COURSEWARE

- Professional Skills
- Agile Fundamentals
- Jira
- Git
- Databases Introduction
- Java Beginner
- Maven
- Testing (Foundation)
  - Introduction to Testing
  - Types of Testing
  - Functional Testing
  - V-model
  - Unit Testing
  - What is Mockito?
  - Mocking Objects With Mockito
- Java Intermediate
- HTML
- CSS
- Javascript
- Spring Boot
- Selenium
- Sonarqube
- Advanced Testing (Theory)
- Cucumber
- MongoDB
- Express
- NodeJS
- React
- Express-Testing
- Networking
- Security

Mocking Objects With Mockito

Contents

- Overview
- Tutorial
  - Adding the Dependency
  - Creating a Mock Object
  - Using The Created Mocked Objects
  - Behaviour Testing
- Exercises

Overview

Mockito allows us to create mock objects within our test classes to simulate and define their behaviour.
The most common use of this is to mock a database connection, which allows us to ensure that the test conditions are the same for each of our test cases. Therefore mocking allows us to test the expected interaction between classes and validate that the right methods have been called.

Tutorial

Adding the Dependency

To begin using Mockito in our tests we first need to import the dependency in Maven, this can be done with the following dependency, change the dependency where necessary.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.2.4</version>
    <scope>test</scope>
</dependency>
```

Creating a Mock Object

To create a mocked object we use the `@Mock` annotation above the variable of the class we want to mock, this must be a class level variable so that it can be used across all tests.
To then use these mocks we need to use the `@InjectMocks` tag above the variable of the class we are testing, this variable must also be a class level variable.
This then allows us to use the mocks within our tests.

The `@RunWith(MockitoJUnitRunner.class)` tag tells Mockito to populate the annotated fields with dummy data, meaning it tells Mockito to mock any objects that have the `@Mock` tag.
All of these tags can be seen in the below example.

Cloud Fundamentals
AWS Foundations
AWS Intermediate
Linux
DevOps
Jenkins Introduction
Jenkins Pipeline
Markdown
IDE Cheatsheet

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class MockitoTest {

    @Mock
    private DatabaseConnector databaseConnector;

    @InjectMocks
    private QueryClass queryClass;

    @Test
    public void testCreate() {
        QueryClass queryClass = new QueryClass(databaseConnector);
    }
}
```

Using The Created Mocked Objects

Now that we have created our mock objects, we need to define their functionality when certain methods within the class are called.
If we do not define the functionality then Mockito will simply return default values, which are generally not what we want to be returned as it will return things like `null` for objects.
We can use the `when()` and `thenReturn()` methods to define the functionality of our mocks and tell Mockito what we want to be returned when a certain method is called.

```
@Test
public void test() {
    Customer mockedCustomer = mock(Customer.class);
    when(mockedCustomer.getUniqueId()).thenReturn(23);
    assertEquals(mockedCustomer.getUniqueId(), 23);
}
```

In the above example we are creating a mock object from the `Customer` class and giving it the reference variable `mockedCustomer`.
Then we are defining the functionality of the mock object by telling Mockito that `when` the `getUniqueId()` method is called`thenReturn` the value 23, instead of actually calling the method.
We are then having the test call the `getUniqueId()` method as our actual value within an `assertEquals()` call, and giving it the value 23 as our expected.
This assert will return true, therefore the test will pass.

We can also specify what we want returned before specifying when we want it returned if we want.
So instead of using `when()` then using `thenReturn()`, we can use `doReturn()` then use `when()`.
If we use this method only the mocked object goes within the parameters of the `when()` method call and the method we are calling goes after the parentheses, this can be seen in the below example.

```
@Test
public void test() {
    Customer mockedCustomer = mock(Customer.class);
    doReturn(23).when(mockedCustomer).getUniqueId();
    assertEquals(mockedCustomer.getUniqueId(), 23);
}
```

The above example provides the same result as the previous example, only we are specifying what we want returned before when we want it returned.

Behaviour Testing

Mockito allows us to test the behaviour of our project, meaning that it allows us to make sure that the right methods are being called the right number of times for what we expect it to do.
This can be done with the `verify()` method, as Mockito keeps track of all the method calls and their parameters made against a mocked object.
If we just want to make sure that a method has been called and are not concerned with how many times it has been called we can do the following.

```
@Test
public void test() {
    Customer mockedCustomer = mock(Customer.class);
    when(mockedCustomer.getUniqueId()).thenReturn(23);
    int id = mockedCustomer.getUniqueId();
    verify(mockedCustomer).getUniqueId();
}
```

The above example is creating our mocked object and telling Mockito that when we call `getUniqueId()` against the mocked object we want it to return 23.
Then we are calling the `getUniqueId()` agaisnt our mocked object and passing the value retuned into an `int` variable called `id`.
Finally we use the `verify()` method with our mocked object as the parameter and tell Mockito that we want to make sure that the `getUniqueId()` method was called against our mocked object.

We test the behaviour of our code further by using a second parameter within the `verify()` method which will be a method call to tell Mockito how many times we expect the method to be called.
Below will be a table of the methods that can be used as a parameter within the `verify()` method call.

Method	Function
times(n)	Tells Mockito that the method should be called *n* times
never()	Tells Mockito that the method should never be called
atLeastOnce()	Tells Mockito that the method should be called one or more times
atLeast(n)	Tells Mockito that the method should be called *n* or more times
atMost(n)	Tells Mockito that the method should be called *n* or less

If we were to modify our previous example with the parameter of `times(1)`, to make sure that our `getUniqueId()` method is only called once against our mocked object, it would look something like the following.

```
@Test
public void test() {
    Customer mockedCustomer = mock(Customer.class);
    when(mockedCustomer.getUniqueId()).thenReturn(23);
    int id = mockedCustomer.getUniqueId();
    verify(mockedCustomer, times(1)).getUniqueId();
}
```

Exercises

1. Using a project you have already created, add some tests that test the behaviour of your project, making sure that each function of your project is calling the correct methods.