FREE eBook

LEARNING mockito

Free unaffiliated eBook created from
Stack Overflow contributors.

#mockito

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: mockito

It is an unofficial and free mockito ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official mockito.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with mockito

Remarks

Mockito is a java Mocking framework that aims at providing the ability to write clean an readable unit tests by using it's simple API. It differs from other mocking frameworks by leaving the expect-run-verify pattern that most other frameworks use.

Instead it only knows one way to mock (non-final) classes and interfaces and allows to verify and stub based on flexible argument matchers.

The current Version 1.10.19 is best obtained using maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.10.19</version>
</dependency>
```

or gradle

```
repositories { jcenter() }
dependencies { testCompile "org.mockito:mockito-core:1.+" }
```

Version 2 is still in beta.

Versions

Version	Maven Central	Release notes	Release Date
2.1.0	mockito-core	changes	2016-10-04
1.10.19	mockito-core	changes	2014-12-31

Examples

Simple unit test using Mockito

The class we are going to test is:

```
public class Service {

    private Collaborator collaborator;

    public Service(Collaborator collaborator) {
        this.collaborator = collaborator;
```

```
    }

    public String performService(String input) {
        return collaborator.transformString(input);
    }
}
```

Its collaborator is:

```
public class Collaborator {

    public String transformString(String input) {
        return doStuff();
    }

    private String doStuff() {
        // This method may be full of bugs
        . . .
        return someString;
    }

}
```

In our test, we want to break the dependency from `Collaborator` and its bugs, so we are going to mock `Collaborator`:

```
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import org.junit.Test;

public class ServiceTest {
    @Test
    public void testPerformService() throws Exception {
        // Configure mock
        Collaborator collaboratorMock = mock(Collaborator.class);
        doReturn("output").when(collaboratorMock).transformString("input");

        // Perform the test
        Service service = new Service(collaboratorMock);
        String actual = service.performService("input");

        // Junit asserts
        String expected = "output";
        assertEquals(expected, actual);
    }
}
```

Using Mockito annotations

The class we are going to test is:

```
public class Service{

    private Collaborator collaborator;
```

```
    public Service(Collaborator collaborator){
        this.collaborator = collaborator;
    }


    public String performService(String input){
        return collaborator.transformString(input);
    }
}
```

Its collaborator is:

```
public class Collaborator {

    public String transformString(String input){
        return doStuff();
    }

    private String doStuff()
    {
        // This method may be full of bugs
        . . .
        return someString;
    }

}
```

In our test, we want to break the dependency from `Collaborator` and its bugs, so we are going to mock `Collaborator`. Using `@Mock` annotation is a convenient way to create different instances of mocks for each test:

```
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

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

@RunWith(MockitoJUnitRunner.class)
public class ServiceTest {

    @Mock
    private Collaborator collaboratorMock;

    @InjectMocks
    private Service service;

    @Test
    public void testPerformService() throws Exception {
        // Configure mock
        doReturn("output").when(collaboratorMock).transformString("input");

        // Perform the test
        String actual = service.performService("input");

        // Junit asserts
```

```
        String expected = "output";
        assertEquals(expected, actual);
    }


    @Test(expected=Exception.class)
    public void testPerformServiceShouldFail() throws Exception {
        // Configure mock
        doThrow(new Exception()).when(collaboratorMock).transformString("input");

        // Perform the test
        service.performService("input");
    }
}
```

Mockito will try to resolve dependency injection in the following order:

1. **Constructor-based injection** - mocks are injected into the constructor with most arguments (if some arguments can not be found, then nulls are passed). If an object was successfully created via constructor, then no other strategies will be applied.
2. **Setter-based injection** - mocks are injected by type. If there are several properties of the same type, then property names and mock names will be matched.
3. **Direct field injection** - same as for setter-based injection.

Note that no failure is reported in case if any of the aforementioned strategies failed.

Please consult the latest `@InjectMocks` for more detailed information on this mechanism in the latest version of Mockito.

Installation and setup

Installation

The preferred way to install Mockito is to declare a dependency on `mockito-core` with a build system of choice. As of July 22nd, 2016, the latest non-beta version is 1.10.19, but 2.x is already encouraged to be migrated to.

Maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
</dependency>
```

Gradle

```
repositories { jcenter() }
dependencies { testCompile "org.mockito:mockito-core:1.+" }
```

There is also `mockito-all` which contains Hamcrest and Objenesis besides Mockito itself. It is delivered through Maven mainly for ant users, but the distribution has been discontinued in Mockito 2.x.

Import

The most of the Mockito facilities are static methods of `org.mockito.Mockito`. Thus, Mockito can be statically imported into a class in this way:

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

Documentation entry point is located in the javadoc of this class.

Mock some methods on an object

Just some methods of an object can be mocked using `spy()` of mockito.

For example, imagine that method class requires some web service to work.

```
public class UserManager {

    List<User> users;

    public UserManager() {
        user = new LinkedLisk<User>();
    }

    public void addUser(User user) {
        if (isValid(user)) {
            user.add(user);
        } else {
            throw new NotValidUserException();
        }
    }

    protected boolean isValid(User user) {
        //some online web service to check if user is valid
    }

    public int numberOfUsers() {
        return users.size();
    }
}
```

`addUser` method has to be tested in order to make a useful Test for `UserManager`. However, a dependency is found here, `isValid` requires an external web service which is not contained in our code. Then, this external dependency should be neutralized.

In this case, if you only mock `isValid` you will be able to test the rest of the `UserManager` methods.

```
@Test
public void testAddUser() {
```

```
    User user = mock(User.class);
    UserManager  manager = spy(new UserManager());

    //it forces to manager.isValid to return true
    doReturn(true).when(manager).isValid(anyObject());

    manager.addUser(user);
    assertTrue(manager.numberOfUsers(), 1);
}
```

You can check easily the scenario where `user` is not valid.

```
@Test(expectedExceptions = NotValidUserException.class)
public void testNotValidAddUser() {
    User user = mock(User.class);
    UserManager  manager = spy(new UserManager());

    //it forces to manager.isValid to return false
    doReturn(false).when(manager).isValid(anyObject());

    manager.addUser(user);
}
```

Simple minimal Mockito Test

This example shows a minimal Mockito test using a mocked `ArrayList`:

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

import java.util.ArrayList;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class MockitoTest
{
    @Mock
    ArrayList<String> listMock;

    @Test
    public void testAppend() {
        // configure the mock to return "foobar" whenever "get()"
        // is called on "listMock" with an int value as parameter
        doReturn("foobar").when(listMock).get(anyInt());
        String result = listMock.get(0);

        assertEquals("foobar", result);
    }
}
```

Verifying arguments with ArgumentCaptor

To validate arguments to methods called on a mock, use the `ArgumentCaptor` class. This will allow you to extract the arguments into your test method and perform assertions on them.

This example tests a method which updates the name of a user with a given ID. The method loads the user, updates the `name` attribute with the given value and saves it afterwards. The test wants to verify that the argument passed to the `save` method is a `User` object with the correct ID and name.

```
// This is mocked in the test
interface UserDao {
    void save(User user);
}

@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    @Mock
    UserDao userDao;

    @Test
    public void testSetNameForUser() {
        UserService serviceUnderTest = new UserService(userDao);

        serviceUnderTest.setNameForUser(1L, "John");

        ArgumentCaptor<User> userArgumentCaptor = ArgumentCaptor.forClass(User.class);

        verify(userDao).save(userArgumentCaptor.capture());
        User savedUser = userArgumentCaptor.getValue();
        assertTrue(savedUser.getId() == 1);
        assertTrue(savedUser.getName().equals("John"));
    }
}
```

Verifying Arguments with ArgumentMatcher

Mockito provides a `Matcher<T>` interface along with an abstract `ArgumentMatcher<T>` class to verify arguments. It uses a different approach to the same use-case than the `ArgumentCaptor`. Additionally the ArgumentMatcher can be used in mocking too. Both use-cases make use of the `Mockito.argThat()` method that provides a reasonably readable test code.

```
verify(someMock).someMethod(Mockito.argThat(new ArgumentMatcher<String>() {

    @Override
    public boolean matches(Object o) {
        return o instanceof String && !((String)o).isEmpty();
    }

});
```

From the JavaDocs of ArgumentMatcher:

Warning: Be reasonable with using complicated argument matching, especially custom argument matchers, as it can make the test less readable. Sometimes it's better to implement equals() for arguments that are passed to mocks (Mockito naturally uses equals() for argument matching).

This can make the test cleaner.

Create objects mocked by Mockito

There are two ways to create object mocked by Mockito:

- via annotation
- via mock function

Via annotation:

With a JUnit test runner:

```
@RunWith(MockitoJUnitRunner.class)
public class FooTest {
    @Mock
    private Bar barMock;

    // ...
}
```

You can also use Mockito's JUnit `@Rule`, which provides the same functionality as the `MockitoJUnitRunner`, but doesn't need a `@RunWith` test runner:

```
public class FooTest {
    @Rule
    public MockitoRule mockito = MockitoJUnit.rule();

    @Mock
    private Bar barMock;

    // ...
}
```

If you can't use `@RunWith` or the `@Rule` annotation you can also init mocks "per hand":

```
public class FooTest {
    @Mock
    private Bar barMock;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    // ...
}
```

Via mock function:

```
public class FooTest {
    private Bar barMock = Mockito.mock(Bar.class);

    // ...
```

```
}
```

Because of type erasure, you cannot mock a generic class as above. You must mock the base class and explicitly cast to the right generic type:

```
public class FooTest {
    private Bar<String> genericBarMock = (Bar<String>) Mockito.mock(Bar.class);

    // ...
}
```

Add behaviour to mocked object

```
Mockito.when(mock.returnSomething()).thenReturn("my val");

mock.returnSomething(); // returns "my val"
mock.returnSomething(); // returns "my val" again
mock.returnSomething(); // returns "my val" again and again and again...
```

If you want different value on second call you can add wanted return argument to thenReturn method:

```
Mockito.when(mock.returnSomething()).thenReturn("my val", "other val");

mock.returnSomething(); // returns "my val"
mock.returnSomething(); // returns "other val"
mock.returnSomething(); // returns "other val" again
```

If you will call method without adding behaviour to mock it will return null:

```
barMock.mock.returnSomethingElse(); // returns null
```

In case that mocked method has parameters, you should declarate values too:

```
Mockito.when(mock.returnSomething("param 1")).thenReturn("my val 1");
Mockito.when(mock.returnSomething("param 2")).thenReturn("my val 2");

mock.returnSomething("param 1"); // returns "my val 1"
mock.returnSomething("param 2"); // returns "my val 2"
mock.returnSomething("param 3"); // returns null
```

If you don't care about param value you can use Matchers.any():

```
Mockito.when(mock.returnSomething(Matchers.any())).thenReturn("p1");

mock.returnSomething("param 1"); // returns "p1"
mock.returnSomething("param other"); // returns "p1"
```

To throw exception use thenThrow method:

```
Mockito.when(mock.returnSomething()).thenThrow(new Exception());
```

```
mock.returnSomething(); // throws Exception
```

Check arguments passed to mock

Lets assume we have this class and we would like to test `doSmth` method. In this case we want to see if parameter "val" is passed to `foo`. Object `foo` is mocked.

```
public class Bar {

    private final Foo foo;

    public Bar(final Foo foo) {
        this.foo = foo;
    }

    public void doSmth() {
        foo.bla("val");
    }
}
```

We can achieve this with `ArgumentCaptor`:

```
@Mock
private Foo fooMock;

@InjectMocks
private Bar underTest;

@Captor
private ArgumentCaptor<String> stringCaptor;

@Test
public void should_test_smth() {
    underTest.doSmth();

    Mockito.verify(fooMock).bla(stringCaptor.capture());

    assertThat(stringCaptor.getValue(), is("val"));
}
```

Verify method calls on mocked object

To check if a method was called on a mocked object you can use the `Mockito.verify` method:

```
Mockito.verify(someMock).bla();
```

In this example, we assert that the method `bla` was called on the `someMock` mock object.

You can also check if a method was called with certain parameters:

```
Mockito.verify(someMock).bla("param 1");
```

If you would like to check that a method was *not* called, you can pass an additional `VerificationMode` parameter to `verify`:

```
Mockito.verify(someMock, Mockito.times(0)).bla();
```

This also works if you would like to check that this method was called more than once (in this case we check that the method `bla` was called 23 times):

```
Mockito.verify(someMock, Mockito.times(23)).bla();
```

These are more examples for the `VerificationMode` parameter, providing more control over the number of times a method should be called:

```
Mockito.verify(someMock, Mockito.never()).bla(); // same as Mockito.times(0)

Mockito.verify(someMock, Mockito.atLeast(3)).bla(); // min 3 calls

Mockito.verify(someMock, Mockito.atLeastOnce()).bla(); // same as Mockito.atLeast(1)

Mockito.verify(someMock, Mockito.atMost(3)).bla(); // max 3 calls
```

Stubbing void methods

`void` methods can be stubbed using the doThrow(), doAnswer(), doNothing(), doCallRealMethod() family of methods.

```
Runnable mock = mock(Runnable.class);

doThrow(new UnsupportedOperationException()).when(mock).run();

mock.run(); // throws the UnsupportedOperationException
```

Note that`void` methods can't be stubbed using `when(..)` cause the compiler don't like `void` methods as argument.

Read Getting started with mockito online: https://riptutorial.com/mockito/topic/2055/getting-started-with-mockito

Chapter 2: Mock

Examples

Simple Mock

Mockito offers a one-size-fits-all mehtod to create mocks of (non-final) classes and interfaces.

```
Dependency mock = Mockito.mock(Dependency.class);
```

This creates a mock instance of `Dependency` regardless of whether `Dependency` is a interface or class.

It is then possible to stub method calls to that mock using the Mockito.when(x).thenReturn(y) notation.

```
Mockito.when(mock.possiblyBuggyMethod()).thenReturn("someString");
```

So that calls to `Dependency.possiblyBuggyMethod()` simply return `"someString"`.

There is another notation that is discouraged in most use cases as it is not typesafe.

```
Mockito.doReturn("someString").when(mock).possiblyBuggyMethod()
```

Mock with defaults

While a simple mock returns null (or defaults for primitives) to every call, it is possible to change that behaviour.

```
Dependency mock = Mockito.mock(Dependency.class, new Answer() {

        @Override
        public Object answer(InvocationOnMock invocationOnMock) throws Throwable {
            return "someString";
        }
    });
```

or using lambdas:

```
Dependency mock = Mockito.mock(Dependency.class, (Answer) invocationOnMock -> "someString");
```

This examples return "someString" to every invocation but it is possible to define any logic in the answer-method.

Mocking a class using annotations

Class under test:

```
public class GreetingsService { // class to be tested in isolation
    private UserService userService;

    public GreetingsService(UserService userService) {
        this.userService = userService;
    }

    public String getGreetings(int userId, LocalTime time) { // the method under test
        StringBuilder greetings = new StringBuilder();
        String timeOfDay = getTimeOfDay(time.getHour());
        greetings.append("Good ").append(timeOfDay).append(", ");
        greetings.append(userService.getFirstName(userId)) // this call will be mocked
                .append(" ")
                .append(userService.getLastName(userId)) // this call will be mocked
                .append("!");
        return greetings.toString();
    }

    private String getTimeOfDay(int hour) { // private method doesn't need to be unit tested
        if (hour >= 0 && hour < 12)
            return "Morning";
        else if (hour >= 12 && hour < 16)
            return "Afternoon";
        else if (hour >= 16 && hour < 21)
            return "Evening";
        else if (hour >= 21 && hour < 24)
            return "Night";
        else
            return null;
    }
}
```

Behavior of this interface will be mocked:

```
public interface UserService {
    String getFirstName(int userId);

    String getLastName(int userId);
}
```

Assume actual implementation of the `UserService`:

```
public class UserServiceImpl implements UserService {
    @Override
    public String getFirstName(int userId) {
        String firstName = "";
        // some logic to get user's first name goes here
        // this could be anything like a call to another service,
        // a database query, or a web service call
        return firstName;
    }

    @Override
    public String getLastName(int userId) {
        String lastName = "";
        // some logic to get user's last name goes here
        // this could be anything like a call to another service,
        // a database query, or a web service call
        return lastName;
```

```
    }
}
```

Junit test with Mockito:

```
public class GreetingsServiceTest {
    @Mock
    private UserServiceImpl userService; // this class will be mocked
    @InjectMocks
    private GreetingsService greetingsService = new GreetingsService(userService);

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testGetGreetings_morning() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(99)).thenReturn("John");
        when(userService.getLastName(99)).thenReturn("Doe");
        // invoke method under test
        String greetings = greetingsService.getGreetings(99, LocalTime.of(0, 45));
        Assert.assertEquals("Failed to get greetings!", "Good Morning, John Doe!", greetings);
    }

    @Test
    public void testGetGreetings_afternoon() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(11)).thenReturn("Jane");
        when(userService.getLastName(11)).thenReturn("Doe");
        // invoke method under test
        String greetings = greetingsService.getGreetings(11, LocalTime.of(13, 15));
        Assert.assertEquals("Failed to get greetings!", "Good Afternoon, Jane Doe!",
greetings);
    }
}
```

"Spy" for partial mocking

@Spy annotation (or method) can be used to partially mock an object. This is useful when you want to partially mock behavior of a class. E.g. Assume that you have a class that uses two different services and and you want to mock only one of them and use the actual implementation of the other service.

Side Note: Although philosophically I wouldn't consider this a "pure unit test" in a true sense, as you are integrating a real class and not testing your class under test in complete isolation. Nevertheless, this could be actually useful in the real world and I often use it when I mock the database using some in memory database implementation so that I can use real DAOs.

```
Class under test:
public class GreetingsService { // class to be tested in isolation
    private UserService userService;
    private AppService appService;
```

```
    public GreetingsService(UserService userService, AppService appService) {
        this.userService = userService;
        this.appService = appService;
    }

    public String getGreetings(int userId, LocalTime time) { // the method under test
        StringBuilder greetings = new StringBuilder();
        String timeOfDay = getTimeOfDay(time.getHour());
        greetings.append("Good ").append(timeOfDay).append(", ");
        greetings.append(userService.getFirstName(userId)) // this call will be mocked
                .append(" ")
                .append(userService.getLastName(userId)) // this call will be mocked
                .append("!");
        greetings.append(" Welcome to ")
                .append(appService.getAppName()) // actual method call will be made
                .append(".");
        return greetings.toString();
    }

    private String getTimeOfDay(int hour) { // private method doesn't need to be unit tested
        if (hour >= 0 && hour < 12)
            return "Morning";
        else if (hour >= 12 && hour < 16)
            return "Afternoon";
        else if (hour >= 16 && hour < 21)
            return "Evening";
        else if (hour >= 21 && hour < 24)
            return "Night";
        else
            return null;
    }
}
```

Behavior of this interface will be mocked:

```
public interface UserService {
    String getFirstName(int userId);

    String getLastName(int userId);
}
```

Assume actual implementation of the `UserService`:

```
public class UserServiceImpl implements UserService {
    @Override
    public String getFirstName(int userId) {
        String firstName = "";
        // some logic to get user's first name
        // this could be anything like a call to another service,
        // a database query, or a web service call
        return firstName;
    }

    @Override
    public String getLastName(int userId) {
        String lastName = "";
        // some logic to get user's last name
        // this could be anything like a call to another service,
        // a database query, or a web service call
```

```
        return lastName;
    }
}
```

Behavior of this interface won't be mocked:

```
public interface AppService {
    String getAppName();
}
```

Assume actual implementation of `AppService`:

```
public class AppServiceImpl implements AppService {
    @Override
    public String getAppName() {
        // assume you are reading this from properties file
        String appName = "The Amazing Application";
        return appName;
    }
}
```

Junit test with Mockito:

```
public class GreetingsServiceTest {
    @Mock
    private UserServiceImpl userService; // this class will be mocked
    @Spy
    private AppServiceImpl appService; // this class WON'T be mocked
    @InjectMocks
    private GreetingsService greetingsService = new GreetingsService(userService, appService);

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testGetGreetings_morning() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(99)).thenReturn("John");
        when(userService.getLastName(99)).thenReturn("Doe");
        // invoke method under test
        String greetings = greetingsService.getGreetings(99, LocalTime.of(0, 45));
        Assert.assertEquals("Failed to get greetings!", "Good Morning, John Doe! Welcome to
The Amazing Application.", greetings);
    }

    @Test
    public void testGetGreetings_afternoon() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(11)).thenReturn("Jane");
        when(userService.getLastName(11)).thenReturn("Doe");
        // invoke method under test
        String greetings = greetingsService.getGreetings(11, LocalTime.of(13, 15));
        Assert.assertEquals("Failed to get greetings!", "Good Afternoon, Jane Doe! Welcome to
The Amazing Application.", greetings);
    }
```

```
}
```

Set private fields in mocked objects

In your class that is under test, you may have some private fields that are not accessible even through constructor. In such cases you can use reflection to set such properties. This is a snippet from such JUnit test.

```
@InjectMocks
private GreetingsService greetingsService = new GreetingsService(); // mocking this class

@Before
public void setUp() {
    MockitoAnnotations.initMocks(this);
    String someName = "Some Name";
    ReflectionTestUtils.setField(greetingsService, // inject into this object
        "name", // assign to this field
        someName); // object to be injected
}
```

I'm using Sptring's `ReflectionTestUtils.setField(Object targetObject, String name, Object value)` method here to simplify, but you can use plain old Java Reflection to do the same.

Read Mock online: https://riptutorial.com/mockito/topic/4573/mock

Chapter 3: Mock final classes and methods

Introduction

Since Mockito 2.x we have the ability to mock final classes and methods.

Examples

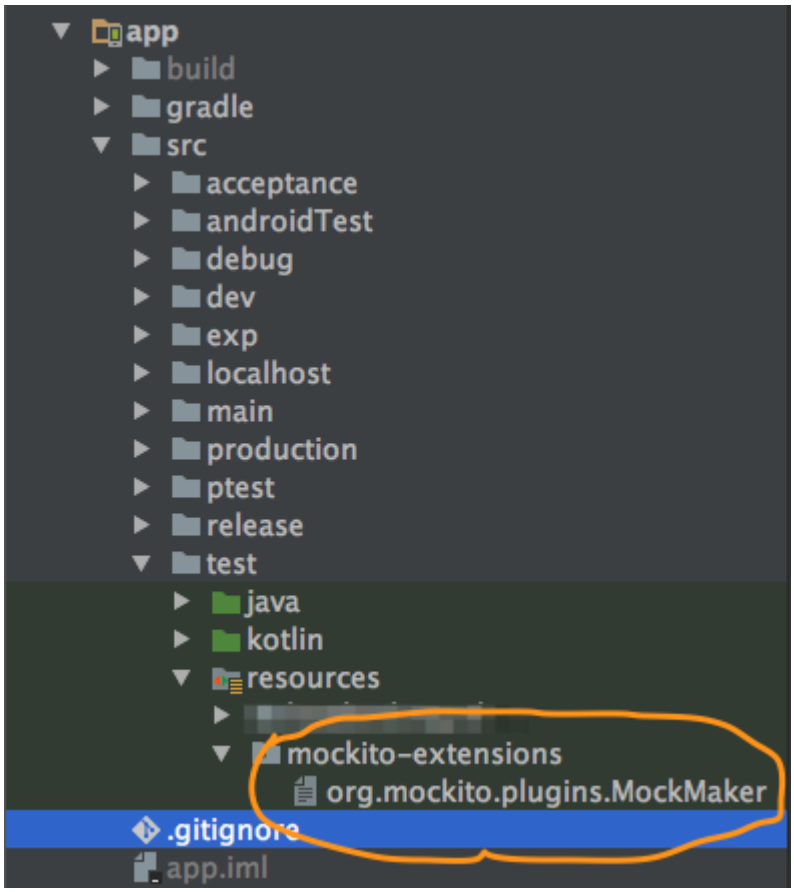
How to make it

Steps:

1. Add dependency to **Mockito** version 2.x in your `gradle` (at the time of writing this text the latest version is **2.7.22**):

```
testCompile "org.mockito:mockito-core:$versions.mockito"
```

2. Create a file in test resources with name `org.mockito.plugins.MockMaker`:



3. Add next line in this file:

```
mock-maker-inline
```

Read Mock final classes and methods online: https://riptutorial.com/mockito/topic/9769/mock-final-

classes-and-methods

Chapter 4: Mocking consecutive calls to a void return method

Introduction

The Mockito docs have an excellent example of how to provide a sequence of answers for multiple calls to a mock. However, they don't cover how to do that for a method that returns void, other than noting that stubbing void methods require using the do family of methods.

Remarks

Remember, for non-void methods, the `when(mock.method()).thenThrow().thenReturn()` version (see docs) is preferred because it is argument type-safe and more readable.

Examples

Faking a transient error

Imagine you're testing code that makes a call to this interface, and you want to make sure your retry code is working.

```
public interface DataStore {
    void save(Data data) throws IOException;
}
```

You could do something like this:

```
public void saveChanges_Retries_WhenDataStoreCallFails() {
    DataStore dataStore = new DataStore();
    Data data = new Data();
    doThrow(IOException.class).doNothing().when(dataStore).save(data);

    dataStore.save(data);

    verify(dataStore, times(2)).save(data);
    verifyDataWasSaved();
}
```

Read Mocking consecutive calls to a void return method online:
https://riptutorial.com/mockito/topic/9010/mocking-consecutive-calls-to-a-void-return-method

Chapter 5: Mockito Best Practices

Examples

BDDMockito style

Behavior Driven Development (BDD) testing style revolves around "given", "when" and "then" stages in tests. However, classical Mockito uses "when" word for "given" phase, and does not include other natural language constructions that can encompass BDD. Thus, BDDMockito aliases were introduced in version 1.8.0 in order to facilitate behavior driven tests.

The most common situation is to stub returns of a method. In the following example, `getStudent(String)` method of the mocked `StudentRepository` will return `new Student(givenName, givenScore)` if invoked with an argument that is equal to `givenName`.

```
import static org.mockito.BDDMockito.*;

public class ScoreServiceTest {

    private StudentRepository studentRepository = mock(StudentRepository.class);

    private ScoreService objectUnderTest = new ScoreService(studentRepository);

    @Test
    public void shouldCalculateAndReturnScore() throws Exception {
        //given
        String givenName = "Johnny";
        int givenScore = 10;
        given(studentRepository.getStudent(givenName))
            .willReturn(new Student(givenName, givenScore));

        //when
        String actualScore = objectUnderTest.calculateStudentScore(givenName);

        //then
        assertEquals(givenScore, actualScore);
    }
}
```

Sometimes it is desired to check if exception thrown from dependency is correctly handled or rethrown in a method under test. Such behavior can be stubbed in "given" phase in this way:

```
willThrow(new RuntimeException())).given(mock).getData();
```

Sometimes it is desired to set up some side effects that a stubbed method should introduce. Especially it can come in handy when:

- the stubbed method is a method that is supposed to change the internal state of a passed object
- the stubbed method is a void method

Such behavior can be stubbed in "given" phase with an "Answer":

```
willAnswer(invocation ->
this.prepareData(invocation.getArguments()[0])).given(mock).processData();
```

When it is desired to verify interactions with a mock, it can be done in "then" phase with `should()` or `should(VerificationMode)` (only since 1.10.5) methods:

```
then(mock).should().getData(); // verifies that getData() was called once
then(mock).should(times(2)).processData(); // verifies that processData() was called twice
```

When it is desired to verify that there were no more interactions with a mock besides already verified, it can be done in "then" phase with `shouldHaveNoMoreInteractions()` (since 2.0.0):

```
then(mock).shouldHaveNoMoreInteractions(); // analogue of verifyNoMoreInteractions(mock) in
classical Mockito
```

When it is desired to verify that there were absolutely no interactions with a mock, it can be done in "then" phase with `shouldHaveNoMoreInteractions()` (since 2.0.0):

```
then(mock).shouldHaveZeroInteractions(); // analogue of verifyZeroInteractions(mock) in
classical Mockito
```

When it is desired to check if methods were invoked in order it can be done in "then" phase with `should(InOrder)` (since 1.10.5) and `should(InOrder, VerificationMode)` (since 2.0.0):

```
InOrder inOrder = inOrder(mock);

// test body here

then(mock).should(inOrder).getData(); // the first invocation on the mock should be getData()
invocation
then(mock).should(inOrder, times(2)).processData(); // the second and third invocations on the
mock should be processData() invocations
```

Read Mockito Best Practices online: https://riptutorial.com/mockito/topic/4651/mockito-best-practices

Chapter 6: Verify method calls

Examples

Simple method call verification

One can verify whether a method was called on a mock by using `Mockito.verify()`.

```
Original mock = Mockito.mock(Original.class);
String param1 = "Expected param value";
int param2 = 100; // Expected param value

//Do something with mock

//Verify if mock was used properly
Mockito.verify(mock).method();
Mockito.verify(mock).methodWithParameters(param1, param2);
```

Verify order of calls

In some cases it may not suffice to know whether more that one methods were called. The calling order of methods is also important. In such case you may use `InOrder` class of `Mockito` to verify the order of methods.

```
SomeClass mock1 = Mockito.mock(SomeClass.class);
otherClass mock2 = Mockito.mock(OtherClass.class);

// Do something with mocks

InOrder order = Mockito.inOrder(mock1, mock2)
order.verify(mock2).firstMethod();
order.verify(mock1).otherMethod(withParam);
order.verify(mock2).secondMethod(withParam1, withParam2);
```

The `InOrder.verify()` works same as `Mockito.verify()` all other aspects.

Verify call arguments using ArgumentCaptor

`ArgumentCaptor` will to receive the actual invocation arguments that has been passed to method.

```
ArgumentCaptor<Foo> captor = ArgumentCaptor.forClass(Foo.class);
verify(mockObj).doSomethind(captor.capture());
Foo invocationArg = captor.getValue();
//do any assertions on invocationArg
```

For cases of multiple invocations of mocked method to receive all invocation arguments

```
List<Foo> invocationArgs = captor.getAllValues();
```

The same approach is used for capturing varargs.

Also there is possibility to create `ArgumentCaptor` using `@Captor` annotation:

```
@Captor
private ArgumentCaptor<Foo> captor;
```

Read Verify method calls online: https://riptutorial.com/mockito/topic/4858/verify-method-calls

Credits

S. No	Chapters	Contributors
1	Getting started with mockito	Abubakkar, Brice, cantido, Chriss, codebox, Community, Constantine, Eugen Martynov, fiskeben, gandreadis, J. Schneider, Kevin Welker, kiuby_88, Lorenzo Murrocu, Mark Rotteveel, Matsemann, nhouser9, Nicktar, Squidward, thug-gamer, Tim van der Lippe, Walery Strauch
2	Mock	Nicktar, RamenChef, Suraj Bajaj
3	Mock final classes and methods	Eugen Martynov
4	Mocking consecutive calls to a void return method	Cameron Stone
5	Mockito Best Practices	Constantine
6	Verify method calls	Abdullah, Sergii Bishyr