Mockito – @Mock, @Spy, @Captor and @InjectMocks

By Eugen Paraschiv

1. Overview

In this tutorial, we'll cover all **the annotations in the Mockito library** – *@Mock*, *@Spy*, *@Captor* and *@InjectMocks* .

2. Enable Mockito Annotations

First – let's see how to enable the use of annotations with Mockito tests.

In order for these annotations to be enabled, we'll need to **annotate the JUnit test with a runner** – *MockitoJUnitRunner* as in the following example:

```
@RunWith(MockitoJUnitRunner.class)
public class MockitoAnnotationTest
{
    ...
}
```

Alternatively we can **enable these annotations programmatically** as well, by invoking *MockitoAnnotations.initMocks()* as in the following example:

```
@Before
public void init() {

MockitoAnnotations.initMocks(this);
}
```

3. *@Mock* Annotation

The most used widely used annotation in Mockito is *@Mock*. We can use *@Mock* to create and inject mocked instances without having to call *Mockito.mock* manually.

In the following example – we'll create a mocked *ArrayList* with the manual way without using *@Mock* annotation:

```
@Test
public void whenNotUseMockAnnotation_thenCorrect()
{
    List mockList = Mockito.mock(ArrayList.class);

    mockList.add("one");
    Mockito.verify(mockList).add("one");
    assertEquals(0, mockList.size());

    Mockito.when(mockList.size()).thenReturn(100);
    assertEquals(100, mockList.size());
}
```

And now we'll do the same but we'll inject the mock using the @*Mock* annotation:

```
@Mock
List<String> mockedList;

@Test
public void whenUseMockAnnotation_thenMockIsInjected()
{
    mockedList.add("one");
    Mockito.verify(mockedList).add("one");
    assertEquals(0, mockedList.size());

    Mockito.when(mockedList.size()).thenReturn(100);
    assertEquals(100, mockedList.size());
}
```

Note how – in both examples, we're interacting with the mock and verifying some of these interactions – just to make sure that the mock is behaving correctly.

4. @*Spy* Annotation

Now – let's see how to use @*Spy* annotation to spy on an existing instance.

In the following example – we create a spy of a *List* with the old way without using @*Spy* annotation:

```
@Test
public void whenNotUseSpyAnnotation_thenCorrect() {
    List<String> spyList = Mockito.spy(new ArrayList<String>
());

    spyList.add("one");
    spyList.add("two");

    Mockito.verify(spyList).add("one");
    Mockito.verify(spyList).add("two");

    assertEquals(2, spyList.size());

    Mockito.doReturn(100).when(spyList).size();
    assertEquals(100, spyList.size());
}
```

Let's now do the same – spy on the list – but do so using the *@Spy* annotation:

```
@Spy
List<String> spiedList = new ArrayList<String>();

@Test
public void whenUseSpyAnnotation_thenSpyIsInjected()
{
    spiedList.add("one");
    spiedList.add("two");

    Mockito.verify(spiedList).add("one");
    Mockito.verify(spiedList).add("two");

    assertEquals(2, spiedList.size());

    Mockito.doReturn(100).when(spiedList).size();
    assertEquals(100, spiedList.size());
}
```

Note how, as before – we're interacting with the spy here to make sure that it behaves correctly. In this example we:

- Used the **real** method *spiedList.add()* to add elements to the *spiedList*.
- **Stubbed** the method *spiedList.size()* to return *100* instead of *2* using *Mockito.doReturn()*.

5. *@Captor* Annotation

Next – let's see how to use the *@Captor* annotation to create an *ArgumentCaptor* instance.

In the following example – we create an *ArgumentCaptor* with the old way without using *@Captor* annotation:

```
@Test
public void whenNotUseCaptorAnnotation_thenCorrect() {
    List mockList = Mockito.mock(List.class);
    ArgumentCaptor<String> arg =
ArgumentCaptor.forClass(String.class);

    mockList.add("one");
    Mockito.verify(mockList).add(arg.capture());

    assertEquals("one", arg.getValue());
}
```

Let's now **make use of *@Captor*** for the same purpose – to create an *ArgumentCaptor* instance:

```
@Mock
List mockedList;

@Captor
ArgumentCaptor argCaptor;

@Test
public void whenUseCaptorAnnotation_thenTheSam() {
    mockedList.add("one");

Mockito.verify(mockedList).add(argCaptor.capture());

    assertEquals("one", argCaptor.getValue());
}
```

Notice how the test becomes simpler and more readable when we take out the configuration logic.

6. *@InjectMocks* Annotation

Now – let's discuss how to use *@InjectMocks* annotation – to inject mock fields into the tested object automatically.

In the following example – we use *@InjectMocks* to inject the mock *wordMap* into the *MyDictionary dic*:

```
@Mock
Map<String, String> wordMap;

@InjectMocks
MyDictionary dic = new MyDictionary();

@Test
public void whenUseInjectMocksAnnotation_thenCorrect() {

Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning");

    assertEquals("aMeaning", dic.getMeaning("aWord"));
}
```

And here is the class *MyDictionary*:

```
public class MyDictionary {
    Map<String, String> wordMap;

    public MyDictionary() {
        wordMap = new HashMap<String, String>();
    }
    public void add(final String word, final String meaning)
{
        wordMap.put(word, meaning);
    }
    public String getMeaning(final String word) {
        return wordMap.get(word);
    }
}
```

7. Notes

Finally – here is **some notes** about Mockito annotations:

- Use annotation to minimize repetitive mock creation code
- Use annotation to make the test more readable
- Use *@InjectMocks* to inject both *@Spy* and *@Mock* instances

8. Conclusion

In this quick tutorial, we learned the basics of **annotations in the Mockito library**.

The implementation of all these examples and code snippets **can be found in my Mockito github project** – this is an Eclipse based project, so it should be easy to import and run as it is.

Get the early-bird price (20% Off) of my upcoming "*Learn Spring Security*" Course:

>> CHECK OUT THE COURSE