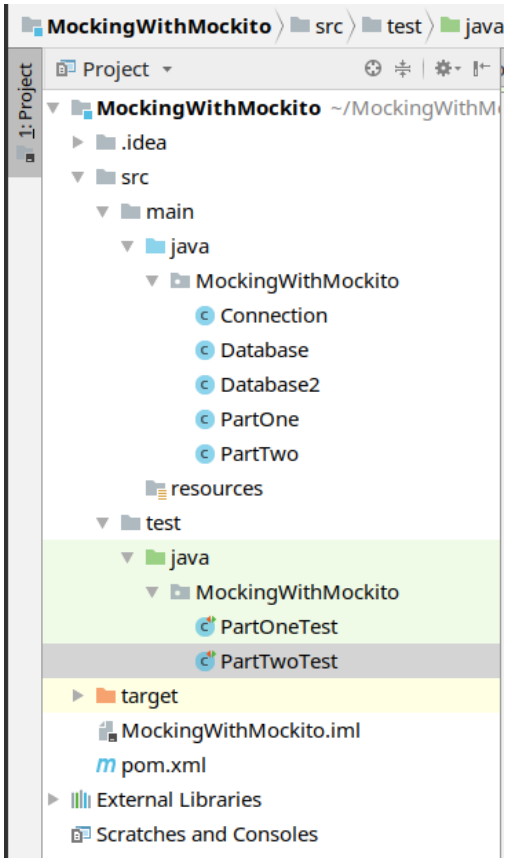
Tutorial: Mocking with Mockito

As you learned from the reading, mocking allows you to test classes in isolation from other classes they depend on. We typically use mocking frameworks to simplify the creation and use of mocks. There are several mocking frameworks available. Mockito is one of the most popular for mocking Java code and is the one we will use in this tutorial.

Setup

1. Create a new Maven or Gradle project and name it MockingWithMockito.
    a. Use whatever Group ID you want.
    b. Import the latest non-beta version of mockito-core and junit-jupiter-api dependencies
    c. Download and unzip the code for this tutorial.
2. Copy or move the classes in the source_files directory into the project's 'src/main/java' directory.
3. Copy or move the tests in the test_files directory into the project's 'src/test/java' directory. You should end up with a project structure that looks like this:



4. Open the PartTwoTest class in ‘src/test/java’. If you see an error that says an expression isn’t supported at your language level (as shown in the following image), follow the instructions for “Intellij Reverts to An Earlier Java Version After you Specify a Latter One” in [this document].

5. You should now have a project with no compile errors. If you still see errors, get help from the instructor, a TA, or another student before continuing.

Part One: Generating mocks, Returning preset values and using argument matchers

Start by looking over the PartOne class in the main/java folder and reading the comments so you have an understanding of the class we are testing. The class is intended to use a database to do a user login. The database doesn’t have enough code to do anything useful yet. We’ll set up a mock that we can use in place of a functioning database class so we can test classes that use Database even before Database has any real functionality.

1. Open PartOneTest and let’s start by creating a mock. Mocks are objects that mimic an actual class. They can be set up with the structure: SomeClass variable = Mockito.mock(SomeClass.class);
2. Insert the line “`databaseMock = Mockito.mock(Database.class);`” in the setUp method as the first line of code. This will create a mock of the database class and store it in the variable databaseMock. You will have to import org.mockito.Mockito

3. We haven't yet told our PartOne object that this is the database that it should use. Add the line `partOne.setDatabase(databaseMock);` below the line that creates the partOne instance to tell our class to use the mock.
4. Our mock is now being used, but it doesn't do anything yet. Whenever a method is called on the mock it will just throw a null pointer exception. We will tell it to return a preset value when the getUser(...) method is called.
5. Go to the loginTest() method and insert the following as the first line in the method: `Mockito.when(databaseMock.getUserId("john", "pass123")).thenReturn("existingUserUUID");` This is telling the mock "Whenever the method getUser() is called with the parameters 'john' and 'pass123' then return the value 'existingUserUUID' ".
    a. **Note:** uuid stands for universally unique identifier. You don't need to memorize this, it's just useful to know for this particular problem.
6. If you run the test for loginTest() it should pass. This is testing the condition on line 16 of the PartOne class where the database returns a user Id for a user it finds on login.
7. Now uncomment the two commented lines in loginTest. This will test the condition on line 14 of the PartOne class (where the user is not found in the database).
8. Right now, John is the only one who has a value in our mock database. We want to test what happens when someone besides 'john' tries to log in. If you run the test right now it will fail because the mock doesn't know what to do when the login function is called for "newUser".
9. When specifying argument values for a method you want to mock with 'Mockito.when' you can use something called an argument matcher which allows you to specify wildcard like values in place of the expected argument values. We will use the anyString() argument matcher to match any String value that is passed in for the parameters. This allows us to specify a different return for any parameters that are not "john" and "pass123" which were specified in a previous step. Add the line `Mockito.when(databaseMock.getUserId(Mockito.anyString(), Mockito.anyString())).thenReturn("");` at the very beginning of the loginTest() function. Now whenever you call get user on anything that isn't john it will return "".
10. Now we need to set up the addUser method on our database mock. Lets use our argument matchers again. Add the line `Mockito.when(databaseMock.addUser(Mockito.anyString(), Mockito.anyString())).thenReturn("newUserUUID");` near the top of your loginTest() method (in your PartOneTest class. You can put it anywhere before the point where it gets invoked in the test. Put it **before** the two existing Mockito.when(...) statements.
11. Run your test again and it should pass.

    What we have confirmed so far is that assuming we have a Database class with properly functioning getUserId and addUser methods, our PartOne class' login method works properly. Notice that we are able to confirm that even before we have a functioning Database class by using a mock. Even after we have a functioning Database class, we would continue to use the mock for this test to keep it's test independent of whether we

have a functioning Database class, and to make the test run faster than it would run if we were actually accessing a database.

Part Two: Verification and Exceptions

1. Open the PartTwoTest class. First we will learn to use a mock to verify that methods either were or were not called as expected on a Mock. We will use verify methods to do this. Sometimes code has no quick way for us to know what methods were actually called based on our results. Verification allows us to know for sure whether a method was called and how many times it was called.
2. If you run the test for logAccountUse it will pass, but you may notice this is because we don't have any assertions. Look at logAccountUse method of PartTwoTest.
3. You may have noticed that logAccountUse returns void. Methods that return void can be difficult to test in isolation. If our database class was complete we may be able to pull some data that shows that logAdminUse, logStudentUse or logStaffUse were called, but that can create some messy, long, and highly dependent tests.
4. We can use Mockito.verify to assert that expected method calls were actually made or were not made on a mock object.
5. First add these lines of code at the end of the test:
```
Mockito.verify(databaseMock).getAccountType("ABC-123");
Mockito.verify(databaseMock).getAccountType("BIG-CHUNG");
Mockito.verify(databaseMock).getAccountType("A-UUID");
Mockito.verify(databaseMock).getAccountType("NOT_A_UUID");
Mockito.verify(databaseMock).getAccountType("BAD_TEST");
```
When verify is called on a mock object it looks at the mock object to confirm that the verified method was called the specified number of times (defaulting to one time) with the specified parameter(s).
6. Run the tests now. You should see that the logAccountUse() test failed. This is because we verified one method call with a uuid of "BAD_TEST" which we never call in our code. Remove that call to verify and run the tests again. They should all pass.
7. Now let's add some verifications to verify that the methods in the PartTwo class' switch statement were called. Add the following to the end of the logAccountUse test:
```
Mockito.verify(databaseMock, Mockito.times(2)).logAdminUse("ABC-123");
Mockito.verify(databaseMock, Mockito.times(1)).logStudentUse("BIG-CHUNG");
Mockito.verify(databaseMock, Mockito.times(1)).logStaffUse("A-UUID");
```
8. Run the tests to see them fail again. You'll notice that we get an error telling us that logAdminUse was called once, not twice. Change the times from 2 to 1 and the tests will pass.
9. Now we have seen that we can make our tests pass and fail by using verify with Mockito. This allows us to test functions that are void or are normally difficult to test.
10. For the final part of the lab, we will see how to make mocks throw exceptions in response to method calls. We have two methods we still need to test: login and register.

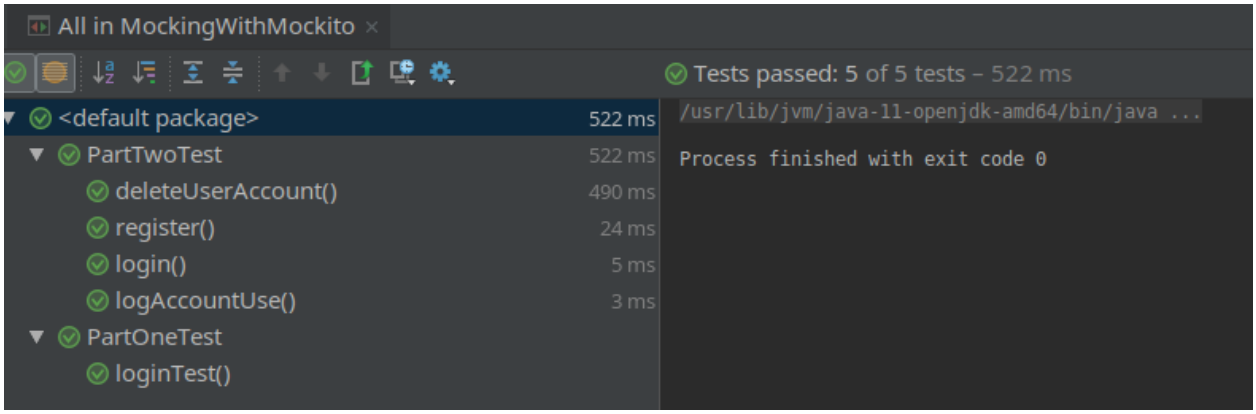
The login() method will return a blank string when encountering an exception and register will throw an exception when encountering an exception.

11. For login() you have some provided code much like we had in our partOne tests. Uncomment the assertion on line 25 of the PartTwoTest and run the tests again. You will see that the login test fails because we haven't made our Database mock object throw an exception when login is called with the values that assertion uses.
12. Add this line of code to the login test above the first assertion and below the previous when: `Mockito.when(databaseMock.getUserId("Trinity","not-here")).thenThrow(SQLException.class);` Using when...thenThrow ensures that given a certain input we will throw the specified exception. While we do not use it in this part of the code, there is a different syntax for making mocked methods that return void throw an exception. The format is `doThrow(new Exception()).when(mockObject).functionCall();`
13. Run the test to confirm that they all pass.
14. We can also verify that a called method throws an expected exception. You can see this in the register() test. In this case, we set up our database mock to throw an exception when it's addUser method is called. Notice the register method in the PartTwo class will call addUser on line 25. We verify that the expected exception is thrown from the register method in the register test. The syntax is a little beyond the scope of this class. The second parameter to the assertThrows assertion is a lambda function that calls the method we are expecting to throw the exception. The first parameter is the class of the exception we expect to have thrown.
15. Finally we have provided you with a blank test template for deleteUserAccount. Write your own test case for this method. You will need to use when and verify to confirm that the deleteUserAccount method is called, and that when it returns false, the deletion attempt is logged (by calling the Database's logErrorMessage method with the appropriate log message). Also confirm that when an SQLException is thrown, the Database's logErrorMessage method is called with the specified String to log the exception.
    a. In other words, you should be have a test for a normal deleteUserAccount, a test where deleteUserAccount returns false, and a test that throws a SQLException

Submission

Submit your two test classes (with the changes you made in this lab) and a screenshot showing all of your tests passing.

Example Screenshot:


All in MockingWithMockito
Tests passed: 5 of 5 tests – 522 ms
<default package> 522 ms
PartTwoTest 522 ms
deleteUserAccount() 490 ms
register() 24 ms
login() 5 ms
logAccountUse() 3 ms
PartOneTest
loginTest()
/usr/lib/jvm/java-11-openjdk-amd64/bin/java ...
Process finished with exit code 0