Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

ANDROID | JAVA | JVM LANGUAGES | SOFTWARE DEVELOPMENT | AGILE | CAREER | COMMUNICATIONS | DEVOPS | META JCG

Home » Java » Core Java » Mocks, Spies, Partial Mocks and Stubbing

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.

Mocks, Spies, Partial Mocks and Stubbing

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

2. Mock, Stub, Spy – What's in a name?

A lot of terminology in mocking is used interchangeably and as both verbs and nouns. We will give a definition of these terms now to avoid confusion in the future.

- **Mock (noun)** – An object which acts as a double for another object.
- **Mock (verb)** – To create a mock object or stub a method.
- **Spy (noun)** – An object which decorates an existing object and allows for stubbing of methods of that object and verification of calls into that object.
- **Spy (verb)** – To create and use a Spy object.
- **Stub (noun)** – An object which can provide 'canned answers' when it's methods are called.
- **Stub (verb)** – To create a canned answer.
- **Partial Mock, Partial Stub (verb)** – Another term for a spy with some of it's methods stubbed.

Technically, Mockito is a Test Spy Framework rather than a Mocking Framework, because it allows us to create spies and verify behaviour, as well as creating mock objects with stubbed behaviour.

As we saw in the last tutorial, we can use the

```
when().thenReturn()
```

methods to stub behaviour of a given interface or class. We will now look at all the ways that we can provide stubs for Mocks and Spies.

3. Stubbing a void method

Given the following interface:

```java
public interface Printer {

    void printTestPage();

}
```

And the following simplistic String buffer based 'word processor' class which uses it:

```java
public class StringProcessor {

    private Printer printer;
    private String currentBuffer;

    public StringProcessor(Printer printer) {
        this.printer = printer;
    }

    public Optional<String> statusAndTest() {
        printer.printTestPage();
        return Optional.ofNullable(currentBuffer);
    }

}
```

We want to write a test method which will test that the current buffer is absent after construction and handle the printing of the test page.

Here is our test class:

```java
public class StringProcessorTest {

    private Printer printer;

    @Test
    public void internal_buffer_should_be_absent_after_construction() {
        // Given
        StringProcessor processor = new StringProcessor(printer);

        // When
        Optional<String> actualBuffer = processor.statusAndTest();

        // Then
        assertFalse(actualBuffer.isPresent());
    }
}
```

We know that

```
statusAndTest()
```

will involve a call to the

```
printTestPage()
```

and that the

```
printer
```

reference is not initialized so we will end up with a

```
NullPointerException
```

if we execute this test. In order to avoid this we simply need to annotate the test class to tell JUnit to run it with Mockito and annotate the Printer as a mock to tell mockito to create a mock for it.

```
@RunWith(MockitoJUnitRunner.class)
public class StringProcessorTest {

    @Mock
    private Printer printer;

    @Test
    public void internal_buffer_should_be_absent_after_construction() {
        // Given
        StringProcessor processor = new StringProcessor(printer);

        // When
        Optional<String> actualBuffer = processor.statusAndTest();

        // Then
        assertFalse(actualBuffer.isPresent());
    }

}
```

Now we can execute our test and Mockito will create an implementation of Printer for us and assign an instance of it to the printer variable. We will no longer get a NullPointerException.

But what if

```
Printer
```

was a class that actually did some work, like printing a physical test page. What if we had chosen to

```
@Spy
```

on it instead of creating a

```
@Mock
```

? Remember a Spy will call the real methods of the spied upon Class unless they are stubbed. We would want to avoid doing anything real when the method was called. Let's make a simple implementation of Printer:

```
public class SysoutPrinter implements Printer {

    @Override
    public void printTestPage() {
        System.out.println("This is a test page");
    }

}
```

And add it as a Spy to our test class and add a new method to test using it:

```
@Spy
    private SysoutPrinter sysoutPrinter;

@Test
    public void internal_buffer_should_be_absent_after_construction_sysout() {
        // Given
        StringProcessor processor = new StringProcessor(sysoutPrinter);

        // When
        Optional<String> actualBuffer = processor.statusAndTest();

        // Then
        assertFalse(actualBuffer.isPresent());
    }
```

If you execute this test now you will see the following output on the console:

```
This is a test page
```

This confirms that our test case is actually executing the real method of the

```
SysoutPrinter
```

class due to the fact that it is a Spy and not a Mock. If the class actually executed a real physical print of a test page this would be highly undesirable!

When we are doing a partial mock or Spy we can stub the method that is called to ensure that nothing happens in it using

```
org.mockito.Mockito.doNothing()
```

Let’s add the following import and test:

```
import static org.mockito.Mockito.*;

@Test
    public void internal_buffer_should_be_absent_after_construction_sysout_with_donothing() {
        // Given
        StringProcessor processor = new StringProcessor(sysoutPrinter);
        doNothing().when(sysoutPrinter).printTestPage();

        // When
        Optional<String> actualBuffer = processor.statusAndTest();

        // Then
        assertFalse(actualBuffer.isPresent());
    }
```

Note the chaining of the methods

```
doNothing.when(sysoutPrinter).printTestPage()
```

: this tells Mockito that when the void method

```
printTestPage
```

of the

```
@Spy
```

```
sysoutPrinter
```

is called that the real method should not be executed and nothing should be done instead. Now when we execute this test we see no output to the screen.

What if we expand our Printer interface to throw a new

```
PrinterNotConnectedException
```

exception if the physical printer is not connected. How can we test this scenario?

First of all let’s create the new, very simple, exception class.

```
public class PrinterNotConnectedException extends Exception {

    private static final long serialVersionUID = -6643301294924639178L;

}
```

And modify our interface to throw it:

```
void printTestPage() throws PrinterNotConnectedException;
```

We also need to modify

```
StringProcessor
```

to do something with the exception if it’s thrown. For the sake of simplicity we will just throw the exception back out to the calling class.

```
public Optional<String> statusAndTest() throws PrinterNotConnectedException
```

Now we want to test that the exception is passed up to the calling class, so we have to force the Printer to throw it. In a similar way to

```
doNothing()
```

we can use

```
doThrow
```

to force the exception.

Let’s add the following test:

```
@Test(expected = PrinterNotConnectedException.class)
    public void printer_not_connected_exception_should_be_thrown_up_the_stack() throws Exception {
        // Given
        StringProcessor processor = new StringProcessor(printer);
        doThrow(new PrinterNotConnectedException()).when(printer).printTestPage();

        // When
        Optional<String> actualBuffer = processor.statusAndTest();

        // Then
        assertFalse(actualBuffer.isPresent());
    }
```

Here we see that we can use

to throw any kind of Exception we want. In this case we are throwing

`PrinterNotConnectedException`

which will satisfy our Test.

Now that we've learned how to stub void methods, let's look at returning some data.

4. Stubbing return values

Let's start to create a Data Access Object for persisting and retrieving Customer objects from a database. This DAO will use the enterprise java

`EntityManager`

interface under the hood to do the actual DB interactions.

In order to use

`EntityManager`

we will use the Hibernate implementation of JPA 2.0, add the following dependency to your pom.xml:

```xml
<dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.0-api</artifactId>
            <version>1.0.1.Final</version>
        </dependency>
```

Now we will create a simple Customer entity to represent the Customer being persisted.

```java
@Entity
public class Customer {

    @Id @GeneratedValue
    private long id;
    private String name;
    private String address;

    public Customer() {

    }

    public Customer(long id, String name, String address) {
        super();
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

We will now create a skeleton DAO which has uses

`@PersistenceContext`

to configure an injected

`EntityManager`

. We don't need to worry about using the Java Persistence Architecture (JPA) or how it works – we will be using Mockito to bypass it completely, but this serves as a good real world example of Mockito in action.

```java
public class CustomerDAO {

    @PersistenceContext
    EntityManager em;

}
```

We will be adding basic Retrieve and Update functionality to our DAO and testing it using Mockito.

To start with the Retrieve method – we will pass in an ID and return the appropriate Customer from the DB, if they exist.

```java
public Optional<Customer> findById(long id) throws Exception {
        return Optional.ofNullable(em.find(Customer.class, id));
    }
```

Here we use Java

`Optional`

to avoid having to do null checks on the results.

Now we can add tests to test this method where the customer is found, and the customer is not found – we will stub the

`find()`

method to return an appropriate Optional in each case, using the Mockito methods

`org.mockito.Mockito.when`

and

`thenReturn()`

Lets create our Test class as follows (

`import static org.mockito.Mockito.*;`

for Mockito methods):

```java
@RunWith(MockitoJUnitRunner.class)
public class CustomerDAOTest {

    private CustomerDAO dao;

    @Mock
    private EntityManager mockEntityManager;

    @Before
    public void setUp() throws Exception {
        dao = new CustomerDAO(mockEntityManager);
    }

    @Test
    public void finding_existing_customer_should_return_customer() throws Exception {
        // Given
        long expectedId = 10;
        String expectedName = "John Doe";
        String expectedAddress = "21 Main Street";
        Customer expectedCustomer = new Customer(expectedId, expectedName, expectedAddress);

        when(mockEntityManager.find(Customer.class, expectedId)).thenReturn(expectedCustomer);

        // When
        Optional<Customer> actualCustomer = dao.findById(expectedId);

        // Then
        assertTrue(actualCustomer.isPresent());
        assertEquals(expectedId, actualCustomer.get().getId());
        assertEquals(expectedName, actualCustomer.get().getName());
        assertEquals(expectedAddress, actualCustomer.get().getAddress());
    }
}
```

We see the usual boilerplate for enabling mockito, mocking the

`EntityManger`

and injecting it into the class under test. Let's look at the test method.

The first lines involve creating a

`Customer`

with known expected values, we then see the call to Mockito telling it to return this customer when the

`EntityManager.find()`

method is called with the specific input parameters we give it. We then do the actual execution of the

`findById()`

method and a group of asserts to ensure we got back the expected values.

This demonstrates the powerful, elegant syntax of Mockito. It almost reads like plain English. When the

```
find()
```

method of the

```
mockEntityManager
```

object is called with the specific inputs

```
Customer.class
```

and

```
expectedId
```

, then return the

```
expectedCustomer
```

object.

If you invoke a Mock with parameters that you haven't told it to expect then it will just return null, as the following test demonstrates:

```
@Test
    public void invoking_mock_with_unexpected_argument_returns_null() throws Exception {
        // Given
        long expectedId = 10L;
        long unexpectedId = 20L;
        String expectedName = "John Doe";
        String expectedAddress = "21 Main Street";
        Customer expectedCustomer = new Customer(expectedId, expectedName, expectedAddress);

        when(mockEntityManager.find(Customer.class, expectedId)).thenReturn(expectedCustomer);

        // When
        Optional<Customer> actualCustomer = dao.findById(unexpectedId);

        // Then
        assertFalse(actualCustomer.isPresent());
    }
```

You can also stub a Mock several different times to achieve different behaviours depending on inputs. Let's get the Mock to return a different customer depending on the input ID:

```
@Test
    public void invoking_mock_with_different_argument_returns_different_customers() throws Exception {
        // Given
        long expectedId1 = 10L;
        String expectedName1 = "John Doe";
        String expectedAddress1 = "21 Main Street";
        Customer expectedCustomer1 = new Customer(expectedId1, expectedName1, expectedAddress1);

        long expectedId2 = 20L;
        String expectedName2 = "Jane Deer";
        String expectedAddress2 = "46 High Street";
        Customer expectedCustomer2 = new Customer(expectedId2, expectedName2, expectedAddress2);

        when(mockEntityManager.find(Customer.class, expectedId1)).thenReturn(expectedCustomer1);
        when(mockEntityManager.find(Customer.class, expectedId2)).thenReturn(expectedCustomer2);

        // When
        Optional<Customer> actualCustomer1 = dao.findById(expectedId1);
        Optional<Customer> actualCustomer2 = dao.findById(expectedId2);

        // Then
        assertEquals(expectedName1, actualCustomer1.get().getName());
        assertEquals(expectedName2, actualCustomer2.get().getName());
    }
```

You can even chain returns to get the mock to do something different on each invocation. Note that if you invoke the mock more times than you have stubbed behaviour for it will continue to behave according to the last stub forever.

```
@Test
    public void invoking_mock_with_chained_stubs_returns_different_customers() throws Exception {
        // Given
        long expectedId1 = 10L;
        String expectedName1 = "John Doe";
        String expectedAddress1 = "21 Main Street";
        Customer expectedCustomer1 = new Customer(expectedId1, expectedName1, expectedAddress1);

        long expectedId2 = 20L;
        String expectedName2 = "Jane Deer";
        String expectedAddress2 = "46 High Street";
        Customer expectedCustomer2 = new Customer(expectedId2, expectedName2, expectedAddress2);

        when(mockEntityManager.find(Customer.class, expectedId1))
            .thenReturn(expectedCustomer1).thenReturn(expectedCustomer2);

        // When
        Optional<Customer> actualCustomer1 = dao.findById(expectedId1);
        assertEquals(expectedName2, actualCustomer2.get().getName());
    }
```

Note that we have input the same ID into both calls, the different behaviour is goverened by the second

```
theReturn()
```

method, this only works because the

```
when()
```

part of the stub explicitly expects and input of

```
expectedId1
```

, if we had passed

```
expectedId2
```

we would have gotten a null response from the mock due to the fact that it is not the expected value in the stub.

Now let's test the case where the customer is missing.

```
@Test
    public void finding_missing_customer_should_return_null() throws Exception {
        // Given
        long expectedId = 10L;
        when(mockEntityManager.find(Customer.class, expectedId)).thenReturn(null);

        // When
        Optional<Customer> actualCustomer = dao.findById(expectedId);

        // Then
        assertFalse(actualCustomer.isPresent());
    }
```

Here we can see that we use the same syntax but this time use it to return null.

Mockito allows you to use VarArgs in

```
thenReturn
```

to stub consecutive calls so if we wanted to we could roll the previous two tests into one as follows:

```
@Test
    public void finding_customer_should_respond_appropriately() throws Exception {
        // Given
        long expectedId = 10L;
        String expectedName = "John Doe";
        String expectedAddress = "21 Main Street";
        Customer expectedCustomer1 = new Customer(expectedId, expectedName, expectedAddress);
        Customer expectedCustomer2 = null;

        when(mockEntityManager.find(Customer.class, expectedId)).thenReturn(expectedCustomer1,
expectedCustomer2);

        // When
        Optional<Customer> actualCustomer1 = dao.findById(expectedId);
        Optional<Customer> actualCustomer2 = dao.findById(expectedId);

        // Then
        assertTrue(actualCustomer1.isPresent());
        assertFalse(actualCustomer2.isPresent());
    }
```

What if our find method throws an exception due to some persistence issue? Let's test that!

```
@Test(expected=IllegalArgumentException.class)
    public void finding_customer_should_throw_exception_up_the_stack() throws Exception {
        // Given
        long expectedId = 10L;

        when(mockEntityManager.find(Customer.class, expectedId)).thenThrow(new
IllegalArgumentException());

        // When
        dao.findById(expectedId);

        // Then
        fail("Exception should be thrown.");
    }
```

We have used the

```
thenThrow()
```

method to throw our exception. Contrast this syntax to our use of

```
doThrow()
```

when stubbing void methods. These are two similar but different methods –

will not work with void methods.

4.1. Using Answers

We saw above that we created a customer with certain expected values. If we wanted to create a few known test users and return them base don their Id's we could use an

```
Answer
```

which we could return from our

```
when()
```

calls.

```
Answer
```

is a Generic type provided by Mockito for providing 'canned responses'. It's

```
answer()
```

method takes an

```
InvocationOnMock
```

object which contains certain information about the current mock method call.

Let's create 3 customers and an Answer to choose which one to return based on the input ID.

First the 3 customers are added as private members of the test class.

```
private Customer homerSimpson, bruceWayne, tyrionLannister;
```

Then add a private

```
setupCustomers
```

method to initialize them and call it from the

```
@Before
```

method.

```
@Before
    public void setUp() throws Exception {
        dao = new CustomerDAO(mockEntityManager);
        setupCustomers();
    }

    private void setupCustomers() {
        homerSimpson = new Customer(1, "Homer Simpson", "Springfield");
        bruceWayne = new Customer(2, "Bruce Wayne", "Gotham City");
        tyrionLannister = new Customer(2, "Tyrion Lannister", "Kings Landing");
    }
```

And now we can create an

```
Answer
```

to return an appropriate Customer based on the ID which was passed to the

```
find()
```

method passed to the mock EntityManager at runtime.

```
private Answer<Customer> withCustomerById = new Answer<Customer>() {
        @Override
        public Customer answer(InvocationOnMock invocation) throws Throwable {
            Object[] args = invocation.getArguments();
            int id = ((Long)args[1]).intValue(); // Cast to int for switch.
            switch (id) {
            case 1 : return homerSimpson;
            case 2 : return bruceWayne;
            case 3 : return tyrionLannister;
            default : return null;
            }
        }
    };
```

We can see that we use

```
InvocationOnMock
```

to pull the arguments which were passed into the Mock method invocation. We know that the second argument is the ID so we can read that and determine the appropriate Customer to return. The name of the answer

```
withCustomerById
```

Now let's write a test which demonstrates this answer in action.

```
@Test
    public void finding_customer_by_id_returns_appropriate_customer() throws Exception {
        // Given
        long[] expectedId = {1, 2, 3};

        when(mockEntityManager.find(eq(Customer.class), anyLong())).thenAnswer(withCustomerById);

        // When
        Optional<Customer> actualCustomer0 = dao.findById(expectedId[0]);
        Optional<Customer> actualCustomer1 = dao.findById(expectedId[1]);
        Optional<Customer> actualCustomer2 = dao.findById(expectedId[2]);

        // Then
        assertEquals("Homer Simpson", actualCustomer0.get().getName());
        assertEquals("Bruce Wayne", actualCustomer1.get().getName());
        assertEquals("Tyrion Lannister", actualCustomer2.get().getName());
    }
```

Let's look at the stubbing line in detail.

```
when(mockEntityManager.find(eq(Customer.class), anyLong())).thenAnswer(withCustomerById);
```

Here we see a couple of new things. The first thing is that instead of doing

```
when().thenReturn()
```

we do

```
when().thenAnswer()
```

and provide our

```
withCustomerById
```

Answer as the Answer to be given. The second thing is that we don't use a real value for the ID passed into

```
mockEntityManager.find()
```

instead we use static

```
org.mockito.Matchers.anyLong()
```

. This is a

```
Matcher
```

and it is used to get Mockito to fire the Answer without checking that a particular Long value has been passed in. Matchers let us ignore the parameters to the mock call and instead concentrate only on the return value.

We also decorated

```
Customer.class
```

with the

```
eq()
```

Matcher – this is due to the fact that you can't mix real values and matchers in Mock method calls, you either have to have all parameters as Matchers or all parameters as real values.

```
eq()
```

provides a Matcher which only matches when the runtime parameter is equal to the specified parameter in the stub. This let's us continue to only return the Answer when the input class type is of type Customer.class without specifying a specific ID.

What all this means is that the three invocations of

```
mockEntityManager.find()
```

with different ID's all result in the same Answer being given, and as we have coded the Answer to respond with appropriate Customer objects for different ID's we have successfully mocked an

```
EntityManager
```

capable of mimicking realistic behaviour.

4.2. A note on Behaviour Driven Development test conventions

You may have noticed that we have adopted a convention in our unit tests of splitting the test into 3 parts – // Given, // When and // Then. This convention is called Behaviour Driven Development and is a very logical way of designing unit tests.

- **// Given** is the setup phase where we initialize data and stub mock classes. It is the same as stating 'given the following initial conditions'.

Mockito supports BDD out of the box in the

`org.mockito.BDDMockito`

class. It replaces the normal stubbing methods –

`when()`

,

`thenReturn()`

,

`thenThrow()`

,

`thenAnswer()`

etc with BDD doppelgangers –

`given()`

,

`willReturn()`

,

`willThrow()`

,

`willAnswer()`

. This allows us to avoid using

`when()`

in the // Given section, as it may be confusing.

Because we are using the BDD convention in our tests we will also use the methods provided by BDDMockito.

Lets rewrite

`finding_existing_customer_should_return_customer()`

using BDDMockito syntax.

```java
import static org.mockito.BDDMockito.*;

@Test
    public void finding_existing_customer_should_return_customer_bdd() throws Exception {
        // Given
        long expectedId = 10L;
        String expectedName = "John Doe";
        String expectedAddress = "21 Main Street";
        Customer expectedCustomer = new Customer(expectedId, expectedName, expectedAddress);

        given(mockEntityManager.find(Customer.class, expectedId)).willReturn(expectedCustomer);

        // When
        Optional<Customer> actualCustomer = dao.findById(expectedId);

        // Then
        assertTrue(actualCustomer.isPresent());
        assertEquals(expectedId, actualCustomer.get().getId());
        assertEquals(expectedName, actualCustomer.get().getName());
        assertEquals(expectedAddress, actualCustomer.get().getAddress());
    }
```

The logic of the test has not changed, it is just more readable in BDD form.

4.3. A tip on using Mockito static method in Eclipse

It can be a pain manually adding static imports for the various Mockito static methods if you want to avoid importing

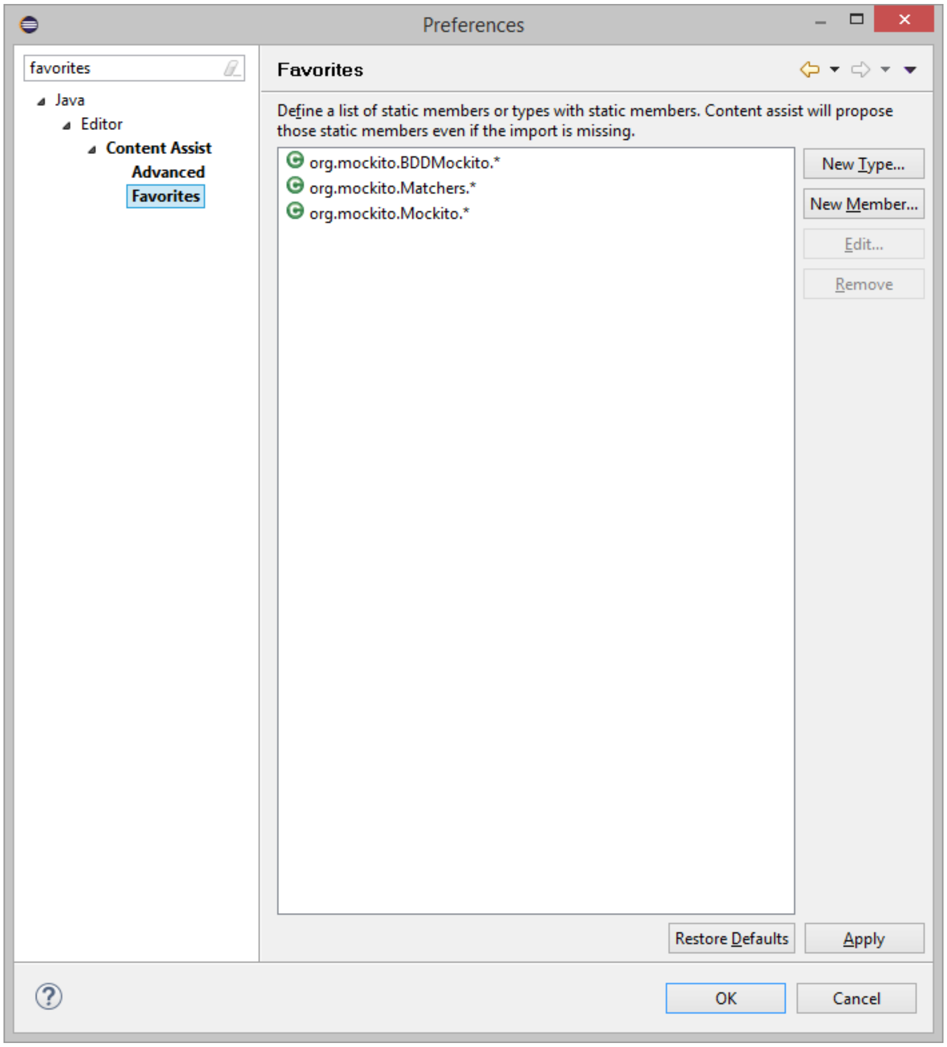
`org.mockito.Mockito.*`

etc. In order to enable content assist in Eclipse for these methods you only need to launch Window -> Preferences and go to Java/Editor/Content Assist/Favorites in the left nav. After that add the following as "New Type..." as per Figure 1.

`org.mockito.Mockito`

```
org.mockito.BDDMockito
```

This will add the Mockito static methods to Eclipse Content Assist allowing you to autocomplete and import them as you use them.



Define a list of static members or types with static members. Content assist will propose those static members even if the import is missing.

- org.mockito.BDDMockito.*
- org.mockito.Matchers.*
- org.mockito.Mockito.*

Figure 1 – Content Assist Favorites

4.4. Using multiple Mocks

We will now look at using multiple mocks in combination together. Lets add a method to our DAO to return a list of all available Customers.

```
public List<Customer> findAll() throws Exception {
        TypedQuery<Customer> query = em.createQuery("select * from CUSTOMER", Customer.class);
        return query.getResultList();
    }
```

Here we see that the

```
createQuery()
```

method of

```
EntityManager
```

returns a generic typed

```
TypedQuery
```

. It takes in as parameters a SQL String and a class which is the return type.

```
TypedQuery
```

itself exposes several methods including

```
List getResultList()
```

query above.

In order to write a test for this method we will want to create a Mock of

```
TypedQuery
```

.

```
@Mock
private TypedQuery<Customer> mockQuery;
```

Now we can stub this mock query to return a list of known customers. Let’s create an answer to do this, and reuse the known Customers which we created previously. You may have noticed that Answer is a Functional Interface, having only one method. We are using Java 8 so we can create a lambda expression to represent our Answer inline, rather than an anonymous inner class like we did in the previous Answer example.

```
given(mockQuery.getResultList()).willAnswer(i -> Arrays.asList(homerSimpson, bruceWayne,
tyrionLannister));
```

Of course we could also code the above stub as

```
given(mockQuery.getResultList()).willReturn(Arrays.asList(homerSimpson, bruceWayne,
tyrionLannister));given
```

which demonstrates the flexibility of Mockito – there are always several different ways of doing the same thing.

Now we have stubbed the behaviour of the mock

```
TypedQuery
```

we can stub the mock

```
EntityManager
```

to return it when requested. Rather than bringing SQL into our test case we will just use the

```
anyString()
```

Matcher to get the mock

```
createQuery()
```

to fire, of course we will also surround the class parameter with an

```
eq()
```

matcher.

The full test looks like this:

```
@Test
    public void finding_all_customers_should_return_all_customers() throws Exception {
        // Given
        given(mockQuery.getResultList()).willAnswer(i -> Arrays.asList(homerSimpson, bruceWayne,
tyrionLannister));
        given(mockEntityManager.createQuery(anyString(), eq(Customer.class))).willReturn(mockQuery);

        // When
        List<Customer> actualCustomers = dao.findAll();

        // Then
        assertEquals(actualCustomers.size(), 3);
    }
```

4.5. Test Yourself! Test Update!

Let’s add the

```
Update()
```

DAO method:

```
public Customer update(Customer customer) throws Exception {
    return em.merge(customer);
}
```

Now see if you can create a test for it. A possible solution has been written in the example code project included with this tutorial. Remember that there are many ways of doing the same thing in Mockito, see if you can think of a few!

5. Argument Matchers

The natural behaviour of Mocktio is to use the method of the object which is passed in as a parameter to see if a particular stubbed behaviour applies. It is possible to avoid using real objects and variables when stubbing however, if it is unimportant to us what those values are. We do this by using Mockito Argument Matchers

We have already seen a couple of the Mockito argument matchers in operation:

```
anyLong()
```

,

```
anyString()
```

and

```
eq
```

. We use these matchers when we don’t particularly care about the input to the Mock, we are only interested in coding it’s return behaviour, and we want it to behave the same way under all conditions.

As already noted, but worth paying special attention to, is that when using argument matchers all arguments must be argument matchers, you can not mix and match real values with argument matchers or you will get a runtime error from Mockito.

Argument Matchers all extend

```
org.mockito.ArgumentMatcher
```

and Mockito includes a library of ready made argument matchers which can be accessed through the static methods of

```
org.mockito.Matchers
```

, to use them just import

```
org.mockito.Matchers.*
```

;

You can look at the javadoc for

```
org.mockito.Matchers
```

to see all the Matchers that Mockito provides, while the following test class demonstrates the usage of some of them:

```
package com.javacodegeeks.hughwphamill.mockito.stubbing;

import static org.junit.Assert.*;
import static org.mockito.Matchers.*;
import static org.mockito.Mockito.*;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class MatchersTest {

    public interface TestForMock {

        public boolean usesPrimitives(int i, float f, double d, byte b, boolean bool);

        public boolean usesObjects(String s, Object o, Integer i);

        public boolean usesCollections(List<String> list, Map<Integer, String> map, Set<Object> set);

        public boolean usesString(String s);

        public boolean usesVarargs(String... s);

        public boolean usesObject(Object o);

    }

    @Mock
    TestForMock test;

    @Test
    public void test() {

        // default behaviour is to return false
        assertFalse(test.usesString("Hello"));

        when(test.usesObjects(any(), any(), any())).thenReturn(true);
        assertTrue(test.usesObjects("Hello", new Thread(), 17));
        Mockito.reset(test);

        when(test.usesPrimitives(anyInt(), anyFloat(), anyDouble(), anyByte(), anyBoolean())).thenReturn(true);
        assertTrue(test.usesPrimitives(1, 43.4f, 3.141592654d, (byte)2, false));
        Mockito.reset(test);

        // Gives unchecked type conversion warning
        when(test.usesCollections(anyList(), anyMap(), anySet())).thenReturn(true);
        assertTrue(test.usesCollections(Arrays.asList("Hello", "World"), Collections.EMPTY_MAP, Collections.EMPTY_SET));
        Mockito.reset(test);

        // Gives no warning
        when(test.usesCollections(anyListOf(String.class), anyMapOf(Integer.class, String.class), anySetOf(Object.class))).thenReturn(true);
        assertTrue(test.usesCollections(Collections.emptyList(), Collections.emptyMap(), Collections.emptySet()));
        Mockito.reset(test);

        // eq() must match exactly
        when(test.usesObjects(eq("Hello World"), any(Object.class),anyInt())).thenReturn(true);
        assertFalse(test.usesObjects("Hi World", new Object(), 360));
        assertTrue(test.usesObjects("Hello World", new Object(), 360));
        Mockito.reset(test);

        when(test.usesString(startsWith("Hello"))).thenReturn(true);
        assertTrue(test.usesString("Hello there"));
        Mockito.reset(test);

        when(test.usesString(endsWith("something"))).thenReturn(true);
        assertTrue(test.usesString("isn't that something"));
        Mockito.reset(test);

        when(test.usesString(contains("second"))).thenReturn(true);
        assertTrue(test.usesString("first, second, third."));
        Mockito.reset(test);

        // Regular Expression
        when(test.usesString(matches("^\\\\w+$"))).thenReturn(true);
        assertTrue(test.usesString("Weak_Password1"));
        assertFalse(test.usesString("@Str0nG!pa$$woR>%42"));
        Mockito.reset(test);

        when(test.usesString((String)isNull())).thenReturn(true);
        assertTrue(test.usesString(null));
        Mockito.reset(test);

        when(test.usesString((String)isNotNull())).thenReturn(true);
        assertTrue(test.usesString("Anything"));
        Mockito.reset(test);

        // Object Reference
        String string1 = new String("hello");
        String string2 = new String("hello");
        when(test.usesString(same(string1))).thenReturn(true);
        assertTrue(test.usesString(string1));
        assertFalse(test.usesString(string2));
        Mockito.reset(test);

        // Compare to eq()
        when(test.usesString(eq(string1))).thenReturn(true);
        assertTrue(test.usesString(string1));
        assertTrue(test.usesString(string2));
        Mockito.reset(test);

        when(test.usesVarargs(anyVararg())).thenReturn(true);
        assertTrue(test.usesVarargs("A","B","C","D","E"));
        assertTrue(test.usesVarargs("ABC", "123"));
        assertTrue(test.usesVarargs("Hello!"));
        Mockito.reset(test);

        when(test.usesObject(isA(String.class))).thenReturn(true);
        assertTrue(test.usesObject("A String Object"));
        assertFalse(test.usesObject(new Integer(7)));
        Mockito.reset(test);

        // Field equality using reflection
        when(test.usesObject(refEq(new SomeBeanWithoutEquals("abc", 123)))).thenReturn(true);
        assertTrue(test.usesObject(new SomeBeanWithoutEquals("abc", 123)));
        Mockito.reset(test);

        // Compare to eq()
        when(test.usesObject(eq(new SomeBeanWithoutEquals("abc", 123)))).thenReturn(true);
        assertFalse(test.usesObject(new SomeBeanWithoutEquals("abc", 123)));
        Mockito.reset(test);

        when(test.usesObject(eq(new SomeBeanWithEquals("abc", 123)))).thenReturn(true);
        assertTrue(test.usesObject(new SomeBeanWithEquals("abc", 123)));
        Mockito.reset(test);
    }

    public class SomeBeanWithoutEquals {
        private String string;
        private int number;

        public SomeBeanWithoutEquals(String string, int number) {
            this.string = string;
            this.number = number;
        }
    }

    public class SomeBeanWithEquals {
            this.string = string;
            this.number = number;
        }

        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + getOuterType().hashCode();
            result = prime * result + number;
            result = prime * result
                    + ((string == null) ? 0 : string.hashCode());
            return result;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            SomeBeanWithEquals other = (SomeBeanWithEquals) obj;
            if (!getOuterType().equals(other.getOuterType()))
                return false;
            if (number != other.number)
                return false;
            if (string == null) {
                if (other.string != null)
                    return false;
            } else if (!string.equals(other.string))
                return false;
            return true;
        }

        private MatchersTest getOuterType() {
            return MatchersTest.this;
        }
    }
}
```

It's also possible to create your own Matchers by extending

```
org.mockito.ArgumentMatcher
```

. Let's create a matcher which fires if a List contains a particular element. We'll also create a static convenience method for creating the Matcher which uses

```
argThat
```

to convert the Matcher into a List for use within the stubbing call. We will implement the

```
matches()
```

method to call the

```
contains
```

method of

```
List
```

to do our actual contains check.

```
public class ListContainsMatcher<T> extends ArgumentMatcher<List<T>> {

    private T element;

    public ListContainsMatcher(T element) {
        this.element = element;
    }

    @Override
    public boolean matches(Object argument) {
        @SuppressWarnings("unchecked")
        List<T> list = (List<T>) argument;
        return list.contains(element);
    }

    public static <T> List<T> contains(T element) {
        return argThat(new ListContainsMatcher<>(element));
    }
}
```

And now a test to demonstrate our new Matcher in action!

```
@RunWith(MockitoJUnitRunner.class)
public class ListContainsMatcherTest {

    public interface TestClass {
        public boolean usesStrings(List<String> list);
        public boolean usesIntegers(List<Integer> list);
    }
    @Mock
    TestClass test;

    @Test
    public void test() throws Exception {
        when(test.usesStrings(contains("Java"))).thenReturn(true);
        when(test.usesIntegers(contains(5))).thenReturn(true);
        assertTrue(test.usesIntegers(integerList));
        assertTrue(test.usesStrings(stringList));
        Mockito.reset(test);

        when(test.usesStrings(contains("Something Else"))).thenReturn(true);
        when(test.usesIntegers(contains(42))).thenReturn(true);
        assertFalse(test.usesStrings(stringList));
        assertFalse(test.usesIntegers(integerList));
        Mockito.reset(test);
    }
}
```

As an exercise try writing your own Matcher which will match if a Map contains a particular key/value pair.

6. Spies and Partial Stubbing

As we saw before it's possible to partially stub a class using the

```
@Spy
```

annotation. Partial stubbing allows us to use a real class in our tests and only stub the specific behaviours that concern us. The Mockito guidelines tell us that spies should be used carefully and occasionally, usually when dealing with legacy code. Best practice is not to use Spy to partially mock the class under test, but instead to partially mock dependencies. The class under test should always be a real object.

Let's imagine that we are dealing with an image manipulation class which works on a

```
java.awt.BufferedImage
```

. This class will take in a

```
BufferedImage
```

into it's constructor and expose a method to fill the image with random coloured vertical stripes and return a thumbnail of the image, based on the input thumbnail height.

```
public class ImageProcessor {

    private BufferedImage image;

    public ImageProcessor(BufferedImage image) {
        this.image = image;
    }

    public Image overwriteImageWithStripesAndReturnThumbnail(int thumbHeight) {
        debugOutputColorSpace();

        Random random = new Random();
        Color color = new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255));

        for (int x = 0; x < image.getWidth(); x++) {
            if (x % 20 == 0) {
                color = new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255));
                for (int y = 0; y < image.getHeight(); y++) {
                    image.setRGB(x, y, color.getRGB());
                }
            }
        }

        Image thumbnail = image.getScaledInstance(-1, thumbHeight, Image.SCALE_FAST);

        Image microScale = image.getScaledInstance(-1, 5, Image.SCALE_DEFAULT);
        debugOutput(microScale);
        return thumbnail;
    }

    private void debugOutput(Image microScale) {
        System.out.println("Runtime type of microScale Image is " + microScale.getClass());

    }

    private void debugOutputColorSpace() {
        for (int i=0; i< image.getColorModel().getColorSpace().getNumComponents(); i++) {
            String componentName = image.getColorModel().getColorSpace().getName(i);
            System.out.println(String.format("Colorspace Component[%d]: %s", i, componentName));
        }
    }
}
```

There's a lot going on in the

```
overwriteImageWithStripesAndReturnThumbnail()
```

method. The first thing it does is output some debug information about the Image's Colorspace. Then it generates some random colours and paints them as horizontal stripes throughout the image, using the images width and height methods. It then does a scale operation to return

We see a lot of interactions with the BufferedImage, most of which are totally internal or random. Ultimately when we want to verify the behaviour of our method the important thing to us is the first call to

```
getScaledInstance()
```

– our class works if the return value of our method is the object which is returned from getScaledInstance(). This is the behaviour of BufferedImage that it is important to us to stub. The problem we face is that there are a lot of other calls to BufferedImages methods. We don't really care about the return values of these methods from the perspective of testing, but if we don't encode behaviour for them somehow they will cause

```
NullPointerException
```

s and possibly other undesirable behaviour.

In order to get around this problem we will create a Spy for the BufferedImage and only stub the

```
getScaledInstance()
```

method which interests us.

Let's create an empty test class with the class under test and Spy created, as well as a Mock for the returned thumbnail.

```
@RunWith(MockitoJUnitRunner.class)
public class ImageProcessorTest {

    private ImageProcessor processor;

    @Spy
    private BufferedImage imageSpy = new BufferedImage(800, 600, BufferedImage.TYPE_INT_ARGB);
    @Mock
    Image mockThumbnail;

    @Before
    public void setup() {
        processor = new ImageProcessor(imageSpy);
    }
}
```

Note that BufferedImage has no default constructor so we've had to instantiate it ourselves using it's parameterized constructor, if it had a default constructor we could have let Mockito instantiate it for us.

Now let's make a first attempt at stubbing the behaviour we are interested in. It makes sense to ignore the input height, width and mode and go ahead and use Argument Matchers for all three. We end up with something like the following:

```
given(imageSpy.getScaledInstance(anyInt(), anyInt(), anyInt())).willReturn(mockThumbnail);
```

Normally this would be the best way to stub for a Spy, however, there's a problem in this case – imageSpy is a real BufferedImage and the stub call passed into

```
given()
```

is a real method call that is actually executed when the stub operation is run by the JVM.

```
getScaledInstance
```

requires that width and height be non zero so this call will result in an

```
IllegalArgumentException
```

being thrown.

One possible solution is to use real arguments in our stub call

```
@Test
    public void scale_should_return_internal_image_scaled() throws Exception {
        // Given
        given(imageSpy.getScaledInstance(-1, 100, Image.SCALE_FAST)).willReturn(mockThumbnail);

        // When
        Image actualImage = processor.overwriteImageWithStripesAndReturnThumbnail(100);

        // Then
        assertEquals(actualImage, mockThumbnail);
    }
```

This test runs successfully and produces the following output on the console

```
Colorspace Component[0]: Red
Colorspace Component[1]: Green
Colorspace Component[2]: Blue
Runtime type of microScale Image is class sun.awt.image.ToolkitImage
```

A side effect of using real values is that the second call to

```
getScaledInstance()
```

But what if we want to continue using Argument Matchers? It's possible to use the

```
doReturn()
```

method (normally used for void methods, if you recall) to stub the

```
getScaledInstance()
```

method without actually calling it at stub time.

```
@Test
public void scale_should_return_internal_image_scaled_doReturn() throws Exception {
    // Given
    doReturn(mockThumbnail).when(imageSpy).getScaledInstance(anyInt(), anyInt(), anyInt());

    // When
    Image actualImage = processor.overwriteImageWithStripesAndReturnThumbnail(100);

    // Then
    assertEquals(actualImage, mockThumbnail);
}
```

This gives the following output:

```
Colorspace Component[0]: Red
Colorspace Component[1]: Green
Colorspace Component[2]: Blue
Runtime type of microScale Image is class $java.awt.Image$$EnhancerByMockitoWithCGLIB$$72355119
```

You can see that the runtime type of the micro image is now the Mock implementation created by Mockito. This is the case because both calls to

```
getScaledInstance
```

match the stub arguments and so the Mock thumbnail is returned from both calls.

There is a way to ensure the real method of the Spy is called in the second instance, this is by using the

```
doCallRealMethod()
```

method of Mockito. As usual Mockito let's you chain together stubbing methods to code different behaviour for consecutive invocations of the stubbed method which match the stub arguments.

```
@Test
    public void scale_should_return_internal_image_scaled_doReturn_doCallRealMethod() throws Exception {
        // Given
        doReturn(mockThumbnail).doCallRealMethod().when(imageSpy).getScaledInstance(anyInt(), anyInt(), anyInt());

        // When
        Image actualImage = processor.overwriteImageWithStripesAndReturnThumbnail(100);

        // Then
        assertEquals(actualImage, mockThumbnail);
    }
```

Which gives the following output

```
Colorspace Component[0]: Red
Colorspace Component[1]: Green
Colorspace Component[2]: Blue
Runtime type of microScale Image is class sun.awt.image.ToolkitImage
```

7. Conclusion

We have looked at a lot of different ways of stubbing behaviour for mocks and spies, and as alluded to there is a near infinite amount of ways one can stub behaviour.

The javadoc for Mockito is a good source of information on the Stubbing methods and particularly on the ArgumentMatchers which Mockito provides out of the box.

We have covered stubbing behaviour in detail and in the next tutorial we will look at verifying the behaviour of Mocks using the Mockito verification framework.

8. Download the Source Code

This was a lesson on Mockito Stubbing. You may download the source code here: mockito2-stubbing

Tagged with: MOCKITO TESTING