
mockito

Why do we need mocking ?

- To test your application components in isolation, without a database, DNS server, SVN repository, filesystem.
- Because a class should have one responsibility only and a clean test should emphasize that.
- In a good encapsulated design, a class should behave the same regardless of the implementation classes of its dependencies.

A simple example of mocking

- You can mock concrete classes as well as interfaces. This creates a proxy to the object.

```
LinkedList mockedList = mock(LinkedList.class);

when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());
```

- Calling a method on the object will either involve a call to the real object...

```
mockedList.get(2);
```

- or it may involve a call that returns the mocked vale

```
mockedList.get(0);              // will return "first"
```

Using Mockito for verification testing

- Create a mock using the mock() method or by annotating an object with @Mock

```
List mockedList = mock(List.class);

@Mock List mockedList;
```

- Example of verifying method calls

```
mockedList.add("one");
mockedList.clear();
verify(mockedList).add("one");
verify(mockedList).clear();
```

Stubbing

- By default, for all methods that return a value, mock returns null, an empty collection or appropriate primitive/primitive wrapper value.
- Stubbing can be overridden: for example common stubbing can go to fixture setup, but the test methods can override it. Please note that overridding stubbing is a potential code smell that points out too much stubbing
- Once stubbed, the method will always return the stubbed value regardless of how many times it is called.
- Last stubbing is more important - when you stubbed the same method with the same arguments many times.

Argument Matchers

- Sometimes, when extra flexibility is required then you might use argument matchers

```
when(mockedList.get(anyInt())).thenReturn("a");
when(mockedList.contains(argThat(isValid()))).thenReturn("a");
System.out.println(mockedList.get(999));
```

- You can also verify a call using an argument matcher

```
verify(mockedList).get(anyInt());
```

- If you are using argument matcher then **all arguments** have to be provided by matchers.

```
verify(mock).someMethod(anyInt(), anyString(),
                        eq("third argument"));
```

Verifying exact number of invocations / at least x / never

- The following two verifications work in the same way. We can use times() to specify the number of invocations expected:

```
verify(mockedList).add("once");
verify(mockedList, times(1)).add("once");
```

- Verify that something **never** happened:

```
verify(mockedList, never()).add("never happened");
```

Verify that something happened at least **n** times:

```
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("five times");
verify(mockedList, atMost(5)).add("three times");
```

Verification in order

```
List firstMock = mock(List.class);
List secondMock = mock(List.class);


 //using mocks
firstMock.add("was called first");
secondMock.add("was called second");


//create inOrder object passing any mocks that need to be verified in order
InOrder inOrder = inOrder(firstMock, secondMock);


//following will make sure that firstMock was called before secondMock


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

- Verification in order is flexible - you don't have to verify all interactions one-by-one but only those that you are interested in testing in order.
- Also, you can create InOrder object passing only mocks that are relevant for in-order verification.

Making sure interaction(s) never happened on mock

Verify that method was never called on a mock:

```
verify(mockOne, never()).add("two");
```

Verify that there were no interactions:

```
verifyZeroInteractions(mockTwo, mockThree);
```

Finding redundant invocations

- Using mocks:

```
mockedList.add("one");
mockedList.add("two");

verify(mockedList).add("one");

//following verification will fail
verifyNoMoreInteractions(mockedList);
```

Spy - Partial mocks

```
List list = spy(new LinkedList());
```

Enable partial mock capabilities selectively on mocks:

Foo mock = mock(Foo.class);

Be sure the real implementation is 'safe'.

//If real implementation throws exceptions or depends on specific state of the object then you're in trouble.

when(mock.someMethod()).thenCallRealMethod();

//Impossible: real method is called so spy.get(0) throws IndexOutOfBoundsException (the list is yet empty)
when(spy.get(0)).thenReturn("foo");

Capturing arguments for further assertions

```
ArgumentCaptor<Person> argument =
  ArgumentCaptor.forClass(Person.class);

verify(mock).
   doSomething(argument.capture());

assertEquals("John",
   argument.getValue().getName());
```

doThrow()|doAnswer()|doNothing()|doReturn() family of methods for stubbing voids (mostly)

```
doThrow(new RuntimeException()).
  when(mockedList).clear();
```

Stubbing consecutive calls (iterator-style stubbing)

- Sometimes we need to stub with different return value/exception for the same method call

```
when(mock.someMethod("some arg"))
        .thenThrow(new RuntimeException())
```

- //First call: throws runtime exception: mock.someMethod("some arg");

- //Second call: prints "foo" System.out.println(mock.someMethod("some arg"));
- //Any consecutive call: prints "foo" as well (last stubbing wins). System.out.println(mock.someMethod("some arg"));

- Alternative, shorter version of consecutive stubbing: when(mock.someMethod("some arg")) .thenReturn("one", "two", "three")

Limitations

- Needs java 1.5+
- Cannot mock final classes, static or private methods - but this is possible with PowerMockito.
- Cannot mock equals(), hashCode(). Firstly, you should not mock those methods. Secondly, Mockito defines and depends upon a specific implementation of these methods. Redefining them might break Mockito.
- Spying on real methods where real implementation references outer Class via OuterClass.this is impossible. Don't worry, this is extremely rare case.