Different Types Of Matchers Provided By Mockito

Last Updated:July 16, 2022

An Introduction to Different Types of Matchers in Mockito.

Mocks and Spies in Mockito were explained in detail in our previous tutorial of detailed Mockito training series.

What are Matchers?

Matchers are like regex or wildcards where instead of a specific input (and or output), you specify a range/type of input/output based on which stubs/spies can be rest and calls to stubs can be verified.

All the Mockito matchers are a part of ‘Mockito’ static class.

Matchers are a powerful tool, which enables a shorthand way of setting up stubs as well as verifying invocations on the stubs by mentioning argument inputs as generic types to specific values depending on the use-case or scenario.

Different Types of Matchers provided by Mockito

What You Will Learn: [hide]

Types of Matchers in Mockito

Argument Matchers

Verification Matchers

Tips & Tricks

Conclusion

Recommended Reading

Types of Matchers in Mockito

There are broadly 2 types of matchers in Mockito or in terms of usage, matchers can be used for the below 2 categories:

Argument Matchers during Stub setup

Verification Matchers for verifying actual calls to stubs

For both types of Matchers i.e. Argument and Verification, Mockito provides a huge set of matchers (Click here to get a complete list of the matchers).

Argument Matchers

Listed down are the most widely used ones:

For all the below, let’s consider testing an IntegerList:

final List mockedIntList = mock(ArrayList.class);

#1) any() – Accepts any object (including null).

when(mockedIntList.get(any())).thenReturn(3);

#2) any(java language class) –

Example: any(ClassUnderTest.class) – This is a more specific variant of any() and will accept only objects of the class type that’s mentioned as the template parameter.

when(mockedIntList.get(any(Integer.class))).thenReturn(3);

#3) anyBoolean(), anyByte(), anyInt(), anyString(), anyDouble(), anyFloat(), anyList() and many more – All of these accept any object of the corresponding data type as well as null values.

when(mockedIntList.get(anyInt())).thenReturn(3);

#4) Specific arguments – In cases where actual arguments are known beforehand, its always recommended to use them as they provide more confidence as against generic argument types.

Example:

when(mockedIntList.get(1)).thenReturn(3);

Verification Matchers

There are some specialized matchers that are available to expect/assert things like no. of invocations on the mock.

For all the below matcher’s, let’s consider the same list of example that we have used before.

final List mockedIntList = mock(ArrayList.class);

#1) Mock Invocations

(i) Simple invocation on Mock verifies whether the mocked method was called/interacted or not by setting up the size of the mocked list to 5.

//arrange

when(mockedList.size()).thenReturn(5);

// act

int size = mockedList.size();

// assert

verify(mockedList).size();

(ii) Specific count of interactions with a mocked method verifies the count of no. of times the mock was expected to be called.

//arrange

when(mockedList.size()).thenReturn(5);

// act

int size = mockedList.size();

// assert

verify(mockedList, times(1)).size();

In order to verify for 0 interactions, simply change the value from 1 to 0 as argument for times() matcher.

//arrange

when(mockedList.size()).thenReturn(5);

// act

int size = mockedList.size();

// assert

verify(mockedList, times(0)).size();

In case of failures, it returns the following exceptions:

a) When the expected invocations are less than the actual invocations:

Example: Wanted 2 times, but invoked 3 times, then Mockito returns – “verification.TooManyActualInvocations”

Example code:

final List<Integer> mockedIntList = mock(ArrayList.class);

// Arrange

when(mockedIntList.get(anyInt())).thenReturn(3);

// Act

int response = mockedIntList.get(5);

response = mockedIntList.get(3);

response = mockedIntList.get(100);

// Assert

verify(mockedIntList, times(2)).get(anyInt());

b) When the expected invocations are more than the actual invocations:

Example: Wanted 2 times, but invoked 1 time, then Mockito returns – “verification.TooLittleActualInvocations”

final List<Integer> mockedIntList = mock(ArrayList.class);

// Arrange

when(mockedIntList.get(anyInt())).thenReturn(3);

// Act

int response = mockedIntList.get(5);

response = mockedIntList.get(3);

response = mockedIntList.get(100);

// Assert

verify(mockedIntList, times(4)).get(anyInt());

(iii) No interactions with the specific method of the mocked object.

final List<Integer> mockedIntList = mock(ArrayList.class);

// Arrange

when(mockedIntList.get(anyInt())).thenReturn(3);

// Act

int response = mockedIntList.get(5);

// Assert

verify(mockedIntList, never()).size();

(iv) Verify the order of mocked interactions – This is particularly useful when you want to ensure the order in which the methods on the mocked objects were called.

Example: Database like operations where a test should verify the order in which the database updates happened.

To illustrate this by Example – Let’s continue with the same list of example.

Now let’s assume the order of calls to list methods were in sequence i.e. get(5), size(), get(2). So, the order of verification should be the same as well.

// Arrange

when(mockedIntList.get(anyInt())).thenReturn(3);

when(mockedIntList.size()).thenReturn(100);

InOrder mockInvocationSequence = Mockito.inOrder(mockedIntList);

// Act

int response = mockedIntList.get(5);

int size = mockedIntList.size();

response = mockedIntList.get(2);

// Assert

mockInvocationSequence.verify(mockedIntList, times(1)).get(anyInt());

mockInvocationSequence.verify(mockedIntList).size();

mockInvocationSequence.verify(mockedIntList, times(1)).get(anyInt());

In case of wrong verification sequence, an exception is thrown by Mockito – i.e. “verification.VerificationInOrderFailure”.

So in the above example, if I change the order of verification by interchanging the last 2 lines, I’ll start getting VerificationInOrderFailure exception.

// Arrange

when(mockedIntList.get(anyInt())).thenReturn(3);

when(mockedIntList.size()).thenReturn(100);

InOrder mockInvocationSequence = Mockito.inOrder(mockedIntList);

// Act

int response = mockedIntList.get(5);

int size = mockedIntList.size();

response = mockedIntList.get(2);

// Assert

mockInvocationSequence.verify(mockedIntList, times(1)).get(anyInt());

mockInvocationSequence.verify(mockedIntList, times(1)).get(anyInt());

mockInvocationSequence.verify(mockedIntList).size();

(v) Verify interaction has occurred atleast/atmost number of times.

(a) atleast:

Example: atleast(3) – Verifies that the mocked object was invoked/interacted with atleast thrice during the test. So any of the interactions 3 or greater than 3 should make the verification successful.

// Arrange

when(mockedIntList.get(anyInt())).thenReturn(3);

// Act

int response = mockedIntList.get(5);

response = mockedIntList.get(2);

// Assert

verify(mockedIntList, atLeast(2)).get(anyInt());

In case of errors i.e when the actual invocations do not match, the same exception is thrown as with the times() matcher i.e. “verification.TooLittleActualInvocations”

(b) atmost:

Example: atmost(3) – verifies if the mocked object got invoked/interacted with atmost thrice during the test. So any of 0,1,2 or 3 interactions with the mock should make the verification successful.

// Arrange

when(mockedIntList.get(anyInt())).thenReturn(3);

// Act

int response = mockedIntList.get(5);

response = mockedIntList.get(2);

// Assert

verify(mockedIntList, atMost(2)).get(anyInt());

verify(mockedIntList, atMost(2)).size();

#2) Argument Matching

In the above invocation, matchers can be combined together with the argument matchers to validate the arguments with which the mock was called.

any()

Specific values – Verify with the specific values when the arguments are known beforehand.

Other argument matchers like – anyInt(), anyString() etc.

Tips & Tricks

#1) Using Argument Capture during verification

Argument Capture verification is typically useful where the argument used by some stubbed method is not passed directly via a method call but is created internally when the method under test is called.

This is essentially useful where your method depends on one or more collaborators whose behavior has been stubbed. The arguments passed to these collaborators are an internal object or entirely new argument set.

Validating the actual argument with which the collaborators would have been called ensures a lot of confidence in the code that is being tested.

Mockito provides ArgumentCaptor which can be used with verification and then when “AgumentCaptor.getValue()” is called, we can assert the actual captured argument against the expected one.

To illustrate this, refer to the example below:

In the below method, calculatePrice is the model with the class InventoryModel is created inside the method body which is then used by InventoryService for update.

Now if you want to write a test to validate what argument was the inventoryService called with, you could simply use ArgumentCaptor object of type InventoryModel class.

Method under test:

public double calculatePrice(int itemSkuCode)

{

double price = 0;

// get Item details

ItemSku sku = itemService.getItemDetails(itemSkuCode);

// update item inventory

InventoryModel model = new InventoryModel();

model.setItemSku(sku);

model.setItemSuppliers(new String[]{"Supplier1"});

inventoryService.updateInventory(model, 1);

return sku.getPrice();

}

Test code: Look at the verify step where inventoryService is verified, the argumentCaptor object is substituted for which argument needs to be matched.

Then simply assert the value by invoking getValue() method on ArgumentCaptor object.

Example: ArgumentCaptorObject.getValue()

public void calculatePrice\_withValidItemSku\_returnsSuccess()

{

// Arrange

ItemSku item1 = new ItemSku();

item1.setApplicableDiscount(5.00);

item1.setPrice(100.00);

CustomerProfile customerProfile = new CustomerProfile();

customerProfile.setExtraLoyaltyDiscountPercentage(2.00);

double expectedPrice = 93.00;

// Arrange

when(mockedItemService.getItemDetails(anyInt())).thenReturn(item1);

ArgumentCaptor<InventoryModel> argCaptorInventoryModel = ArgumentCaptor.forClass(InventoryModel.class);

// Act

priceCalculator.calculatePrice(1234);

// Assert

verify(mockedItemService).getItemDetails(anyInt());

verify(mockedInventoryService).updateInventory(argCaptorInventoryModel.capture(), eq(1));

assertEquals(argCaptorInventoryModel.getValue().itemSku, item1);

Without ArgumentCaptor there would be no way to identify what argument was the service call made with. Best possible is to use “any()” or “any(InventoryModel.class)” to verify arguments.

#2) Common Exceptions/Errors while using Matchers

While using Matchers, there are certain conventions that should be followed, which if not followed, results in an exception being thrown. The most common one which I came across is while stubbing and verifying.

If you are using any argumentMatchers and if the stubbed method has more than one argument(s), then either all the arguments should be mentioned with matchers, else none of them should have matchers. Now, what does this mean?

Let’s try to understand this with a scenario (and then code sample for this scenario)

Suppose the method under test has a signature like –

concatenateString(String arg1, String arg2)

Now when stubbing – suppose you know the value of arg1, but arg2 is unknown, so you decide to use an argument matcher like – any() or anyString() and specifying a value for the first argument like some text “hello”.

When the above step is implemented and the test is executed, the test throws an exception called “InvalidUseOfMatchersException”

Let’s try to understand this with an Example:

Test code:

// Arrange

when(a gMatcher.concatenateString("hello", anyString())).thenReturn("hello world!");

// Act

String response = argMatcher.concatenateString("hello", "abc");

// Assert

verify(argMatcher).concatenateString(anyString(), anyString());

Class under test:

public class ArgMatcher {

public String concatenateString(String arg1, String arg2)

{

return arg1.concat(arg2);

}

}

When the above test is executed, it returns in “InvalidUseOfMatchersException”

Now, what’s the reason for this exception?

It’s the stubbing using part matchers and part fixed string i.e. we have mentioned one argument matcher as “hello” and second as anyString(). Now there are 2 ways to get rid of these kinds of exceptions (Also please note – that this behavior applies to both Mock setups as well as behavior).

#1) Use Argument Matchers for all the arguments:

// Arrange

when(a gMatcher.concatenateString(anyString(), anyString())).thenReturn("hello world!");

// Act

String response = argMatcher.concatenateString("hello", "abc");

// Assert

verify(argMatcher).concatenateString(anyString(), anyString());

#2) Use eq() as the Argument Matcher where the argument is known. So instead of specifying the argument as “hello”, specify it as “eq(“hello”) and this should make the stubbing successful.

// Arrange

when(argMatcher.concatenateString(anyString(), eq("world"))).thenReturn("hello world!");

// Act

String response = argMatcher.concatenateString("hello", "world");

// Assert

verify(argMatcher).concatenateString(anyString(), eq("world"));

Conclusion

In this article, we saw how to use different types of matchers provided by Mockito.

Here, we covered the most widely used ones. For referring to the complete list, Mockito Library documentation is a good source of reference.