Assignment: Mocking test dependencies

In layered designed applications it's very common that a class (in business layer for instance) depends on other classes (from persistence layer for instance) that triggers database queries or maybe save files or send emails and etc.

When unit testing the focus is on the class under test, the so-called unit. We're not interested in triggering these dependencies.

By using dependency inversion, we are already in a good position in order to unit test these classes. We can create a Fake implementation of an interface and pass a instance of it to the class under test via its constructor. You've been doing this since pass semester and well, as you know, implementing Fake classes can be a lot of work.

On this assignment you will learn a better way of doing this!

Mockito is a powerful framework you can use to professionalize your back-end testing. You can mock interfaces and validate methods have been executed.

Download the starter project here.

Difficulty: ☆☆☆

Learning objectives:

- I can use Mockito as mocking framework to improve my back-end testing
- I can create mocks in my unit tests
- I can specify mock's behavior for each test
- I can verify the expected mock behavior
- I can use JUnit 5 Mockito's extension, @Mock and @InjectMocks annotations in order to automatically create mocks and inject them in the class under test
- I can unit test my app's business layer classes mocking dependencies

Step 1: Getting familiar with Mockito

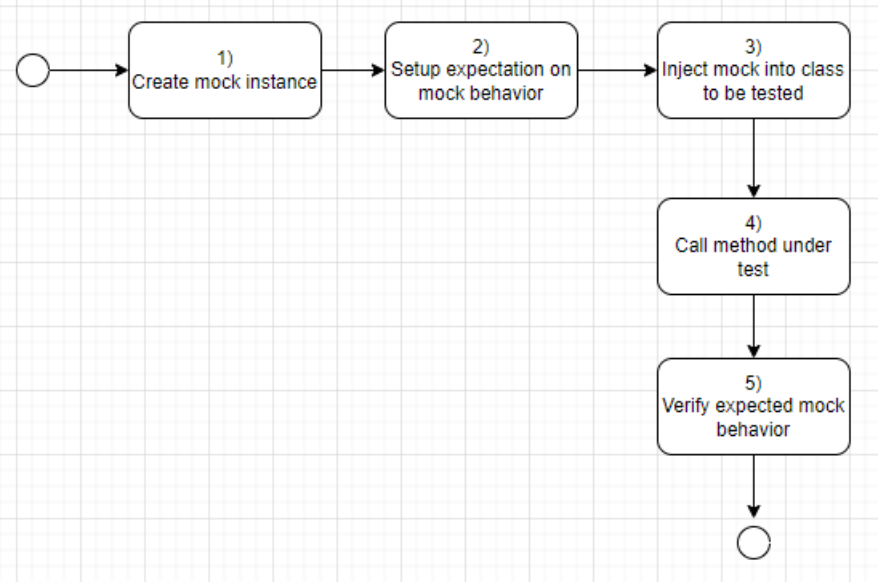
Got to Mockito's website and check the Why and How sections.

After reading these sections try answering with your own words:

1. Why Mockito in Java/Android projects?
2. How does mocking a class or interface looks like?

Step 2: Mocking lifecycle

When working with mocks usually the following steps are usual:



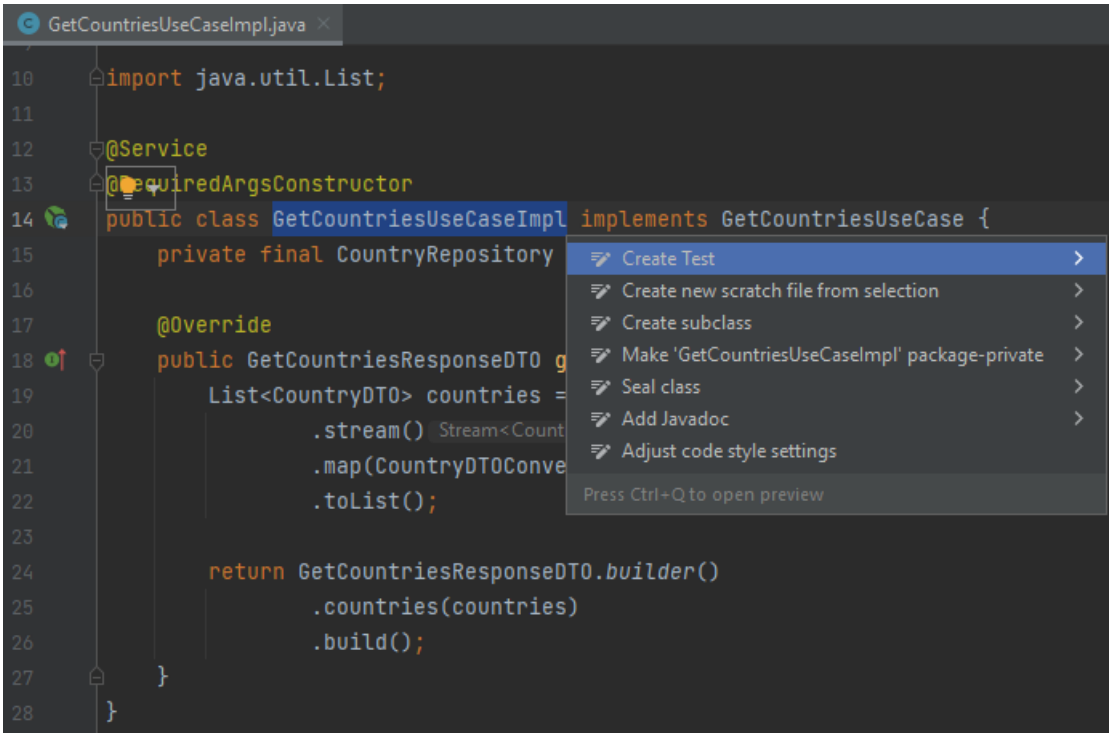
Keep this in mind for the following steps.

Step 3: Testing class *GetCountriesUseCaseImpl*

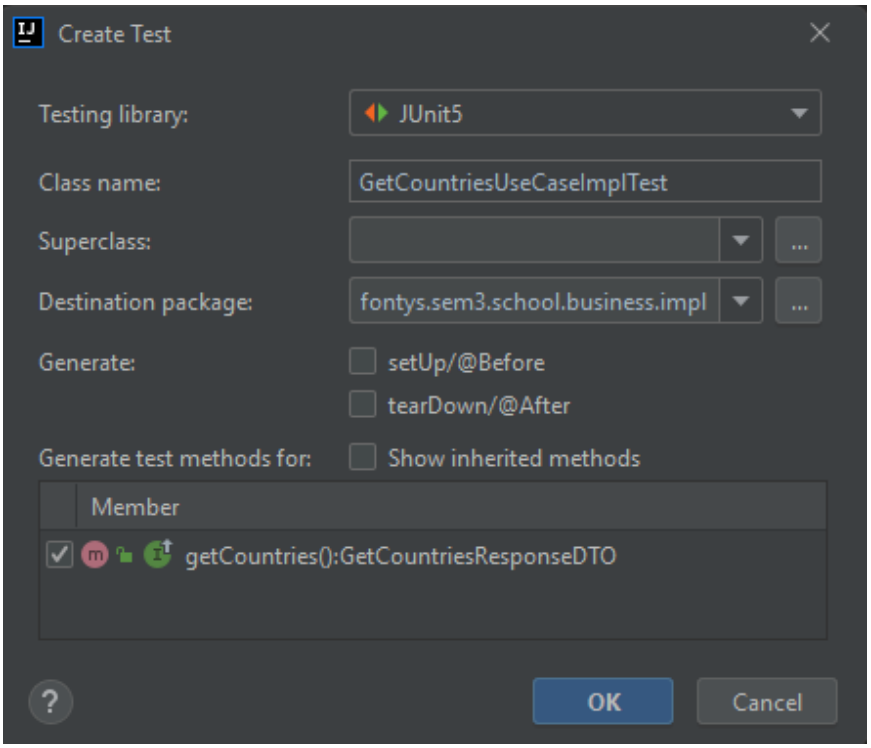
We will now create a unit test for class GetCountriesUseCaseImpl (it already exists in the starter project).

```
@Service
@RequiredArgsConstructor
public class GetCountriesUseCaseImpl implements GetCountriesUseCase {
    private final CountryRepository countryRepository;

    @Override
    public GetCountriesResponse getCountries() {
        List<Country> countries = countryRepository.findAll()
                .stream()
                .map(CountryConverter::convert)
                .toList();

        return GetCountriesResponse.builder()
                .countries(countries)
                .build();
    }
}
```

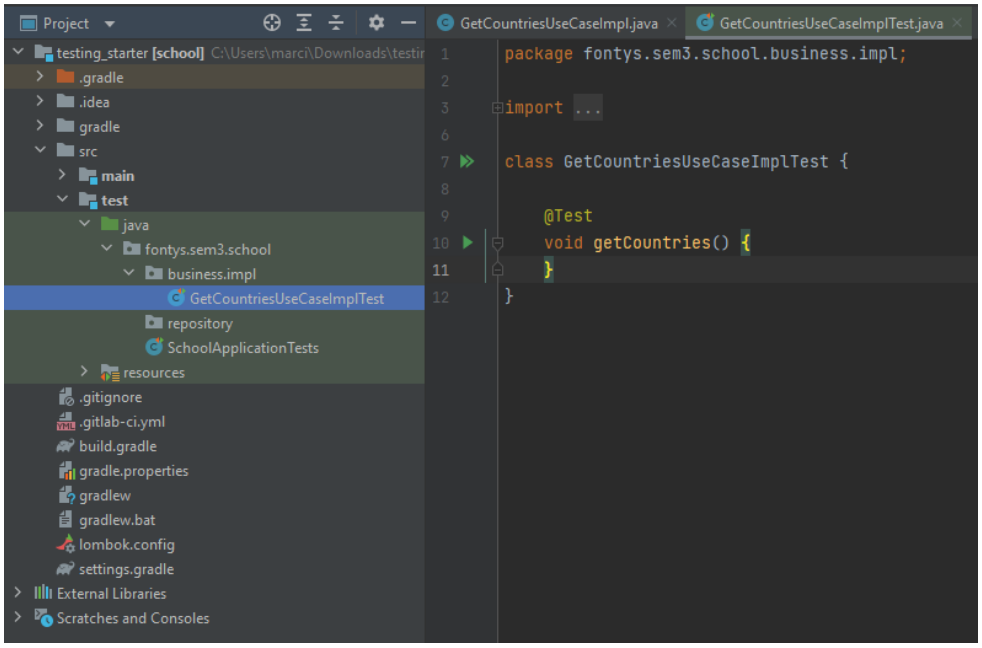
This can be easily done by right-clicking on the class name and selecting “Create Test”. See the image below:



Then select the methods you want to test and OK.



IntelliJ will create the test class in the correct test folder automatically:



What's happening in method GetCountriesUseCaseImpl.getCountries()?

1. Uses *countryRepository* in order to fetch all the countries;
2. converts each country JPA entity into a Country class;
3. finally returns a response object with the countries converted

Nothing special with steps 2 and 3 but step 1 involves a dependency that normally would trigger database calls. That's exactly what we would like to mock!

This is how our test will look like:

```
package fontys.sem3.school.business.impl;

import fontys.sem3.school.domain.Country;
import fontys.sem3.school.domain.GetCountriesResponse;
import fontys.sem3.school.repository.CountryRepository;
import fontys.sem3.school.repository.entity.CountryEntity;
import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

class GetCountriesUseCaseImplTest {

    @Test
    void getCountries_shouldReturnAllCountriesConverted() {
        CountryRepository countryRepositoryMock =
mock(CountryRepository.class);

        CountryEntity netherlandsEntity =
CountryEntity.builder().id(1L).code("NL").build();
        CountryEntity brazilEntity =
CountryEntity.builder().id(2L).code("BR").build();
        when(countryRepositoryMock.findAll())
                .thenReturn(List.of(netherlandsEntity, brazilEntity));
```

```
        GetCountriesUseCaseImpl getCountriesUseCase = new
GetCountriesUseCaseImpl(countryRepositoryMock);
        GetCountriesResponse actualResult =
getCountriesUseCase.getCountries();

        Country netherlands = Country.builder().id(1L).code("NL").build();
        Country brazil = Country.builder().id(2L).code("BR").build();
        GetCountriesResponse expectedResult = GetCountriesResponse
                .builder()
                .countries(List.of(netherlands, brazil))
                .build();
        assertEquals(expectedResult, actualResult);

        verify(countryRepositoryMock).findAll();
    }
}
```

Let's break it down with the mocking lifecycle in mind.

	Code	Explanation
1	`CountryRepository countryRepositoryMock = mock(CountryRepository.class);`	Static method *mock* creates a mock instance of type CountryRepository.
2	`when(countryRepositoryMock.findAll())` `    .thenReturn(List.of(netherlandsEntity, brazilEntity));`	Static method *when* receives the expected mocked method invocation and *thenReturn* configures what the invocation should return. In this case when method findAll of the mock is invoked it should return a list of two countries: Netherlands and Brazil.
3	`GetCountriesUseCaseImpl getCountriesUseCase = new GetCountriesUseCaseImpl(countryRepositoryMock);`	Instantiates the class to be tested passing the mocked repository in the constructor.
4	`GetCountriesResponse actualResult = getCountriesUseCase.getCountries();`	Invokes the method under test and assigns the result to a variable.
5	`verify(countryRepositoryMock).findAll();`	Verifies that method findAll from countryRepositoryMock was invoked a single time as expected.

Now that you understand the code, make sure your test runs smoothly!

Step 4: Using Junit 5 Mockito Extension

Mockito provides a Junit extension that simplifies writing unit tests like the previous one.

It takes care of steps one and three of the mocking lifecycle. This is how the same test using this extension looks like:

```
package fontys.sem3.school.business.impl;

import fontys.sem3.school.domain.Country;
import fontys.sem3.school.domain.GetCountriesResponse;
import fontys.sem3.school.repository.CountryRepository;
import fontys.sem3.school.repository.entity.CountryEntity;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
class GetCountriesUseCaseImplTest {

    @Mock
    private CountryRepository countryRepositoryMock;

    @InjectMocks
    private GetCountriesUseCaseImpl getCountriesUseCase;

    @Test
    void getCountries_shouldReturnAllCountriesConverted() {
        CountryEntity netherlandsEntity =
CountryEntity.builder().id(1L).code("NL").build();
        CountryEntity brazilEntity =
CountryEntity.builder().id(2L).code("BR").build();
        when(countryRepositoryMock.findAll())
                .thenReturn(List.of(netherlandsEntity, brazilEntity));

        GetCountriesResponse actualResult =
getCountriesUseCase.getCountries();

        Country netherlands = Country.builder().id(1L).code("NL").build();
        Country brazil = Country.builder().id(2L).code("BR").build();
        GetCountriesResponse expectedResult = GetCountriesResponse
                .builder()
                .countries(List.of(netherlands, brazil))
                .build();
        assertEquals(expectedResult, actualResult);

        verify(countryRepositoryMock).findAll();
    }
}
```

Notice annotation @ExtendWith(MockitoExtension.class) on the class level and also the fields annotated with @Mock and @InjectMocks.

Before running each test *MockitoExtension* creates automatically mock instances and assign them to fields annotated with @Mock.

The field annotated with @InjectMocks is of the type we want to test and will be instantiated with all the declared mocks automatically by *MockitoExtension* before each test.

The rest of the code is pretty much the same. Change the test to use this extension and check if it works fine for you!

Step 5: Testing class *CreateCountryUseCaseImpl*. Your turn!

Class *CreateCountryUseCaseImpl* also need testing. It has a bit more logic than the previous class but nonetheless you should make sure all code flows are working as expected (one test method per flow!).

Just as in the previous example create a new test class for it and use Mockito's help for mocking dependency *countryRepository* again.

Tip: In order configure a mock to throw an exception when a specific method call occurs you should use *thenThrow* like in the example below:

```
when(myMock.myMethod(expectedParameter))
        .thenThrow(new MyException());
```

Step 6: Testing class *DeleteCountryUseCaseImpl*. Your turn!

Class *DeleteCountryUseCaseImpl* also need testing. The logic there is very simple, it only calls repository's method:

```
void deleteById(ID id)
```

When mocking void methods the *when* (setup mock behavior) part should be skipped. But make sure you *verify* the mock method was called with the expected parameter!

Please implement a new test class for *DeleteCountryUseCaseImpl* and make sure it works as expected.

Step 7: I want to know more…

Mockito indeed has more to offer but explore more as you need. Here is the main reference with all supported features. IntelliJ's autocompletion also helps a lot!

Final step: Use Mockito as part of your test workflow

Now that you know how to use a mocking library there is no need any more to create your own “Fake” class implementations for unit testing.

Time to delete them from your individual project and do proper mocking in your unit tests with Mockito!