Mockito’s Mock Methods

By baeldung

1. Overview

This tutorial illustrates various uses of the standard static *mock* methods of the *Mockito* API.

As with other articles focused on the Mockito framework (like Mockito Verify or Mockito When/Then), the *MyList* class shown below will be used as the collaborator to be mocked in test cases:

```
public class MyList extends AbstractList<String>
{
    @Override
    public String get(int index) {
        return null;
    }

    @Override
    public int size() {
        return 1;
    }
}
```

2. Simple Mocking

The simplest overloaded variant of the *mock* method is the one with a single parameter for the class to be mocked:

```
public
static <T> T mock(Class<T> classToMock)
```

We will use this method to mock a class and set an expectation:

```
MyList listMock = mock(MyList.class);
when(listMock.add(anyString())).thenReturn(false);
```

Then execute a method on the mock:

```
boolean added =
listMock.add(randomAlphabetic(6));
```

The following code confirms that the *add* method has been invoked on the mock, and that the invocation returns a value which matches the expectation we set before:

```
verify(listMock).add(anyString());
assertThat(added, is(false));
```

3. Mocking with Mock's Name

In this section, we will cover another variant of the *mock* method which is provided with an argument specifying the name of the mock:

```
public static <T> T mock(Class<T> classToMock,
String name)
```

Generally speaking, the name of a mock has nothing to do with working code, but may be helpful when it comes to debugging, where the mock's name is used to track down verification errors.

To make sure that the provided name of a mock is included in the message of an exception thrown from an unsuccessful verification, we will rely on a JUnit implementation of the *TestRul*e interface, called *ExpectedException*, and include it in a test class:

```
@Rule
public ExpectedException thrown =
ExpectedException.none();
```

This rule will be used to handle exceptions thrown from test methods.

In the following code, we create a mock for the *MyList* class and name it *myMock*:

```
MyList listMock = mock(MyList.class,
"myMock");
```

Afterwards, set an expectation on a method of the mock and execute it:

```
when(listMock.add(anyString())).thenReturn(false);
listMock.add(randomAlphabetic(6));
```

We will create an intentionally failed verification that should throw an exception with the message containing information about the mock. In order to do it, expectations on the exception need to be set first:

```
thrown.expect(TooLittleActualInvocations.class);
thrown.expectMessage(containsString("myMock.add"));
```

The following verification should fail and throw an exception matching what were expected:

```
verify(listMock,
times(2)).add(anyString());
```

Here is the thrown exception's message:

```
org.mockito.exceptions.verification.TooLittleActualInvocations:
myMock.add(<any>);
Wanted 2 times:
at com.baeldung.mockito.MockitoMockTest
  .whenUsingMockWithName_thenCorrect(MockitoMockTest.java:...)
but was 1 time:
at com.baeldung.mockito.MockitoMockTest
  .whenUsingMockWithName_thenCorrect(MockitoMockTest.java:...)
```

As we can see, the mock's name has been included in the exception message, which will be useful to find the failure point in case of an unsuccessful verification.

4. Mocking with *Answer*

Here, we will demonstrate the use of a *mock* variant in which the strategy for the mock's answers to interaction is configured at creation time. This *mock* method's signature in the Mockito documentation looks like the following:

```
public static <T> T mock(Class<T> classToMock,
Answer defaultAnswer)
```

Let's start with the definition of an implementation of the *Answer* interface:

```
class CustomAnswer implements Answer<Boolean> {
    @Override
    public Boolean answer(InvocationOnMock invocation) throws Throwable
{
        return false;
    }
}
```

The *CustomAnswer* class above is used for the generation of a mock:

```
MyList listMock = mock(MyList.class, new
CustomAnswer());
```

If we do not set an expectation on a method, the default answer, which was configured by the *CustomAnswer* type, will come into play. In order to prove it, we will skip over the expectation setting step and jump to the method execution:

```
boolean added =
listMock.add(randomAlphabetic(6));
```

The following verification and assertion confirm that the *mock* method with an *Answer* argument has worked as expected:

```
verify(listMock).add(anyString());
assertThat(added, is(false));
```

5. Mocking with *MockSettings*

The final *mock* method that is covered within this article is the variant with a parameter of the *MockSettings* type. This overloaded method is use to provide a non-standard mock.

There are several custom settings that are supported by methods of the *MockSettings* interface, such as registering a listener for method invocations on the current mock with *invocationListeners*, configuring serialization with *serializable*, specifying the instance to spy on with *spiedInstance*, configuring Mockito to attempt to use a constructor when instantiating a mock with *useConstructor*, and some others.

For the convenience, we will reuse the *CustomAnswer* class introduced in the previous section to create a *MockSettings* implementation that defines a default answer.

A *MockSettings* object is instantiated by a factory method as follows:

```
MockSettings customSettings = withSettings().defaultAnswer(new
CustomAnswer());
```

That setting object will be used in the creation of a new mock:

```
MyList listMock = mock(MyList.class,
customSettings);
```

Similar to the preceding section, we will invoke the *add* method of a *MyList* instance and verify that a *mock* method with a *MockSettings* argument works as it is meant to by using the following code snippet:

```
boolean added =
listMock.add(randomAlphabetic(6));
verify(listMock).add(anyString());
assertThat(added, is(false));
```

6. Conclusion

This tutorial has covered the *mock* method of Mockito in detail. The implementation of these examples and code snippets can be found in a GitHub project.