**Core Object Oriented Development using Java**

OOD Week 2

Mockito

Tutorial

© FDM Group Ltd 2021. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this tutorial cover?

This tutorial will introduce you to the use of Mockito in TDD.

# How long will the tutorial take to complete?

2 hours

# What should you have already completed?

TDD / Junit 5

# What do you need?

In order to complete this tutorial exercise you will need:

* Java Development Kit 1.8 or above
* Apache Maven
* Eclipse IDE Kepler or above
* Junit 5
* Mockito-all 3.7.7 or above

# What does this tutorial cover?

* Reasons to use Mockito
* Stubbing
* Verifying behaviours

# Mocking Frameworks

Many modern Object Oriented languages have frameworks (libraries, if you prefer) that allow developers to utilise *Mocking* as part of the Test Driven Development Process. Java is no exception and whilst there are a number of mocking frameworks in Java, one of the most popular (and easiest to learn) is **Mockito**. The API can be found here:

## <https://javadoc.io/static/org.mockito/mockito-core/3.7.7/org/mockito/Mockito.html>

## What is mocking and why would I want to use it?

Mocking frameworks typically provide a number of benefits for when we are applying the TDD process.

### Testing Behaviour

**Question: What is it that JUnit really tests? Are there any limitations to the types of methods we can test with JUnit?**

**Answer: JUnit tests the return values of methods. JUnit *cannot* test void methods. This is a big limitation.**

Mocking frameworks allow developers to write tests that test the *behaviour* of their application. Behaviour here is defined to mean a *method call*. As such we can call a method within our application and then observe *which methods it calls* in other objects, *how many* *times* it calls those methods, and *the parameters that are passed to them*.

### Mock Objects

We can also use Mockito to produce M**ock Objects**. Mock objects are *fake* versions of real objects; they are hollow with no defined functionality. Methods inside mock objects perform no logic, if they are defined to return a value, they will simply return the default value for that type; zero for primitives or null for objects.

**Question: Why is this useful?**

**Answer: Isolation. They allow us to isolate parts of our system from one another.**

#### Scenario 1

Imagine you have just finished developing a class via unit testing that must make a series of calls to a database.

All of your tests pass, your code works as expected.

**Question: If the database crashes, should all your tests still pass? Does the fact that the database is down now mean that your code is incorrect?**

**Answer: Your code has not changed. Your tests *should* still pass. So we need a way that we can isolate our test code from the database. We will achieve this via mocking.**

#### Scenario 2

Imagine you’ve been given the job of writing a class which depends on an interface. Objects of concrete classes which implement the interface will be injected into your class. The problem is that those classes implementing the interface haven’t been written yet.

**So how do you test your code?**

You could wait until the other classes have been written, but this seems very inefficient.

Once again, the solution is to create mock objects of the classes which haven’t yet been written. In fact we can create a mock object directly from the interface. This will allow us to test our code immediately.

Note that even if the other classes our code depends on have already been written, it is still a good idea to use mock objects. This prevents coding errors in those other classes causing your tests to fail even though your code is good.

# Setting Up

Add the mockito dependency to the pom.xml of your project after your JUnit dependency:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.7.7</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>3.7.7</version>

<scope>test</scope>

## </dependency>

## Test Scope Reminder

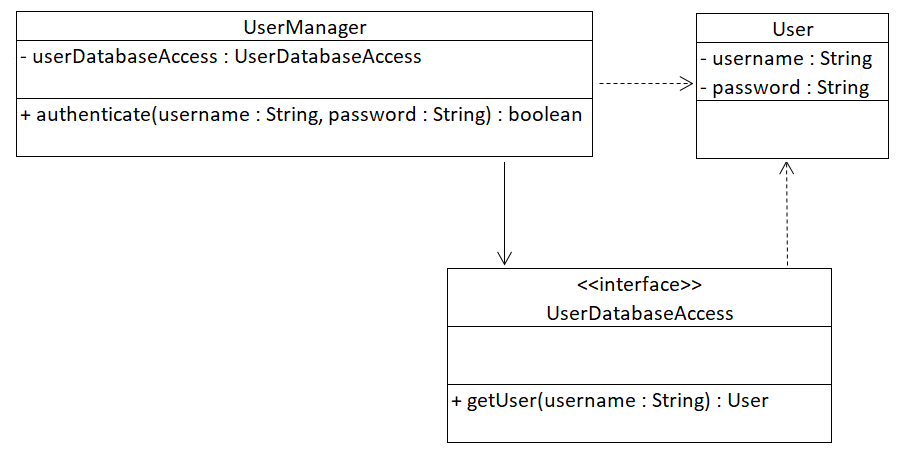
The <scope>test</scope> within our dependency tells our Maven project to only include these libraries when we are running our tests.

Remember: tests are for developers, not users. When we release our production code to our client base, we will not be sending our tests with it. As such, we shouldn’t include the JUnit or Mockito dependencies either.

# Mockito Example

## Scenario

In our example we will be writing part of an application which authenticates a user’s logon details and logs them on if they are correct. We will be writing the UserManager class in the UML below and testing its authenticate() method.



Here’s a summary of what the authenticate() method will do:

* When a user tries to log on, the username and password they enter will be passed to the authenticate method.
* The authenticate method will call the getUser() method of the UserDatabaseAccess object passing in the username. If the getUser() method finds a user with that name in the database it will return the user object, otherwise it will return null.
* The authenticate method will check if the user object’s username and password match the username and password entered by the user. If they do, the method will return true. Otherwise it will return false.

## Which objects are real and which are mock?

In our diagram, we can see that UserDatabaseAccess is an interface. So it doesn’t have any code. As you should already be aware, it’s not possible to instantiate an interface. There may not even be a real database for it to interact with yet. We’re going to make a mock object of UserDatabaseAccess to help our test run.

The User class is a concrete class, so we could make a real object from it. But we’re not going to. It’s possible that there are coding errors in the User class, so we’re going to make a mock object of that just in case.

In fact the only real object in our test case is going to be of the UserManager class. In general all of our test cases should be like this. We should create a real object of the class being tested and mock objects of everything else.

### Setting up

Under src/main/java create a package called com.fdmgroup.mockitoTutorial.

Create a package of the same name under src/test/java.

In the test package, create a Junit test case called TestUserManager.

In the main package, create the UserDatabaseAccess interface and the User class as shown in the UML. Generate getters and setters for the User class. **Do not create the UserManager class yet**. Remember that we are doing TDD so we’re going to write the tests for the the UserManager class before we write any code.

### Creating objects for our test case

Let’s start by creating 2 mock User objects and one mock UserDatabaseAccess object. You’ll need to import the @Mock annotation to make this compile:

@ExtendWith(MockitoExtension.**class**)

**class** TestUserManager {

@Mock

User mockUser1, mockUser2;

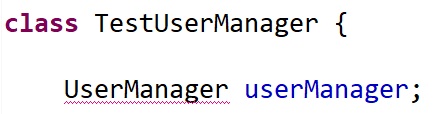
@Mock

UserDatabaseAccess mockUserDatabaseAccess;

Let’s explain what’s happening:

* @Mock allows us to create references to mock objects, but it doesn’t create the mock objects themselves.
* The @ExtendWith annotation above the class header is what creates the mock objects.

Finally, let’s create the one real object: UserManager:



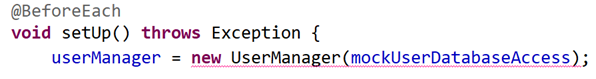
Notice the compile error on UserManager. As you’ll remember this is because we haven’t created the class yet. Use the Eclipse auto-complete to create the UserManager class.

### Dependency injection

UserDatabaseAccess is a member variable of the UserManager class. We’re going to use UserManger’s constructor to inject a UserDatabaseAccess object into the UserManger.

In our tests we can feed the constructor the mock UserDatabaseAccess object. After our testing is complete, we can feed it a real object of one of the concrete classes implementing UserDatabaseAccess.

When we write the code, we can see that there’s a compile error because currently UserManager doesn’t have a constructor:



The auto-complete can create a constructor and fix the compile error. However it will just generate an empty constructor. To make the constructor work, we should now go into the UserManager class and fill in the code:

**public** **class** UserManager {

UserDatabaseAccess userDatabaseAccess;

**public** UserManager(UserDatabaseAccess userDatabaseAccess) {

**this**.userDatabaseAccess = userDatabaseAccess;

}

}

At this point we’ve created all of the objects we’re going to need for our tests. We’re now ready to write the first test.

## Test 1 – testing a simple behaviour

In our first test we’re going to check that the authenticate() method calls the mock UserDatabaseAccess.getUser() method once, passing in the username which was passed into authenticate().

This test is going to be very different to the tests you’ve previously done. There will be no assert. Instead we’re going to use Mockito’s ‘verify’ method.

Before we can use verify, we need to manually add the following import to our test case:

**import** **static** org.mockito.Mockito.\*;

Now let’s write the test:

@Test

**void** test\_authenticate\_callsGetUserWithArgUserOne\_whenUserOnePassedIn() {

**boolean** authenticated = userManager.authenticate("user.one","password1");

*verify*(mockUserDatabaseAccess).getUser("user.one");

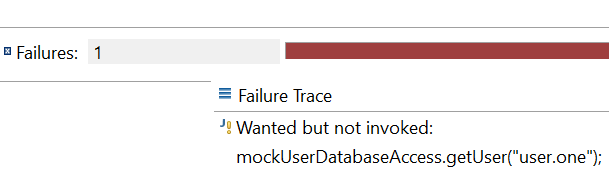
}

You’ll notice that there’s a compile error when the authenticate method is called. This is because the method doesn’t exist yet. Use the auto-complete to create it.

Now let’s look at the verify statement in detail. The verify checks that the mockDatabaseAccess object’s getUser() method has been called with the argument “user.one” at some point in this test.

There’s only one line of code in the test before the verify. This calls the authenticate() method of userManager. For the test to pass, the authenticate method will need to call the getUser() method passing in “user.one”.

If we run the test now it will fail and we’ll see the message ‘wanted but not invoked’ in the Failure Trace:



This is exactly what we want to see at this stage. We haven’t written any code in the authenticate() method so it’s a good thing that the test has failed.

Now let’s write the code in the authenticate() method to pass the test:

User user = userDatabaseAccess.getUser(username);

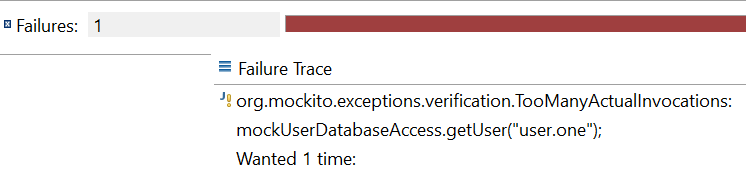
If we now run the test again it will pass.

Out of interest, let’s now call the getUser() method a second time in our authenticate() method:

User user = userDatabaseAccess.getUser(username);

user = userDatabaseAccess.getUser(username);

This time if we run the test, it fails with the following error:

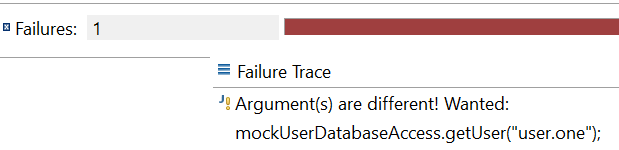


Our verify statement will only pass if the getUser() method is called exactly once.

Let’s try another experiment. This time in the authenticate() method we’ll hard code the call to getUser() with the wrong username:

User user = userDatabaseAccess.getUser("xyz");

This time when we run the test we get a different error:

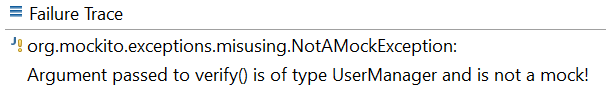


The test will only pass if the correct argument is passed into the getUser() method.

To summarise the verify statement works as follows:

verify(mockObject).method(argument);

Note that verify will only work on a mock object. If you pass in a real object you’ll get the following error:



Before continuing with the next section, make sure you remove all the bad code we wrote on this page and ensure that the test now passes.

## Mock object methods

Before we write any more tests, let’s take a closer look at how a mock object works.

Let’s create a dummy test so that we can see what happens when a method from a mock object is called:

@Test

**void** dummyTest() {

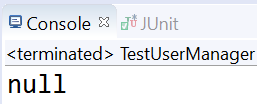
mockUser1.setUsername("dummy.user");

System.***out***.println(mockUser1.getUsername());

}

We don’t care whether this test passes or fails, we just want to see what is printed out in the console when getUsername() is called on mockUser1.

When we run the test, this is what we get in the console:



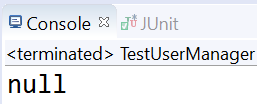
Let’s go one step further and hard code a value into the getUsername() method of the User class:

**public** String getUsername() {

**return** "dummy.username";

}

If we run the test again, we still get a null value:



So what’s going on here? Well, the answer is very simple. In a mock object, all code in all methods is ignored. So when we call a void method like setUsername() it does nothing.

When we call a method with a return type, it returns the default value for that return type. So as getUsername() returns a String, we get the default value of a String i.e. null. A method which returned an int would return 0 and a method returning a double would return 0.0.

Right now you’d be justified in asking “how is any of this useful?”. If methods in mock objects only ever returned null or zero, they wouldn’t be very useful. In the next section, we’ll look at how to get mock object methods to return other values.

Before moving on to the next section, delete the dummy test that we used in this section.

## Stubbing

In this section we’re going to look at how to hard code a variety of return values for our mock objects. This process is called stubbing and uses a Mockito method called ‘when’.

Note that ‘when’ requires the same manual import as ‘verify’.

In our tests, when we call the getUser() method of our mockUserDatabaseAccess object with the argument “user.one” we want the method to return the mockUser1 object. If we call it with the argument “user.two”, we want it to return the mockUser2 object.

Right now, whether we call getUser() with “user.one” or “user.two” all we’re going to get back is null. Let’s change that now by using ‘when’.

Add the following code to the setup() method:

*when*(mockUserDatabaseAccess.getUser("user.one")).thenReturn(mockUser1);

*when*(mockUserDatabaseAccess.getUser("user.two")).thenReturn(mockUser2);

These two lines have hard coded return values for specific arguments fed into the getUser() method.

If now if we call getUser() with either “user.one” or “user.two” it will return the object we want. If we were to call getUser() with another value e.g. “user.three”, it would still return null.

Let’s do the same thing for the mock User objects:

*when*(mockUser1.getUsername()).thenReturn("user.one");

*when*(mockUser1.getPassword()).thenReturn("password1");

*when*(mockUser2.getUsername()).thenReturn("user.two");

*when*(mockUser2.getPassword()).thenReturn("password2");

Notice that we’re only doing this for the getter methods. The setter methods are void, so there is no return value to hard code. In fact if you try using when with a setter method it will give you an error.

We now have enough hard coded return values that we can start writing some more tests for our UserManager class.

## Test 2 – Testing a return value with assertEquals

In our next test, we’ll check that if a valid username and password are passed into the authenticate() method, the method will return true.

@Test

**void** test\_authenticate\_returnsTrue\_whenUserOneAndPasswordOnePassedIn() {

**boolean** authenticated = userManager.authenticate("user.one","password1");

*assertTrue*(authenticated);

}

Notice that this is a very simple test which focuses on the ‘act’ and ‘assert’ stages. Following our stubbing in the setup method, there is no more need to mention Mockito in this test or any of our further tests.

When we run the test it will fail as the authenticate() method is hard coded to return false.

Right now, we’re not going to write the code to make the test pass. We’ll save this until we’ve written all the tests.

## Further tests

The previous test could easily be passed simply by hardcoding the return statement to true. This wouldn’t be very helpful. We need a few more tests to guarantee that the code will behave exactly as we want it to.

Add the following 5 tests:

@Test

**void** test\_authenticate\_returnsFalse\_whenUserOneAndPasswordTwoPassedIn() {

**boolean** authenticated = userManager.authenticate("user.one","password2");

*assertFalse*(authenticated);

}

@Test

**void** test\_authenticate\_returnsTrue\_whenUserTwoAndPasswordTwoPassedIn() {

**boolean** authenticated = userManager.authenticate("user.two","password2");

*assertTrue*(authenticated);

}

@Test

**void** test\_authenticate\_returnsFalse\_whenUserTwoAndPasswordOnePassedIn() {

**boolean** authenticated = userManager.authenticate("user.two","password1");

*assertFalse*(authenticated);

}

@Test

**void** test\_authenticate\_returnsFalse\_whenUserThreeAndPasswordOnePassedIn() {

**boolean** authenticated = userManager.authenticate("user.three","password1");

*assertFalse*(authenticated);

}

When you run the tests, you’ll notice that all the tests with assertTrue fail and all the tests with asssertFalse pass. This is because the authenticate() method is hard coded to return false. This is not a problem. When you’re testing a method which returns a boolean, this will always happen.

Let’s write the code to pass all the tests:

**public** **boolean** authenticate(String username, String password) {

User user = userDatabaseAccess.getUser(username);

**if** (user != **null** && password.equals(user.getPassword())) {

**return** **true**;

}

**return** **false**;

}

We can now be certain that the code in our authenticate() method works correctly. Thanks to Mockito we’ve been able to do this even though we don’t have any concrete UserDatabaseAccess class or even a database.

When the code is eventually used in a live application with real objects and a real database it will definitely work.

In the next section, we’ll move onto a different example to see how we can test more complex behaviours.

# A different Mockito example

In the previous example, we tested a simple behaviour in our first test. In this example, we’re going to test a more complex behaviour.

We’ve added an extra method to our UserDatabaseAccess interface. The new method is called addUser(). It takes a User object as an argument and writes it into the database.

The class we’re testing is called UserAdmin. Its createMultipleUsers() method takes a list of usernames and a default password. It will create User objects for each username in the list and give each account the default password. It will then pass each newly created User object to the addUser() method to be written to the database.

## 

Notice that createMultipleUsers is a void method, so there’s no return value for us to check. As we saw in our previous example, UserDatabaseAccess is an interface, so it won’t actually write anything to a database.

Before continuing, add the addUser() method to the UserDatabaseAccess interface. Don’t create the UserAdmin class yet. Remember that this is TDD so we’re going to write our test case first.

## Creating the test case

Create a new test case called TestUserAdmin.

Manually add the import required for verify.

In the test case, we’ll create a real UserAdmin object and a mock UserDatabaseAccess object. This time we won’t be creating any user objects.

Use the auto-complete to create an empty UserAdmin class and then to create a constructor.

Ensure that you give the UserAdmin class a UserDatabaseAccess member variable and set it in the constructor.

If you can’t remember how to do any of these things, refer back to the previous example.

## Test 1 – testing multiple calls to a method

In our previous example, we used a verify statement to check that a method was called exactly once. In this test we’re going to check that the addUser() method is called once for each name in the List of usernames passed into the createMultipleUsers() method. Specifically, we’ll pass a list of 3 names into createMultipleUsers() and check that addUser() is called 3 times.

The test will look like this:

@Test

**void** test\_createMultipleUsers\_callsAddUserThreeTimesPassingInAUser\_whenListWithThreeNamesPassedIn() {

List<String> userNames =

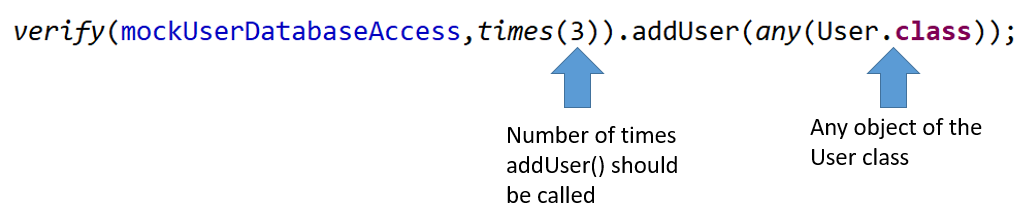
**new** ArrayList<String>(Arrays.*asList*("user1","user2","user3"));

userAdmin.createMultipleUsers(userNames,"defaultPassword");

*verify*(mockUserDatabaseAccess,*times*(3)).addUser(*any*(User.**class**));

}

Let’s look in more detail at the verify statement:



If we run the test it will fail with the ‘wanted but not invoked’ error.

Let’s add the code required to pass the test:

**public** **void** createMultipleUsers(List<String> userNames, String password) {

**for** (String username : userNames) {

User user = **new** User();

user.setUsername(username);

user.setPassword(password);

userDatabaseAccess.addUser(user);

}

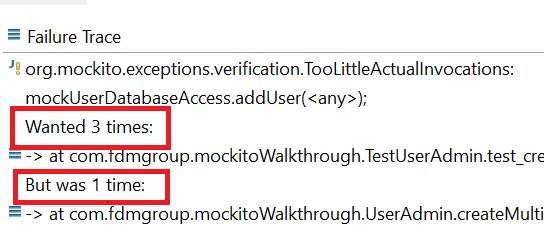
}

If we run the test, it will now pass.

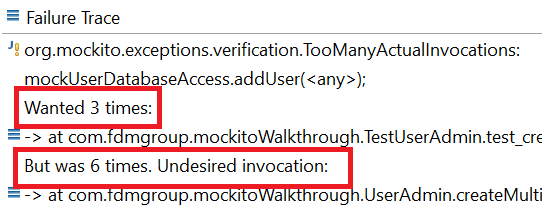
### Proving the verify statement checks for exactly 3 calls

Let’s now double check that the verify statement really does require 3 calls to the addUser() method. We’ll do this by adding some bad code to our createMultipleUsers() method.

Let’s start by adding a break statement at the end of our loop. This will mean that the addUser() method is only called once. If we run the test, it will fail with the following message:



Now we’ll remove the break statement and instead we’ll call the addUser() method twice in the loop. If we run the test again it will fail with this message:



We’ve just proven that the test will only pass if the addUser() method is called exactly three times. Now remove the extra call to addUser(), re-run the test and ensure that it passes before moving on.

### Matchers

In our test, the call to addUser() didn’t specify a specifc User object. Instead it used the following syntax:

addUser(*any*(User.**class**))

As long as an object of the User class is passed into the addUser() method the right number of times, the test will pass.

Any(User.class) is known as a ‘matcher’. It simply checks that the object passed in matches the correct class. Mockito has a few built in matchers for common classes such as anyString(), anyInt() and anyDouble()

### Further tests

We’ve tested that the addUser() method is called exactly 3 times when a List of 3 User objects is passed in. On it own this test is not enough as it’s possible to pass it by hardcoding 3 calls to the addUser() method.

To prevent this we should write another test passing in a List with a different number of objects. E.g. pass a List of 5 names and check that addUser() is called 5 times.

## Other ways of using verify

verify(object,never()).method() checks a method has not been called

verify(object,atLeastOnce()).method() checks a method has been called at least once