Mockito Guide

# About

Unit testing involves testing functionality in isolation to reduce the effect of other classes or the system. Mockito is used in conjunction with JUnit to create test double objects or *mock* objects. Which proves to be very useful in *Top-Down programming* as it allows users to verify the behavior of the S*ystem Under Test* without fully implementing all the lower level dependencies.

Here, we will discuss how to import and use Mockito.

The remaining document is organized in the following manner:

1. First, a brief overview of some concepts and terms.
2. Next, an example of Mockito used in Spring.
3. Finally, an example of Mockito used in Android.

# Concepts/Terms

## Why/When to use Mockito

Mockito is meant to test classes which have lower-level dependencies without actually needing the dependencies. It allows you to *mock* the dependencies at runtime and define their behavior as needed. This ideology fits in with Top-Down programming because it ensures the higher-level code functions as intended. If your higher-level class works then you will not have to redesign the entire project if you find out later that your design has a major flaw.

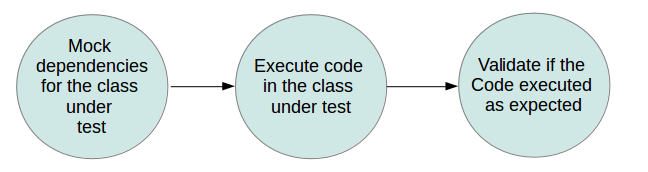
## System Under Test (SUT)

System Under Test refers to the code which is being tested by the unit tests you have written. However, most likely your code will only be testing individual classes so we can call it Class Under Test (CUT).

## Mocking

*Mocking* is the process of creating *mock* objects to abstract away the implementation details of a certain class. This allows classes to be tested without their dependencies.

## Typical Workflow



# Import Mockito into Android/Spring

## Android

// Add this dependency to your app level build.gradle

androidTestCompile "org.mockito:mockito-android:3.3.0"

## Spring Boot

<-- Add this dependency to your pom.xml -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.3.0</version>

</dependency>

# Mockito Examples

After you have imported Mockito, you will use the various methods and annotations provided by the framework to create mock objects. If you have been working with Spring Boot, you probably already have some experience working with annotations. The following are the key method and annotation:

* The static method “mock()”
* The annotation “@Mock”

The most common object to mock is some sort of data source/provider, like a database/repository. If you mock a data provider, you can provide test data to the SUT without affecting any existing data and allows for isolated testing. The mocked object is passed to the SUT using a method/constructor. Note, this requires the SUT to not include any hard dependencies.

## Generic Example

import static org.mockito.Mockito.\*;

public class MockitoTest {

@Mock

MyDatabase databaseMock;

@Rule

public MockitoRule mockitoRule = MockitoJUnit.rule();

@Test

public void mockTest() {

ClassToTest t = new ClassToTest(databaseMock);

boolean check = t.method("test"); // Runs some code from class being tested

assertTrue(check); // Checks the return value of the method

verify(databaseMock). method ("test"); // Verifies whether the method

// correctly reached the mocked object.

}

}

## Android

A key thing to remember when using Mockito with Android is to avoid testing methods and classes that you did not implement such as activities, already created views, etc. If your classes have a lot of Android framework dependencies, you might want to look into Roboelectric. However, you can test custom views, classes and interfaces that you have created and you can mock the context in which these classes function. You can even mock something like a SharedPreferences and check if data is being used correctly. For an example of mocking in Android, please look at this [**link**](https://github.com/android/testing-samples/blob/master/unit/BasicSample/app/src/test/java/com/example/android/testing/unittesting/BasicSample/SharedPreferencesHelperTest.java). \*Make sure to put your unit tests in the src/test/.. folder.

## Spring Boot

It is much easier to mock classes in Spring Boot because it follows the MVC pattern. Therefore, there are usually many data sources/data providers to mock and test. Typically, you will want to mock a repository, then test the controller methods on the mocked repository, and verify if the expected values match the actual values sent to the mocked repository. For an example, look at this **link**.

## Helpful links

<https://github.com/android/testing-samples/blob/master/unit/BasicSample/app/src/test/java/com/example/android/testing/unittesting/BasicSample/SharedPreferencesHelperTest.java>

<https://www.baeldung.com/mockito-series>

<https://developer.android.com/training/testing/unit-testing/local-unit-tests#mocking-dependencies>

# Acknowledgements

**If some parts of this document are not clear, or you feel that there could be more examples – please let us know on Piazza.**