure two things:

1. The number of copies is added to the input text.
2. The state of the collate parameter is added to the input text.
3. The original text is maintained.

We could also want to verify the formatting and the asterisks in the real world, but for now let's content ourselves with verifying the two criteria above.

In the test we will initialize the test data, execute the call to

```
diagnosticPrint()
```

and then use

```
verify()
```

in conjunction with the

```
capture()
```

method of ArgumentCaptor to capture the text argument. We will then do necessary asserts on the captured String to verify the behaviour we expect by using the

```
getValue()
```

method to retrieve the captured text.

```
01  @Test
02  public void verify_diagnostic_information_added_to_text() {
03      // Given
04      String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
05              + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
06      Integer copies = 3;
07      Boolean collate = true;
08      String expectedCopies = "Copies: " + copies;
09      String expectedCollate = "Collate: " + collate;
10
11      // When
12      diagnostics.diagnosticPrint(text, copies, collate);
13
14      // Then
15      verify(printer).print(textCaptor.capture(), eq(copies), eq(collate));
16      assertTrue(textCaptor.getValue().contains(expectedCopies));
17      assertTrue(textCaptor.getValue().contains(expectedCollate));
18      assertTrue(textCaptor.getValue().contains(text));
19  }
```

Note that

```
capture()
```

matcher to ensure we pass through the expected copies and collate parameters.

If there are multiple invocations on the mocked method we can use the

```
getValues()
```

method of

```
ArgumentCaptor
```

to get a List of all the Strings which were passed through as the text parameter in each call.

Let's create a new method in PrinterDiagnostics which will perform a diagnostic print of a single collated copy as well as the original print:

```
1  public void diagnosticAndOriginalPrint(String text, Integer copies, Boolean collate) {
2      diagnosticPrint(text, copies, collate);
3      printer.print(text, copies, collate);
4  }
```

We can now test this with the following test method:

```
01 @Test
02 public void verify_diagnostic_information_added_to_text_and_original_print() {
03     // Given
04     String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, "
05             + "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
06     Integer copies = 3;
07     Boolean collate = true;
08     String expectedCopies = "Copies: " + copies;
09     String expectedCollate = "Collate: " + collate;
10
11     // When
12     diagnostics.diagnosticAndOriginalPrint(text, copies, collate);
13
14     // Then
15     verify(printer, times(2)).print(textCaptor.capture(), eq(copies), eq(collate));
16     List<String> texts = textCaptor.getAllValues();
17     assertEquals(2, texts.size());
18
19     // First captured text is Diagnostic Print
20     assertTrue(texts.get(0).contains(expectedCopies));
21     assertTrue(texts.get(0).contains(expectedCollate));
22     assertTrue(texts.get(0).contains(text));
23
24     // Second captured text is normal Print
25     assertFalse(texts.get(1).contains(expectedCopies));
26     assertFalse(texts.get(1).contains(expectedCollate));
27     assertEquals(text, texts.get(1));
28 }
```

Note that we have to use

```
times(2)
```

in our verification because we are expecting to invoke the

```
print()
```

method twice.

ArgumentCaptors are particularly useful when our parameters are complex objects or are created by the code under test. You can easily capture the argument and do any type of verification and validation you need on it.

6. Conclusion

We have looked in detail at the verification phase of Mockito. We have examined the ways which we can verify behaviour out of the box, create our own Verification Modes and use Argument Captors for doing further more complex assertions on our Data.

In the next tutorial we will examine how the Hamcrest Matcher library lets us take our test verification even further, allowing us to do very fine grained behavioural validation.

7. Download the Source Code

This was a lesson on Mockito Verification. You may download the source code here: mockito3-verification

Tagged with: MOCKITO TESTING

Do you want to know how to develop your skillset to become a Java Rockstar?