Extracted from:

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

Ken Kousen

Foreword by Venkat Subramaniam

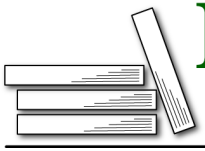
Edited by Margaret Eldridge

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

Ken Kousen

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Margaret Eldridge
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-967-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2023

To Ginger and Xander, the people that matter most to me in the world.

Creating Custom Argument Matchers

To create your own matcher, the ArgumentMatchers class has a method called argThat, which takes an ArgumentMatcher (singular) as an argument. ArgumentMatcher is an interface containing a single abstract method, with this signature:

```
boolean matches(T argument)
```

This method should return true if the argument matches whatever condition you implement, and false otherwise.

In the dark days before lambda expressions were added in Java 8, implementing your own matcher wasn't exactly difficult, but it was verbose. The example given in the documentation is hardly inspiring (comments added):

ListCustomMatcher.java

```
// custom matcher that implements the ArgumentMatcher interface
class ListOfTwoElements implements ArgumentMatcher<List> {
    public boolean matches(List list) {
        return list.size() == 2;
    }

    public String toString() {
        //printed in verification errors
        return "[list of 2 elements]";
    }
}

    // create the mock
    List mock = mock(List.class);

    // set the expectation using argThat and the custom matcher
    when(mock.addAll(argThat(new ListOfTwoElements()))).thenReturn(true);

// somewhere in the actual test, test a method that invokes addAll
// with a two-element list:
    mock.addAll(Arrays.asList("one","two"));

// verify that the test called addAll with the custom matcher
    verify(mock).addAll(argThat(new ListOfTwoElements()));
```

That's a fair amount of work just to see if the mock had one of its methods invoked with a list of size two. Fortunately, with lambda expressions you can write this test easily. Simply provide a lambda expression that takes a single argument of the proper type and returns a Boolean. You can eliminate the class in the example and write the following to set the expectation:

```
verify(mock).addAll(argThat(list -> list.size() == 2));
```

Easy peasy, lemon squeezy. By removing the friction of implementing your own class and instantiating it both when setting the expectations and when verifying them, the whole process becomes much easier.

A working example of using lambdas to implement custom matchers is shown in Testing void Methods Using Interactions, on page ?. Here, though, is an example from the PersonService tests, which leads to the next subtle (but easy to fix) trap. We want to check for IDs that do not exist. Since we know all the IDs in the sample data set are less than 14, we could use a custom matcher like this:

PersonServiceTest.java

```
@Test
@Disabled("Do not use argThat with integers")
public void findByIdsThatDoNotExist_argThat() {
    when(repository.findById(argThat(id -> id > 14)))
        .thenReturn(Optional.empty());

    List<Person> personList = service.findByIds(15, 42, 78, 999);
    assertTrue(personList.isEmpty());

    verify(repository, times(4)).findById(anyInt());
}
```

This leads to a problem, however. The test compiles, but we've fallen into another trap, which involves argThat with primitive types. Fortunately this, too, is easy to fix.

Using Argument Matchers for Primitive Types

Invoking the findById method on PersonRepository returns an empty Optional whenever the requested ID was greater than 14. That was expressed using a custom matcher implemented using a lambda expression:

```
when(repository.findById(argThat(id -> id > 14))
        .thenReturn(Optional.empty());
```

That code looks simple enough, and the lambda expression is correct. Unfortunately, when you run the test, you get a NullPointerException.

Why would the return value from the argThat method be null? Even weirder, if you look at the JavaDocs for the argThat[2] method, even though the signature claims it returns T (the class), the docs say it's supposed to return null, so again, why is this a problem?

The answer lies in this warning for argThat:

2. https://www.javadoc.io/doc/org.mockito/mockito-core/2.7.13/org/mockito/hamcrest/MockitoHamcrest.html

NullPointerException auto-unboxing caveat. In rare cases when matching primitive parameter types you *must* use relevant intThat(), floatThat(), etc. method. This way you will avoid NullPointerException during auto-unboxing. Due to how Java works we don't really have a clean way of detecting this scenario and protecting the user from this problem. Hopefully, the JavaDoc describes the problem and solution well. If you have an idea how to fix the problem, let us know via the mailing list or the issue tracker.

In other words, the problem is related to unboxing wrapper classes into primitive types, and the docs claim this isn't easily fixable. There is, however, an easy fix. When dealing with primitives, instead of calling the argThat method, call one of its primitive variations: byteThat, shortThat, charThat, intThat, longThat, floatThat, doubleThat, and booleanThat.

So the right way to add the custom matcher in this case is simply to use intThat:

PersonServiceTest.java

```
@Test
public void findByIdsThatDoNotExist_intThat() {
    // Custom matcher as lambda argument to intThat:
    when(repository.findById(intThat(id -> id > 14)))
        .thenReturn(Optional.empty());

    List<Person> personList = service.findByIds(15, 42, 78, 999);
    assertTrue(personList.isEmpty());

    verify(repository, times(4)).findById(anyInt());
}
```

Now everything works. That list of methods based on the primitive types is the last category of methods in the ArgumentMatchers class.

Custom Matchers for Primitives

When using a custom argument matcher that is based on primitive types, use the methods designed for that purpose: byteThat, shortThat, intThat, longThat, floatThat, doubleThat, instead of argThat. That approach will avoid potential null pointer exceptions due to unboxing.

We've now covered all the important methods in the ArgumentMatchers class, including how to implement your own custom argument matcher. We can now verify that methods are called with the proper arguments. But to verify the protocol, we also want to check that the methods on the mocked objects are called in the proper order. Fortunately, Mockito makes that easy as well, as you'll see next.

Verifying the Order of Methods Called

One last check we can make is to verify that a particular method on one stub was invoked before another method on a different stub. We'll need an additional class in Mockito to do that, called org.mockito.InOrder, and we'll create an instance of it using another static method from the Mockito class, also called inOrder (but with a lowercase first letter).

Here's the signature of the inOrder method:

```
public static InOrder inOrder(Object... mocks)
```

The inOrder method takes a vararg list of mocks, so to use it you create the mocks first. It's interesting that you can check that different methods were called on different mocks in the order specified. So if the object we're testing had multiple dependencies and we mock them, we can verify that methods are called in the proper order even across different mock objects.

The example in the docs makes this clearer:

```
InOrder inOrder = inOrder(firstMock, secondMock);

inOrder.verify(firstMock).add("was called first");
inOrder.verify(secondMock).add("was called second");
```

This code checks that the add method was called on the first mock with the string argument "was called first", and then the add method was called on the second mock with argument "was called second". You have to explicitly list the calls in order, but they work.

In Chapter 1, Build a Testing Foundation, on page ?, we looked at a class called HelloMockito, which contained two dependencies: one for the PersonRepository we've been using throughout the current chapter and one on a TranslationService that translated the greeting message into whatever language we wanted. We can now check that the appropriate methods in those dependencies are not only invoked the proper number of times (once each), but also in the proper order, as shown in this excerpt from the greetAPersonThatExists test:

HelloMockitoTestFull.java

```
@Test
@DisplayName("Greet Admiral Hopper")
void greetAPersonThatExists() {
    // set the expectations on the mocks
    when(repository.findById(anyInt()))
        .thenReturn(Optional.of(new Person(1, "Grace", "Hopper",
            LocalDate.of(1906, Month.DECEMBER, 9))));
    when(translationService
        .translate("Hello, Grace, from Mockito!", "en", "en"))
```

```
            .thenReturn("Hello, Grace, from Mockito!");

        // test the greet method
        String greeting = helloMockito.greet(1, "en", "en");
        assertEquals("Hello, Grace, from Mockito!", greeting);

        // verify the methods are called once, in the right order
        InOrder inOrder = inOrder(repository, translationService);
        inOrder.verify(repository).findById(anyInt());
        inOrder.verify(translationService)
            .translate(anyString(), eq("en"), eq("en"));
    }
```

In that test, the mocks were created using the `@Mock` annotation and injected into the class under test using the `@InjectMocks` annotation. We set the expectations on the mock, using the `anyInt` argument matcher, tested the `greet` method, and then used the `InOrder` class to not only `verify` the methods were called but confirmed that `findById` was called on the mock repository first, followed by `translate` on the mock translation service. Note also the use of the `eq` matcher for the last two `String` arguments in the `translate` method.

In other words, now you know all the features of Mockito shown in that test, from annotations to setting expectations to verifying method order. Hopefully it all makes sense now.