CODEGYM

Java Developer in 12 months

Mentor-supported training program from CodeGym

MODULE 3. JAVA PROFESSIONAL

Lesson #4 Mockito

CODEGYM

Lesson plan

- Mockito Library
- Mocking objects
- Invoking mock methods with parameters
- Identifying specific behaviors of objects

CODEGYM

Mockito Library

This library is a standard for testing Spring, which is actually an industry standard in Java backend development.

To add the Mockito library to your **pom.xml**, you can use the following code:

```
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-core</artifactId>
   <version>4.2.0</version>
   <scope>test</scope>
</dependency>
```

CODEGYM

Mock objects

During the development and evolution of testing, there has often been a need to substitute a real object with a "mock" (a virtual object) in the code. Such virtual objects are called **mocks**, derived from the word "mock" meaning a model or imitation.

The Mockito library has elevated the usage of these mocks to new heights. Hence, the name of the library itself is derived from this concept.

The @Mock annotation

There are two ways to work with mock objects in Mockito.
The first way is to create a fully virtual object:

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

The second way is to wrap an existing object in a certain wrapper:

```
@RunWith(MockitoJUnitRunner.class)
class Test {
   @Mock
   List mockList;

   @Test
   public void whenNotUseMockAnnotation_thenCorrect() {
   }
}
```

CODEGYM

The @Spy annotation

The second important type of objects in Mockito is wrappers around existing objects. They allow you to both utilize existing classes and intercept method invocations and variable accesses of such objects, enabling you to modify their behavior where needed.

To create a wrapper around an object, you need to write the code:

```
ClassName variableName = Mockito.spy(ClassName.class);
```

In the simplest form, the wrapper object simply redirects the method calls to the original object, which it holds a reference to internally. Everything will work as with the original object.

You can also create a wrapper using the @Spy annotation.

```
@RunWith(MockitoJUnitRunner.class)
class Test {
   @Spy
   List mockList = new ArrayList<String>();
}
```

CODEGYM

Mocking objects

The doReturn() method

The Mockito library allows you to define the desired behavior for a mock object.

If you want a mock object to return a specific result when a certain method is called, you can add this "rule" to the object using the following code:

```
Mockito.doReturn(result).when(object).methodName();
```

The when() method

There is another way to add behavior rules to a mock object using the Mockito.when() method. It looks like this:

```
Mockito.when(object.methodName()).thenReturn(result);
```

This is an alternative way of defining the behavior rule for a mock object. Let's compare it to the previous method:

```
Mockito.doReturn(result).when(object).methodName();
```

The first example indeed has a couple of limitations:

- the invocation `object.methodName()` can be misleading and confusing
- It will not work if the method `methodName()` returns `void`

The doThrow() method

To make a method throw an exception, you can define a rule using the doThrow() method.

```
Mockito.doThrow(Exception.class).when(object).methodName();
```

And the second variant:

```
Mockito.when(object.methodName).thenThrow(Exception.class);
```

If you need to throw a specific exception object, you can use the following construct:

```
Mockito.doThrow(new Exception()).when(result).methodName();
```

Method invocations on mock objects with parameters

If you want a specific return value for a method based on a certain parameter, you can define the rule like this:

```
Mockito.doReturn(result).when(object).methodName(parameter);
```

Parameter templates

If you want to add a rule to a mock object that applies to a method with any arguments, there is a special object for that:

```
Mockito.any()
```

The **Mockito.any()** object has the type **Object**, so there are equivalents for parameters of different types:

Method	Parameter type
any()	Object, including null
any(ClassName.class)	ClassName
anyInt()	int
anyBoolean()	boolean
anyDouble()	double
anyList()	List

CODEGYM

The doAnswer() method

If you need a virtual method to have complex behavior,

such as returning values based on parameters or converting a string to uppercase,

there is a special method called **doAnswer().** It takes a function as an argument, which allows you to define the desired behavior.

```
Mockito.doAnswer(function).when(object).methodName(parameter);
```

Detecting specific behavior in objects

The verify() method

If you need to ensure that the tested class has called the correct methods on the expected objects, with the correct number of times and parameters, Mockito provides a family of methods called **Mockito.verify(...)**. The general rule for verifying method invocation looks like this:

```
Mockito.verify(object).methodName(parameter);
```

Note that here we are simply setting a rule of action that should occur in the future. The actual verification will be performed after the test method has finished executing:

- **First, we set the rule of how a method should be invoked.**
- **Then, the method is called (or not called).**
- **After the test method has finished, Mockito verifies whether the rules were met.**

The verify() method with checking the number of invocations

The rule for checking the number of method invocations can be specified as follows:

```
Mockito.verify(object, quantity).methodName(parameter);
```

Important! The quantity is not of type **int**, but a special object that can specify different patterns. There are specific methods available to define different scenarios:

never()	verifies that the method was never called
times(n)	n times
atLeast(n)	n or more times
atLeastOnce()	1 or more than 1 time
atMost(n)	n or less times
only()	There should be only one invocation and only to this method

You can also verify that apart from the specified method invocations, **no other interactions with the object occurred.**

Method invocation order

To define a strict order of method invocations, you can use the special object called **InOrder**. First, you need to create an InOrder object:

```
InOrder inOrder = Mockito.inOrder(object);
```

And then you can add the rules to the InOrder object by calling the **verify()** methods on it.

Checking Exceptions in Mockito

The fact that exceptions were thrown can be verified using the same approach as method invocations. However, in this case, you need to use the **thenThrow()** method. The general format of such a rule is as follows:

```
Mockito.verify(object.methodName()).thenThrow(Exception.class);
```

CODEGYM

Mocking a static method mockStatic()

1. Create a special mock object of the class:

```
MockedStatic<ClassName>controlObject= Mockito.mockStatic(ClassName.class);
```

2. Add rules of behavior to this object:
Rules should be attached to this object using other methods.

```
controlObject.when(ClassName::methodName).thenReturn(result);
```

3. It is important to **wrap the usage of this object in a try-with-resources block** so that the object is immediately cleaned up and Mockito can clear any associated rules.

Homework

MODULE 3. JAVA PROFESSIONAL

Level 4 Mockito

MODULE 3. JAVA PROFESSIONAL

Answers to questions

MODULE 3. JAVA PROFESSIONAL