Baeldung

Mockito Guide

1. Getting Started with Mockito @Mock, @Spy, @Captor and @InjectMocks

2. Mockito When/Then Examples

3. Mockito's Mock Methods

4. Mockito Verify Method

5. Mockito ArgumentMatchers

6. Mockito – Using Spies

7. Using Mockito ArgumentCaptor

8. Mocking Void Methods With Mockito

9. Mocking Static Methods With Mockito

10. Mock Final Classes and Methods With Mockito

11. Mocking Exception Throwing Using Mockito

12. Mockito and JUnit 5 – Using ExtendWith

1. Getting Started with Mockito @Mock, @Spy, @Captor and @InjectMocks

1. Overview

In this chapter, we'll cover the following **annotations of the Mockito library:** *@Mock*, *@Spy*, *@Captor*, and *@InjectMocks.*

Before we get started, let's explore different ways to enable the use of annotations with Mockito tests.

2.1. MockitoJUnitRunner

The first option we have is to **annotate the JUnit test with a *MockitoJUnitRunner***:

```
1. @ExtendWith(MockitoExtension.class)
2. public class MockitoAnnotationUnitTest {
3.     ...
4. }
```

2.2. MockitoAnnotations.openMocks()

Alternatively, we can **enable Mockito annotations programmatically** by invoking *MockitoAnnotations.openMocks()*:

```
1. @BeforeEach
2. public void init() {
3.     MockitoAnnotations.openMocks(this);
4. }
```

2.3. MockitoJUnit.rule()

Finally, we can use a *MockitoJUnit.rule()*:

```
1. public class MockitoAnnotationsInitWithMockitoJUnitRuleUnitTest {
2.     @Rule
3.     public MockitoRule initRule = MockitoJUnit.rule();
4.     ...
5. }
```

In this case, we must remember to make our rule *public*.

3. @Mock Annotation

The most 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 manually without using the **@Mock annotation**:

```
1.  @Test
2.  public void whenNotUseMockAnnotation_thenCorrect() {
3.      List mockList = Mockito.mock(ArrayList.class);
4.
5.      mockList.add("one");
6.      Mockito.verify(mockList).add("one");
7.      assertEquals(0, mockList.size());
8.
9.      Mockito.when(mockList.size()).thenReturn(100);
10.     assertEquals(100, mockList.size());
11. }
```

Now we'll do the same, but we'll inject the mock using the *@Mock annotation*:

```
1.  @Mock
2.  List<String> mockedList;
3.
4.  @Test
5.  public void whenUseMockAnnotation_thenMockIsInjected() {
6.      mockedList.add("one");
7.      Mockito.verify(mockedList).add("one");
8.      assertEquals(0, mockedList.size());
9.
10.     Mockito.when(mockedList.size()).thenReturn(100);
11.     assertEquals(100, mockedList.size());
12. }
```

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 the *@Spy annotation* to spy on an existing instance.

```
1.  @Test
2.  public void whenNotUseSpyAnnotation_thenCorrect() {
3.      List<String> spyList = Mockito.spy(new ArrayList<String>());
4.
5.      spyList.add("one");
6.      spyList.add("two");
7.
8.      Mockito.verify(spyList).add("one");
9.      Mockito.verify(spyList).add("two");
10.
11.     assertEquals(2, spyList.size());
12.
13.     Mockito.doReturn(100).when(spyList).size();
14.     assertEquals(100, spyList.size());
15. }
```

Now we'll do the same thing, spy on the list, but we'll use the *@Spy* annotation:

```
1.  @Spy
2.  List<String> spiedList = new ArrayList<String>();
3.
4.  @Test
5.  public void whenUseSpyAnnotation_thenSpyIsInjectedCorrectly() {
6.      spiedList.add("one");
7.      spiedList.add("two");
8.
9.      Mockito.verify(spiedList).add("one");
10.     Mockito.verify(spiedList).add("two");
11.
12.     assertEquals(2, spiedList.size());
13.
14.     Mockito.doReturn(100).when(spiedList).size();
15.     assertEquals(100, spiedList.size());
16. }
```

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()*

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

In the following example, we'll create an *ArgumentCaptor* without using the *@Captor* annotation:

```
1.  @Test
2.  public void whenNotUseCaptorAnnotation_thenCorrect() {
3.      List mockList = Mockito.mock(List.class);
4.      ArgumentCaptor<String> arg = ArgumentCaptor.forClass(String.
5.  class);
6.
7.      mockList.add("one");
8.      Mockito.verify(mockList).add(arg.capture());
9.
10.
11.     assertEquals("one", arg.getValue());
12. }
```

Now let's make use of *@Captor* for the same purpose, to create an *ArgumentCaptor* instance:

```
1.  @Mock
2.  List mockedList;
3.
4.  @Captor
5.  ArgumentCaptor argCaptor;
6.
7.  @Test
8.  public void whenUseCaptorAnnotation_thenTheSame() {
9.      mockedList.add("one");
10.     Mockito.verify(mockedList).add(argCaptor.capture());
11.
12.     assertEquals("one", argCaptor.getValue());
13. }
```

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 the *@InjectMocks* annotation to inject mock fields into the tested object automatically.

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

```
1.  @Mock
2.  Map<String, String> wordMap;
3.
4.  @InjectMocks
5.  MyDictionary dic = new MyDictionary();
6.
7.  @Test
8.  public void whenUseInjectMocksAnnotation_thenCorrect() {
9.      Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning");
10.
11.     assertEquals("aMeaning", dic.getMeaning("aWord"));
12. }
```

Here's the class MyDictionary:

```
1.  public class MyDictionary {
2.      Map<String, String> wordMap;
3.
4.      public MyDictionary() {
5.          wordMap = new HashMap<String, String>();
6.      }
7.      public void add(final String word, final String meaning) {
8.          wordMap.put(word, meaning);
9.      }
10.     public String getMeaning(final String word) {
11.         return wordMap.get(word);
12.     }
13. }
```

7. Injecting a Mock Into a Spy

Similar to the above test, we might want to inject a mock into a spy:

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

@Spy
MyDictionary spyDic = new MyDictionary();
```

However, Mockito doesn't support injecting mocks into spies, and the following test results in an exception:

```
@Test
public void whenUseInjectMocksAnnotation_thenCorrect() {
    Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning");

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

If we want to use a mock with a spy, we can manually inject the mock through a constructor:

```
MyDictionary(Map<String, String> wordMap) {
    this.wordMap = wordMap;
}
```

Instead of using the annotation, we can now create the spy manually:

```
@Mock
Map<String, String> wordMap;
MyDictionary spyDic;
@BeforeEach
public void init() {
    MockitoAnnotations.openMocks(this);
    spyDic = Mockito.spy(new MyDictionary(wordMap));
}
```

The test will now pass.

Often, we may run into a ***NullPointerException*** when we try to actually use the instance annotated with *@Mock* or *@Spy*:

```
public class MockitoAnnotationsUninitializedUnitTest {

    @Mock
    List<String> mockedList;

    @Test(expected = NullPointerException.class)
    public void whenMockitoAnnotationsUninitialized_
thenNPEThrown() {
        Mockito.when(mockedList.size()).thenReturn(1);
    }
}
```

Most of the time, this happens simply because we forget to properly enable Mockito annotations.

So we have to keep in mind that each time we want to use Mockito annotations, we must take the extra step and initialize them, as we already explained earlier.

By calling *publish()*, we're given a *ConnectableFlux*. This means that calling *subscribe()* won't cause it to start emitting, allowing us to add multiple subscriptions:

```
publish.subscribe(System.out::println);
publish.subscribe(System.out::println);
```

If we try running this code, nothing will happen. It's not until we call connect() that the *Flux* will start emitting:

```
publish.connect();
```

9. Notes

Finally, here are some notes about Mockito annotations:

- Mockito's annotations minimize repetitive mock creation code.
- They make tests more readable.
- *@InjectMocks* is necessary for injecting both *@Spy* and *@Mock* instances.

10. Conclusion

In this brief chapter, we explained the basics of **annotations in the Mockito library.**

The implementation of all of these examples can be found on GitHub. This is a Maven project, so it should be easy to import and run as it is.

2. Mockito When/Then Examples

In this chapter, we'll learn **how to use Mockito to configure behavior** in a variety of examples and use cases.

The **format of this section is example focused** and practical, no extraneous details or explanations are necessary.

We're going to be **mocking a simple list** implementation:

```
1.  public class MyList extends AbstractList<String> {
2.
3.      @Override
4.      public String get(final int index) {
5.          return null;
6.      }
7.      @Override
8.      public int size() {
9.          return 1;
10.     }
11. }
```

Configure simple return behavior for mock:

```
1. MyList listMock = mock(MyList.class);
2. when(listMock.add(anyString())).thenReturn(false);
3.
4. boolean added = listMock.add(randomAlphabetic(6));
5. assertThat(added).isFalse();
```

Configure return behavior for mock in an alternative way:

```
1. MyList listMock = mock(MyList.class);
2. doReturn(false).when(listMock).add(anyString());
3.
4. boolean added = listMock.add(randomAlphabetic(6));
5. assertThat(added).isFalse();
```

Configure mock to throw an exception on a method call:

```
1. MyList listMock = mock(MyList.class);
2. when(listMock.add(anyString())).thenThrow(IllegalStateException.
3. class);
4.
5. assertThrows(IllegalStateException.class, () -> listMock.
6. add(randomAlphabetic(6)));
```

Configure the behavior of a method with void return type to throw an exception:

```
1. MyList listMock = mock(MyList.class);
2. doThrow(NullPointerException.class).when(listMock).clear();
3.
4. assertThrows(NullPointerException.class, () -> listMock.clear());
```

Configure the behavior of multiple calls:

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

assertThrows(IllegalStateException.class, () -> {
    listMock.add(randomAlphabetic(6));
    listMock.add(randomAlphabetic(6));
});
```

Configure the behavior of a spy:

```
MyList instance = new MyList();
MyList spy = spy(instance);

doThrow(NullPointerException.class).when(spy).size();

assertThrows(NullPointerException.class, () -> spy.size());
```

Configure method to call the real, underlying method on a mock:

```
MyList listMock = mock(MyList.class);
when(listMock.size()).thenCallRealMethod();

assertThat(listMock).hasSize(1);
```

Configure mock method call with custom Answer:

```
MyList listMock = mock(MyList.class);
doAnswer(invocation -> “Always the same”).when(listMock).
get(anyInt());

String element = listMock.get(1);
assertThat(element).isEqualTo(“Always the same”);
```

3. Conclusion

The goal of this chapter is simply to have this information readily available.

The implementation of all of these examples and code snippets can be found on GitHub.

3. Mockito's Mock Methods

In this chapter, we'll illustrate the various uses of the standard static mock methods of the *Mockito* API.

As in other chapters 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:

```
1.  public class MyList extends AbstractList<String> {
2.
3.      @Override
4.      public String get(int index) {
5.          return null;
6.      }
7.
8.      @Override
9.      public int size() {
10.         return 1;
11.     }
12. }
```

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

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

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

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

Then we'll execute a method on the mock:

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

The following code confirms that we invoked the *add* method on the mock. The invocation returns a value that matches the expectation we set before:

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

3. Mocking With Mock's Name

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

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

Generally speaking, the name of a mock has nothing to do with the working code. However, it may be helpful in debugging, as we use the mock's name to track down verification errors.

To ensure the exception message thrown from an unsuccessful verification includes the provided name of a mock, we'll use *assertThatThrownBy*.
In the following code, we'll create a mock for the *MyList* class and name it *myMock*:

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

Then we'll set an expectation on a method of the mock, and execute it:

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

Next, we'll call the verification inside the *assertThatThrownBy*, and verify the instance of the exception thrown:

```
1. assertThatThrownBy(() -> verify(listMock, times(2)).
2. add(anyString()))
3.     .isInstanceOf(TooFewActualInvocations.class)
```

Further, we can also verify the exception's message that it should contain the information about the mock:

```
assertThatThrownBy(() -> verify(listMock, times(2)).
add(anyString()))
    .isInstanceOf(TooFewActualInvocations.class)
    .hasMessageContaining("myMock.add");
```

Here's 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 exception message includes the mock's name, which will be useful for finding the failure point in case of an unsuccessful verification.

Here we'll demonstrate the use of a *mock* variant in which we'll configure the strategy for the mock's answers to interaction 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;
    }
}
```

We'll use the *CustomAnswer* class above for the generation of a mock:

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

If we don't set an expectation on a method, the default answer, configured by the *CustomAnswer* type, will come into play. In order to prove this, we'll 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 worked as expected:

```
verify(listMock).add(anyString());
assertThat(added).isFalse();
```

The final mock method we'll cover in this chapter is the variant with a parameter of the *MockSettings* type. We use this overloaded method to provide a non-standard mock.

There are several custom settings 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*, etc.

For convenience, we'll 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:

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

We'll use that setting object in the creation of a new mock:

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

Similar to the preceding section, we'll invoke the *add* method of a *MyList* instance, and verify that the *mock* method with a *MockSettings* argument works as expected:

```
1. boolean added = listMock.add(randomAlphabetic(6));
2.
3. verify(listMock).add(anyString());
4. assertThat(added).isFalse();
```

6. Conclusion

In this chapter, we covered the *mock* method of Mockito in detail. The implementation of these examples and code snippets can be found in the GitHub project.

4. Mockito Verify Method

In this chapter, we'll illustrate **how to use Mockito verify** in a practical manner throughout a variety of use cases.

We're going to be **mocking a simple list** implementation:

```
1.  public class MyList extends AbstractList<String> {
2.
3.      @Override
4.      public String get(final int index) {
5.          return null;
6.      }
7.
8.      @Override
9.      public int size() {
10.         return 1;
11.     }
12. }
```

Verify simple invocation on mock:

```
1. List<String> mockedList = mock(MyList.class);
2. mockedList.size();
3.
4. verify(mockedList).size();
```

Verify the number of interactions with mock:

```
1. List<String> mockedList = mock(MyList.class);
2. mockedList.size();
3. verify(mockedList, times(1)).size();
```

Verify no interaction with the whole mock occurred:

```
1. List<String> mockedList = mock(MyList.class);
2. verifyNoInteractions(mockedList);
```

Verify no interaction with a specific method occurred:

```
1. List<String> mockedList = mock(MyList.class);
2. verify(mockedList, times(0)).size();
```

Verify there are no unexpected interactions (this should fail):

```
1. List<String> mockedList = mock(MyList.class);
2. mockedList.size();
3. mockedList.clear();
4.
5. verify(mockedList).size();
6. assertThrows(NoInteractionsWanted.class, () ->
7. verifyNoMoreInteractions(mockedList));
```

Verify the order of interactions:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
mockedList.add("a parameter");
mockedList.clear();

InOrder inOrder = Mockito.inOrder(mockedList);
inOrder.verify(mockedList).size();
inOrder.verify(mockedList).add("a parameter");
inOrder.verify(mockedList).clear();
```

Verify an interaction hasn't occurred:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();

verify(mockedList, never()).clear();
```

Verify an interaction has occurred at least a certain number of times:

```
List<String> mockedList = mock(MyList.class);
mockedList.clear();
mockedList.clear();
mockedList.clear();

verify(mockedList, atLeast(1)).clear();
verify(mockedList, atMost(10)).clear();
```

Verify interaction with the exact argument:

```
List<String> mockedList = mock(MyList.class);
mockedList.add("test");

verify(mockedList).add("test");
```

Verify interaction with flexible/any argument:

```
List<String> mockedList = mock(MyList.class);
mockedList.add("test");

verify(mockedList).add(anyString());
```

Verify interaction using argument capture:

```
List<String> mockedList = mock(MyList.class);
mockedList.addAll(Lists.<String> newArrayList("someElement"));

ArgumentCaptor<List<String>> argumentCaptor = ArgumentCaptor.
forClass(List.class);
verify(mockedList).addAll(argumentCaptor.capture());

List<String> capturedArgument = argumentCaptor.getValue();
assertThat(capturedArgument).contains("someElement");
```

The implementation of all of these examples and code snippets can be found on GitHub.

5. Mockito ArgumentMatchers

In this chapter, we'll learn **how to use the *ArgumentMatcher*, and discuss how it differs from the *ArgumentCaptor***.

We need to add a single artifact:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
```

The latest version of *Mockito* can be found on *Maven Central*.

We can configure a mocked method in various ways. One option is to return a fixed value:

```
1. doReturn("Flower").when(flowerService).analyze("poppy");
```

In the above example, the *String* "Flower" is returned only when the *analyze* method of *FlowerService* receives the String "poppy."

But there may be a case where **we need to respond to a wider range of values or unknown values.**

In these scenarios, **we can configure our mocked methods with *argument matchers*:**

```
1. when(flowerService.analyze(anyString())).thenReturn("Flower");
```

Now, because of the *anyString* argument matcher, the result will be the same no matter what value we pass to analyze. *ArgumentMatchers* allow us flexible verification or stubbing.

If a method has more than one argument, we can't just use *ArgumentMatchers* for only some of the arguments. *Mockito* requires that we provide all arguments either by *matchers* or exact values.

Here we can see an example of an incorrect approach:

```
1. when(flowerService.isABigFlower("poppy", anyInt())).
2. thenReturn(true);
```

We can verify this by running the below test:

```
assertThrows(InvalidUseOfMatchersException.class,
    () -> when(flowerService.isABigFlower("poppy", anyInt())).
thenReturn(true));
```

To fix this and keep the String name "poppy" as desired, we'll use eq *matcher*:

```
when(flowerService.isABigFlower(eq("poppy"), anyInt())).
thenReturn(true);
```

Let's run the test to confirm this:

```
when(flowerService.isABigFlower(eq("poppy"), anyInt())).
thenReturn(true);

Flower flower = new Flower("poppy", 15);

Boolean response = flowerController.isABigFlower(flower);
assertThat(response).isTrue();
```

There are two more points to note when we use matchers:

- **We can't use them as a return value**; we require an exact value when stubbing calls.
- **We can't use argument *matchers* outside of verification or stubbing.**

As per the second point, Mockito will detect the misplaced argument and throw an InvalidUseOfMatchersException.

A bad example of this would be:

```
flowerController.isAFlower("poppy");

String orMatcher = or(eq("poppy"), endsWith("y"));
assertThrows(InvalidUseOfMatchersException.class, () ->
verify(flowerService).analyze(orMatcher));
```

The way we'd implement the above code is:

```
1. verify(flowerService).analyze(or(eq("poppy"), endsWith("y")));
```

Mockito also provides *AdditionalMatchers* to implement common logical operations ('not', 'and', 'or') on *ArgumentMatchers* that match both primitive and non-primitive types.

Creating our own matcher allows us to select the best possible approach for a given scenario, and produce high-quality tests that are clean and maintainable.

For instance, we can have a *MessageController* that delivers messages. It'll receive a *MessageDTO*, and from that, it'll create a Message that *MessageService* will deliver.

Our verification will be simple; we'll verify that we called the *MessageService* exactly 1 time with any *Message*:

```
1. MessageDTO messageDTO = new MessageDTO();
2. messageDTO.setFrom("me");
3. messageDTO.setTo("you");
4. messageDTO.setText("Hello, you!");
5. messageController.createMessage(messageDTO);
6. verify(messageService, times(1)).deliverMessage(any(Message.class));
```

Since the **Message is constructed inside the method under test,** we must use any as the matcher.

This approach doesn't let us validate the data inside the Message, which can be different from the data inside the *MessageDTO*.

For this reason, we'll implement a custom argument matcher:

```
1. public class MessageMatcher implements ArgumentMatcher<Message> {
2.
3.     private Message left;
4.     // constructors
5.     @Override
6.     public boolean matches(Message right) {
7.         return left.getFrom().equals(right.getFrom()) &&
8.           left.getTo().equals(right.getTo()) &&
9.           left.getText().equals(right.getText()) &&
10.          right.getDate() != null &&
11.          right.getId() != null;
12.    }
13. }
```

To use our matcher, we need to modify our test and replace *any* by *argThat*:

```
1.  MessageDTO messageDTO = new MessageDTO();
2.  messageDTO.setFrom("me");
3.  messageDTO.setTo("you");
4.  messageDTO.setText("Hello, you!");
5.
6.  messageController.createMessage(messageDTO);
7.
8.  Message message = new Message();
9.  message.setFrom("me");
10. message.setTo("you");
11. message.setText("Hello, you!");
12.
13. verify(messageService, times(1)).deliverMessage(argThat(new
14. MessageMatcher(message)));
```

Now we know our Message instance will have the same data as our *MessageDTO*.

Both techniques, *custom argument matchers* and *ArgumentCaptor*, can be used to make sure certain arguments are passed to mocks.

However, ***ArgumentCaptor* may be a better fit if we need it to assert on argument values** to complete the verification, **or if our *custom argument matcher* isn't likely to be reused**.

Custom argument matchers via *ArgumentMatcher* are usually better for stubbing.

In this chapter, we explored *ArgumentMatcher*, a feature of *Mockito*. We also discussed how it differs from *ArgumentCaptor*.

As always, the full source code of the examples is available on GitHub.

6. Mockito – Using Spies

In this chapter, we'll illustrate how to make the most out of **spies in Mockito**.

We'll talk about the *@Spy* annotation, and how to stub a spy. Finally, we'll go into the difference between *Mock* and *Spy*.

Let's start with a simple example of **how to use a *spy***.

Simply put, the API is *Mockito.spy()* to **spy on a real object**.

This will allow us to call all the normal methods of the object, while still tracking every interaction, just as we would with a mock.

Now let's do a quick example where we'll spy on an existing *ArrayList object*:

```
1.  @Test
2.  void givenUsingSpyMethod_whenSpyingOnList_thenCorrect() {
3.      List<String> list = new ArrayList<String>();
4.      List<String> spyList = spy(list);
5.
6.      spyList.add("one");
7.      spyList.add("two");
8.
9.      verify(spyList).add("one");
10.     verify(spyList).add("two");
11.
12.     assertThat(spyList).hasSize(2);
13. }
```

Note how **the real method *add()* is actually called**, and how the size of *spyList* becomes 2.

Next, let's see how to use the *@Spy* annotation. We can use the *@Spy* annotation instead of *spy()*:

```
1.  @Spy
2.  List<String> spyList = new ArrayList<String>();
3.
4.  @Test
5.  void givenUsingSpyAnnotation_whenSpyingOnList_thenCorrect() {
6.      spyList.add("one");
7.      spyList.add("two");
8.
9.      verify(spyList).add("one");
10.     verify(spyList).add("two");
11.
12.     assertThat(aSpyList).hasSize(2);
13. }
```

To **enable Mockito annotations** (such as *@Spy*, *@Mock*, ...), we need to use *@ExtendWith(MockitoExtension.class)*, which initializes mocks and handles strict stubbings.

Now let's see how to stub a *Spy*. We can configure/override the behavior of a method using the same syntax we would use with a mock.

Here we'll use *doReturn()* to override the *size()* method:

```
1.  @Test
2.  void givenASpy_whenStubbingTheBehaviour_thenCorrect() {
3.      List<String> list = new ArrayList<String>();
4.      List<String> spyList = spy(list);
5.
6.      assertEquals(0, spyList.size());
7.
8.      doReturn(100).when(spyList).size();
9.      assertThat(spyList).hasSize(100);
10. }
```

Let's discuss the difference between *Mock* and *Spy* in Mockito. We won't examine the theoretical differences between the two concepts, just how they differ within Mockito itself.

When Mockito creates a mock, it does so from the *Class* of a Type, not from an actual instance. The mock simply creates a **bare-bones shell instance** of the Class, entirely instrumented to track interactions with it.

On the other hand, **the spy will wrap an existing instance**. It'll still behave in the same way as the normal instance; the only difference is that it'll also be instrumented to track all the interactions with it.

Here we'll create a *mock* of the *ArrayList* class:

```
1. @Test
2. void whenCreateMock_thenCreated() {
3.     List mockedList = mock(ArrayList.class);
4.     mockedList.add("one");
5.     verify(mockedList).add("one");
6.     assertThat(mockedList).hasSize(0);
7. }
```

As we can see, adding an element into the mocked list doesn't actually add anything; it just calls the method with no other side effects.

A spy, on the other hand, will behave differently; it'll actually call the real implementation of the *add* method, and add the element to the underlying list:

```
1. @Test
2. void whenCreateSpy_thenCreate() {
3.     List spyList = Mockito.spy(new ArrayList());
4.
5.     spyList.add("one");
6.     Mockito.verify(spyList).add("one");
7.
8.     assertThat(spyList).hasSize(1);
9. }
```

In this final section, we'll learn about the Mockito *NotAMockException*. **This exception is one of the common exceptions we'll likely encounter when misusing mocks or spies.**

Let's start by understanding the circumstances in which this exception can occur:

```
1. List<String> list = new ArrayList<String>();
2. doReturn(100).when(list).size();
```

When we run this code snippet, we'll get the following error:

```
1. org.mockito.exceptions.misusing.NotAMockException:
2. Argument passed to when() is not a mock!
3. Example of correct stubbing:
4.     doThrow(new RuntimeException()).when(mock).someMethod();
```

Thankfully, it's quite clear from the Mockito error message what the problem is here. In our example, the *list* object isn't a mock. **The Mockito *when()* method expects a mock or spy object as the argument.**

As we can also see, the Exception message even describes what a correct invocation should look like. Now that we have a better understanding of what the problem is, let's fix it by following the recommendation:

```
1. final List<String> spyList = spy(new ArrayList<>());
2. assertThatNoException().isThrownBy(() -> doReturn(100).
3. when(spyList).size());
```

Our example now behaves as expected, and we no longer see the Mockito *NotAMockException*.

In this brief chapter, we discussed the most useful examples of using Mockito spies.

We learned how to create a spy, use the *@Spy annotation*, stub a *spy*, and finally, the difference between *Mock* and *Spy*.

The implementation of all of these examples **can be found** on GitHub.

7. Using Mockito ArgumentCaptor

In this chapter, we'll cover a common scenario of using Mockito *ArgumentCaptor* in our unit tests.

ArgumentCaptor allows us to capture an argument passed to a method to inspect it. This **is especially useful when we can't access the argument outside of the method we'd like to test.**

For example, consider an *EmailService* class with a send method that we'd like to test:

```
1.  public class EmailService {
2.
3.
4.      private DeliveryPlatform platform;
5.
6.
7.      public EmailService(DeliveryPlatform platform) {
8.          this.platform = platform;
9.      }
10.
11.
12.     public void send(String to, String subject, String body,
13. boolean html) {
14.         Format format = Format.TEXT_ONLY;
15.         if (html) {
16.             format = Format.HTML;
17.         }
18.         Email email = new Email(to, subject, body);
19.         email.setFormat(format);
20.         platform.deliver(email);
21.     }
22.
23.
24.     ...
25. }
```

In *EmailService.send*, notice how *platform.deliver* takes a new *Email* as an argument. As part of our test, we'd like to check that the format field of the new *Email* is set to *Format.HTML*. To do this, we need to capture and inspect the argument that is passed to platform.deliver.

Let's see how we can use *ArgumentCaptor* to help us.

2.1. Set Up the Unit Test

First, let's create our unit test class:

```
@ExtendWith(MockitoExtension.class)
class EmailServiceUnitTest {


    @Mock
    DeliveryPlatform platform;


    @InjectMocks
    EmailService emailService;

    ...
}
```

We're using the *@Mock annotation* to mock *DeliveryPlatform*, which is automatically injected into our *EmailService* with the *@InjectMocks* annotation. Refer to our Mockito annotations chapter for further details.

2.2. Add an ArgumentCaptor Field

Second, let's add a new *ArgumentCaptor* field of type *Email* to store our captured argument:

```
@Captor
ArgumentCaptor<Email> emailCaptor;
```

2.3. Capture the Argument

Third, let's use *verify()* with the *ArgumentCaptor* to capture the *Email*:

```
verify(platform).deliver(emailCaptor.capture());
```

We can then get the captured value, and store it as a new *Email* object:

```
Email emailCaptorValue = emailCaptor.getValue();
```

2.4. Inspect the Captured Value

Finally, let's see the whole test with an assert to inspect the captured *Email* object:

```
@Test
void whenDoesSupportHtml_expectHTMLEmailFormat() {
    String to = "info@baeldung.com";
    String subject = "Using ArgumentCaptor";
    String body = "Hey, let'use ArgumentCaptor";


    emailService.send(to, subject, body, true);


    verify(platform).deliver(emailCaptor.capture());
    Email value = emailCaptor.getValue();
    assertThat(value.getFormat()).isEqualTo(Format.HTML);
}
```

Although **we can use an *ArgumentCaptor* with stubbing, we should generally avoid doing so**. To clarify, in Mockito, this generally means avoiding using an *ArgumentCaptor* with *Mockito.when*. With stubbing, we should use an *ArgumentMatcher* instead.

Let's look at a couple of reasons why we should avoid stubbing.

3.1. Decreased Test Readability

First, consider a simple test:

```
1.  Credentials credentials = new Credentials("baeldung", "correct_
2.  password", "correct_key");
3.  when(platform.authenticate(eq(credentials))).
4.  thenReturn(AuthenticationStatus.AUTHENTICATED);
5.
6.  assertTrue(emailService.authenticatedSuccessfully(credentials));
```

Here we used *eq(credentials)* to specify when the mock should return an object.
Now consider the same test using an *ArgumentCaptor* instead:

```
1.  Credentials credentials = new Credentials("baeldung", "correct_
2.  password", "correct_key");
3.  when(platform.authenticate(credentialsCaptor.capture())).
4.  thenReturn(AuthenticationStatus.AUTHENTICATED);
5.
6.  assertTrue(emailService.authenticatedSuccessfully(credentials));
7.  assertEquals(credentials, credentialsCaptor.getValue());
```

In contrast to the first test, notice how we have to perform an extra assert on the last line to do the same as *eq(credentials).*

Finally, notice how it isn't immediately clear what *credentialsCaptor.capture()* refers to. **This is because we have to create the captor outside the line we use it on, reducing readability**.

3.2. Reduced Defect Localization

Another reason is that if *emailService.authenticatedSuccessfully* doesn't call *platform.authenticate*, we'll get an exception:

```
org.mockito.exceptions.base.MockitoException:
No argument value was captured!
```

This is because our stubbed method hasn't captured an argument. However, the real issue isn't in our test itself, but in the actual method we're testing.

In other words, **it misdirects us to an exception in the test, whereas the actual defect is in the method we're testing**.

In this short chapter, we looked at a general use case of using *ArgumentCaptor*. We also looked at the reasons to avoid using *ArgumentCaptor* with stubbing.

As usual, all of our code samples are available on GitHub.

8. Mocking Void Methods With Mockito

In this short chapter, we'll focus on mocking *void* methods with Mockito. As with other chapters focused on the Mockito framework (such as Mockito Verify, Mockito When/Then, and Mockito's Mock Methods), the MyList class shown below will be used as the collaborator in test cases.

We'll add a new method for this chapter:

```
1. public class MyList extends AbstractList<String> {
2.
3.     @Override
4.     public void add(int index, String element) {
5.         // no-op
6.     }
7. }
```

Void methods can be used with Mockito's *doNothing()*, *doThrow()*, and *doAnswer()* methods, making mocking and verifying intuitive:

```
@Test
public void whenAddCalled_thenVerified() {
    MyList myList = mock(MyList.class);
      doNothing().when(myList).add(isA(Integer.class), isA(String.
class));
    myList.add(0, "");

    verify(myList, times(1)).add(0, "");
}
```

However, *doNothing()* is Mockito's default behavior for *void* methods.

This version of *whenAddCalledVerified()* accomplishes the same thing as the one above:

```
@Test
void whenAddCalled_thenVerified() {
    MyList myList = mock(MyList.class);
    myList.add(0, "");

    verify(myList, times(1)).add(0, "");
}
```

doThrow() generates an exception:

```
@Test
void givenNull_whenAddCalled_thenThrowsException() {
    MyList myList = mock(MyList.class);
    assertThrows(Exception.class, () -> {
     doThrow().when(myList).add(isA(Integer.class), isNull());
    });
    myList.add(0, null);
}
```

We'll cover doAnswer() below.

One reason to override the default behavior with *doNothing()* is to capture arguments.

In the example above, we used the *verify()* method to check the arguments passed to *add()*.

However, we may need to capture the arguments and do something more with them.

In these cases, we use *doNothing()* just as we did above, but with an *ArgumentCaptor*:

```
1.  @Test
2.  void givenArgumentCaptor_whenAddCalled_thenValueCaptured() {
3.      MyList myList = mock(MyList.class);
4.
5.          ArgumentCaptor<String>  valueCapture  =  ArgumentCaptor.
6.  forClass(String.class);
7.      doNothing().when(myList).add(any(Integer.class), valueCapture.
8.  capture());
9.
10.     myList.add(0, “captured”);
11.
12.     assertEquals(“captured”, valueCapture.getValue());
13. }
```

A method may perform more complex behavior than merely adding or setting value.

For these situations, we can use Mockito's *Answer* to add the behavior we need:

```
1.  @Test
2.  void givenDoAnswer_whenAddCalled_thenAnswered() {
3.      MyList myList = mock(MyList.class);
4.
5.      doAnswer(invocation -> {
6.          Object arg0 = invocation.getArgument(0);
7.          Object arg1 = invocation.getArgument(1);
8.
9.          assertEquals(3, arg0);
10.         assertEquals("answer me", arg1);
11.         return null;
12.     }).when(myList).add(any(Integer.class), any(String.class));
13.
14.     myList.add(3, "answer me");
15. }
```

As explained in Mockito's Java 8 Features, we use a lambda with *Answer* to define custom behavior for *add()*.

Partial mocks are an option too. Mockito's *doCallRealMethod()* can be used for *void* methods:

```
@Test
void givenDoCallRealMethod_whenAddCalled_thenRealMethodCalled() {
    MyList myList = mock(MyList.class);

    doCallRealMethod().when(myList).add(any(Integer.class),
any(String.class));
    myList.add(1, “real”);

    verify(myList, times(1)).add(1, “real”);
}
```

This way, we can call the actual method and verify it at the same time.

In this brief chapter, we covered four different ways to approach *void* methods when testing with Mockito.

As always, the examples are available in this GitHub project.

In this chapter, we learned how to build different types of URIs using WebClient and *DefaultUriBuilder*.

Along the way, we covered various types and formats of query parameters. Finally, we wrapped up by changing the default encoding mode of the URL builder.

As always, all of the code snippets from the chapter are available over on GitHub repository.

9. Mocking Static Methods With Mockito

When writing tests, we'll often encounter situations where we need to mock a static method. **Before version 3.4.0 of Mockito, it wasn't possible to mock static methods directly, only with the help of PowerMockito**.

In this chapter, we'll take a look at how we can now mock static methods using the latest version of Mockito.

The focus of our tests will be a simple static utility class:

```
1.  public class StaticUtils {
2.
3.
4.      private StaticUtils() {}
5.
6.
7.      public static List<Integer> range(int start, int end) {
8.          return IntStream.range(start, end)
9.            .boxed()
10.           .collect(Collectors.toList());
11.     }
12.
13.
14.     public static String name() {
15.         return “Baeldung”;
16.     }
17. }
```

For demonstration purposes, we have one method with some arguments and another one that simply returns a String.

Generally speaking, some might say that when writing clean object-orientated code, we shouldn't need to mock static classes. **This could typically hint at a design issue or code smell in our application.**

Why? First, a class depending on a static method has tight coupling, and second, it nearly always leads to code that's difficult to test. Ideally, a class shouldn't be responsible for obtaining its dependencies, and if possible, they should be externally injected.

So it's always worth investigating if we can refactor our code to make it more testable. Of course, this isn't always possible, and sometimes we need to mock static methods.

Let's go ahead and see how we can mock the name method from our StaticUtils class:

```
@Test
void givenStaticMethodWithNoArgs_whenMocked_
thenReturnsMockSuccessfully() {
    assertThat(StaticUtils.name()).isEqualTo("Baeldung");


    try (MockedStatic<StaticUtils> utilities = Mockito.
mockStatic(StaticUtils.class)) {
        utilities.when(StaticUtils::name).thenReturn("Eugen");
        assertThat(StaticUtils.name()).isEqualTo("Eugen");
    }


    assertThat(StaticUtils.name()).isEqualTo("Baeldung");
}
```

As previously mentioned, since Mockito 3.4.0, we can use the Mockito. *mockStatic(Class<T> classToMock)* method to mock invocations to static method calls. **This method returns a MockedStatic object for our type, which is a scoped mock object.**

Therefore, in our unit test above, the utilities variable represents a mock with a thread-local explicit scope. **It's important to note that scoped mocks must be closed by the entity that activates the mock**. This is why we define our mock within a try-with-resources construct, so that the mock is closed automatically when we finish with our scoped block.

This is a particularly nice feature, since it assures that our static mock remains temporary. As we know, if we're playing around with static method calls during our test runs, this will likely lead to adverse effects on our test results due to the concurrent and sequential nature of running tests.

On top of this, another nice side effect is that our tests will still run quite fast, since Mockito doesn't need to replace the classloader for every test. In our example, we reiterate this point by checking before and after our scoped block that our static method name returns a real value.

Now let's see another common use case when we need to mock a method that has arguments:

```
@Test
void givenStaticMethodWithArgs_whenMocked_
thenReturnsMockSuccessfully() {
    assertThat(StaticUtils.range(2, 6)).containsExactly(2, 3, 4,
5);


    try (MockedStatic<StaticUtils> utilities = Mockito.
mockStatic(StaticUtils.class)) {
        utilities.when(() -> StaticUtils.range(2, 6))
          .thenReturn(Arrays.asList(10, 11, 12));


        assertThat(StaticUtils.range(2, 6)).containsExactly(10,
11, 12);
    }


    assertThat(StaticUtils.range(2, 6)).containsExactly(2, 3, 4,
5);
}
```

Here **we follow the same approach, but this time we use a lambda expression inside our when clause where we specify the method along with any arguments that we want to mock**. Pretty straightforward!

In this brief chapter, we explored a couple examples of how we can use Mockito to mock static methods. To sum up, Mockito provides a graceful solution using a narrower scope for mocked static objects via one small lambda.
As always, the full source code of the chapter is available on GitHub.

10. Mock Final Classes and Methods With Mockito

In this short chapter, we'll focus on how to mock final classes and methods using Mockito.

As with other chapters focused on the Mockito framework (such as Mockito Verify, Mockito When/Then and Mockito's Mock Methods), we'll use the *MyList* class shown below as the collaborator in test cases.

We'll add a new method for this tutorial:

```
1. public class MyList extends AbstractList<String> {
2.     final public int finalMethod() {
3.         return 0;
4.     }
5. }
```

And we'll also extend it with a *final* subclass:

```
1. public final class FinalList extends MyList {
2.
3.
4.     @Override
5.     public int size() {
6.         return 1;
7.     }
8. }
```

Once we've properly configured Mockito, we can mock a final method like any other:

```
1.  @Test
2.  public void whenMockFinalMethod_thenMockWorks() {
3.
4.
5.      MyList myList = new MyList();
6.
7.
8.      MyList mock = mock(MyList.class);
9.      when(mock.finalMethod()).thenReturn(1);
10
11.
12.     assertThat(mock.finalMethod()).isNotZero();
13. }
```

By creating a concrete instance and a mock instance of *MyList*, we can compare the values returned by both versions of *finalMethod()*, and verify that the mock is called.

Mocking a final class is just as easy as mocking any other class:

```
1.  @Test
2.  public void whenMockFinalClass_thenMockWorks() {
3.
4.
5.      FinalList mock = mock(FinalList.class);
6.      when(mock.size()).thenReturn(2);
7.
8.
9.      assertThat(mock.size()).isNotEqualTo(1);
10. }
```

Similar to the test above, we'll create a concrete instance and a mock instance of our final class, mock a method, and verify that the mocked instance behaves differently.

In this quick chapter, we covered how to mock final classes and methods with Mockito by using a Mockito extension.

The full examples, as always, can be found on GitHub.

11. Mocking Exception Throwing Using Mockito

In this quick chapter, we'll focus on how to configure a method call to throw an exception with Mockito.

Here's the simple dictionary class that we'll use:

```
1.  class MyDictionary {
2.
3.      private Map<String, String> wordMap;
4.
5.
6.      public void add(String word, String meaning) {
7.          wordMap.put(word, meaning);
8.      }
9.
10.
11.     public String getMeaning(String word) {
12.         return wordMap.get(word);
13.     }
14. }
```

First, if our method return type isn't *void*, we can use *when().thenThrow():*

```
@Test
void givenNonVoidReturnType_whenUsingWhenThen_
thenExceptionIsThrown() {
    MyDictionary dictMock = mock(MyDictionary.class);
    when(dictMock.getMeaning(anyString())).
thenThrow(NullPointerException.class);

    assertThrows(NullPointerException.class, () -> dictMock.
getMeaning("word"));
}
```

Notice that we configured the *getMeaning()* method, which returns a value of type *String*, to throw a *NullPointerException* when called.

Now if our method returns *void*, we'll use *doThrow()*:

```
1.  @Test
2.  void givenVoidReturnType_whenUsingDoThrow_thenExceptionIsThrown()
3.  {
4.      MyDictionary dictMock = mock(MyDictionary.class);
5.      doThrow(IllegalStateException.class).when(dictMock)
6.          .add(anyString(), anyString());
7.
8.      assertThrows(IllegalStateException.class, () -> dictMock.
9.  add("word", "meaning"));
10. }
```

Here we configured an *add()* method, which returns *void*, to throw *IllegalStateException* when called.

We can't use *when().thenThrow()* with the *void* return type, as the compiler doesn't allow *void* methods inside brackets.

To configure the exception itself, we can pass the exception's class, as in our previous examples, or as an object:

```
@Test
void givenNonVoidReturnType_
whenUsingWhenThenAndExeceptionAsNewObject_thenExceptionIsThrown()
{
    MyDictionary dictMock = mock(MyDictionary.class);
    when(dictMock.getMeaning(anyString())).thenThrow(new
NullPointerException("Error occurred"));

    assertThrows(NullPointerException.class, () -> dictMock.
getMeaning("word"));
}
```

And we can do the same with *doThrow()*:

```
@Test
void givenNonVoidReturnType_
whenUsingDoThrowAndExeceptionAsNewObject_thenExceptionIsThrown() {
    MyDictionary dictMock = mock(MyDictionary.class);
    doThrow(new IllegalStateException("Error occurred")).
when(dictMock)
        .add(anyString(), anyString());

    assertThrows(IllegalStateException.class, () -> dictMock.
add("word", "meaning"));
}
```

We can also configure *Spy* to throw an exception the same way we did with the mock:

```
@Test
void givenSpyAndNonVoidReturnType_whenUsingWhenThen_
thenExceptionIsThrown() {
    MyDictionary dict = new MyDictionary();
    MyDictionary spy = Mockito.spy(dict);
    when(spy.getMeaning(anyString())).
thenThrow(NullPointerException.class);

    assertThrows(NullPointerException.class, () -> spy.
getMeaning("word"));
}
```

In this chapter, we explored how to configure method calls to throw an exception in Mockito.

As always, the full source code can be found on GitHub.

12. Mockito and JUnit 5 – Using ExtendWith

In this quick chapter, we'll demonstrate **how to integrate Mockito with the JUnit 5 extension model**. To learn more about the JUnit 5 extension model, have a look at this article.

First, we'll show how to create an extension that automatically creates mock objects for any class attribute or method parameter annotated with *@Mock*.

Then we'll use our Mockito extension in a JUnit 5 test class.

Let's add the JUnit 5 (jupiter) and mockito dependencies to our pom.xml:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
```

The latest versions of *junit-jupiter-engine* and *mockito-core* can be downloaded from Maven Central.

Mockito provides an implementation for JUnit5 extensions in the library: *mockito-junit-jupiter*.

We'll include this dependency in our *pom.xml*:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
```

Let's build our test class, and attach the Mockito extension to it:

```
@ExtendWith(MockitoExtension.class)
class UserServiceUnitTest {


    UserService userService;


    // ...
}
```

We can use the *@Mock* annotation to inject a mock for an instance variable that we can use anywhere in the test class:

```
@Mock UserRepository userRepository;
```

We can also inject mock objects into method parameters:

```
@BeforeEach
void init(@Mock SettingRepository settingRepository) {
    userService = new DefaultUserService(userRepository,
settingRepository, mailClient);

    lenient().when(settingRepository.getUserMinAge()).
thenReturn(10);

    when(settingRepository.getUserNameMinLength()).thenReturn(4);

    lenient().when(userRepository.
isUsernameAlreadyExists(any(String.class)))
            .thenReturn(false);
}
```

Please note the use of *lenient()* here. *Mockito* throws an *UnsupportedStubbingException* when an initialized mock isn't called by one of the test methods during execution. We can avoid this strict stub checking by using this method when initializing the mocks.

We can even inject a mock object into a test method parameter:

```
@Test
void givenValidUser_whenSaveUser_thenSucceed(@Mock MailClient
mailClient) {
    // Given
    user = new User("Jerry", 12);
    when(userRepository.insert(any(User.class))).then(new
Answer<User>() {
        int sequence = 1;

        @Override
        public User answer(InvocationOnMock invocation) throws
Throwable {
            User user = (User) invocation.getArgument(0);
            user.setId(sequence++);
            return user;
        }
    });


    userService = new DefaultUserService(userRepository,
settingRepository, mailClient);


    // When
    User insertedUser = userService.register(user);

    // Then
    verify(userRepository).insert(user);
    assertNotNull(user.getId());
    verify(mailClient).sendUserRegistrationMail(insertedUser);
}
```

Note that the *MailClient* mock we inject as a test parameter will NOT be the same instance we injected in the *init* method.

Junit 5 has provided a nice model for the extension. We demonstrated a simple Mockito extension that simplified our mock creation logic.

All of the code used in this chapter can be found on our GitHub project.