MOCKITO Framework:Tasty mocking framework for unit tests in Java

This tutorial explains testing with the Mockito framework for writing software tests.

1.INTRODUCTION:

Mockito is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications. Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

1.1.PREREQUISITES:

The following tutorial is based on an understanding of unit testing with the JUnit framework.

1.2.WHY?

Mockito is a mocking framework that tastes really good. It lets you write beautiful tests with a clean & simple API. Mockito doesn’t give you hangover because the tests are very readable and they produce clean verification errors.

Java mocking is dominated by expect-run-verify libraries like EasyMock or jMock. Mockito offers simpler and more intuitive approach: you ask questions about interactions *after* execution. Using mockito, you can verify what you want. Using expect-run-verify libraries you are often forced to look after irrelevant interactions.

No expect-run-verify also means that Mockito mocks are often *ready* without expensive setup upfront. They aim to be transparent and let the developer to focus on testing selected behavior rather than absorb attention.Mockito has very slim API, almost no time is needed to start mocking. There is only one kind of mock, there is only one way of creating mocks. Just remember that stubbing goes before execution, verifications of interactions go afterwards. You'll soon notice how natural is that kind of mocking when TDD-ing java code

1.3.FEATURES:

- Mocks concrete classes as well as interfaces
- Little annotation syntax sugar - @Mock
- Verification errors are clean - click on stack trace to see failed verification in test; click on exception's cause to navigate to actual interaction in code. Stack trace is always clean.
- Allows flexible verification in order (e.g: verify in order what you want, not every single interaction)
- Supports exact-number-of-times and at-least-once verification
- Flexible verification or stubbing using argument matchers (anyObject(), anyString() or refEq() for reflection-based equality matching).

2. MAVEN DEPENDENCIES:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter</artifactId>
   <version>1.3.1.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <version>1.3.1.RELEASE</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>2.8.9</version>
</dependency>
```

3.Testing with MOCK Objects:

3.1. Target and challenge of unit testing:

A unit test should test functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test, if possible.This can be done via using test replacements (*test doubles*) for the real dependencies. Test doubles can be classified like the following:

- A *dummy object* is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.
- *Fake* objects have working implementations, but are usually simplified. For example, they use an in memory database and not a real database.
- A *stub* class is an partial implementation for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually don’t respond to anything outside what’s programmed in for the test. Stubs may also record information about calls.
- A *mock object* is a dummy implementation for an interface or a class in which you define the output of certain method calls. Mock objects are configured to perform a certain behavior during a test. They typical record the interaction with the system and test can validate that.

Test doubles can be passed to other objects which are tested. Your tests can validate that the class reacts correctly during tests. For example, you can validate if certain methods on the mock object were called. This helps to ensure that you only test the class while running tests and that your tests are not affected by any side effects.

3.2. Mock object generation:

You can create mock objects manually (via code) or use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects at runtime and define their behavior.

The classical example for a mock object is a data provider. In production an implementation to connect to the real data source is used. But for testing a mock object simulates the data source and ensures that the test conditions are always the same.

These mock objects can be provided to the class which is tested. Therefore, the class to be tested should avoid any hard dependency on external data.

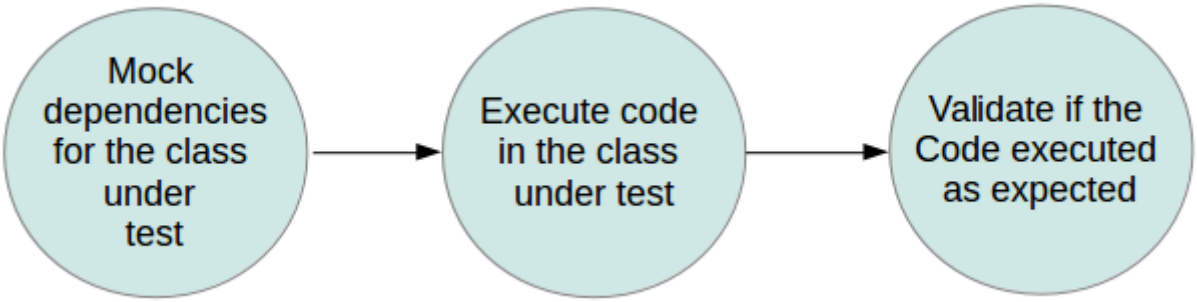
Mocking or mock frameworks allows testing the expected interaction with the mock object. You can, for example, validate that only certain methods have been called on the mock object.

3.3. Using Mockito for mocking objects:

Mockito is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

If you use Mockito in tests you typically:

- Mock away external dependencies and insert the mocks into the code under test
- Execute the code under test
- Validate that the code executed correctly



4. Using the Mockito API:

4.1. Creating mock objects with Mockito:

Mockito provides several methods to create mock objects:

- Using the static mock() method.
- Using the @Mock annotation.

If you use the @Mock annotation, you must trigger the creation of annotated objects. The MockitoRule allows this. It invokes the static method MockitoAnnotations.initMocks(this) to populate the annotated fields. Alternatively you can use @RunWith(MockitoJUnitRunner.class).

Example:

```
import static org.mockito.Mockito.*;
public class MockitoTest  {
    @Mock
1.  MyDatabase databaseMock;
2.  @Rule public MockitoRule mockitoRule = MockitoJUnit.rule();
    @Test
    public void testQuery()  {
3.    ClassToTest t  = new ClassToTest(databaseMock);
4.    boolean check = t.query("* from t");
5.    assertTrue(check);
6.    verify(databaseMock).query("* from t");
     }
  }
```

1. Tells Mockito to mock the databaseMock instance

2. Tells Mockito to create the mocks based on the @Mock annotation

3. Instantiates the class under test using the created mock

4. Executes some code of the class under test

5. Asserts that the method call returned true

6. Verify that the query method was called on the MyDatabase mock

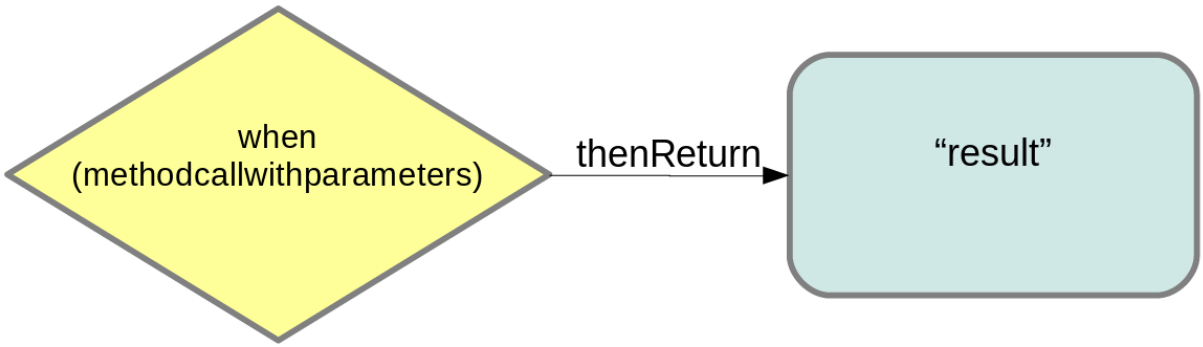
4.2. Configuring mocks:

Mockito allows to configure the return values of its mocks via a fluent API. Unspecified method calls return "empty" values:

- null for objects
- 0 for numbers
- false for boolean
- empty collections for collections.

4.2.1. "when thenReturn" and "when thenThrow:

Mocks can return different values depending on arguments passed into a method. The when(…).thenReturn(…) method chain is used to specify a a return value for a method call with pre-defined parameters.



You also can use methods like anyString or anyInt to define that dependent on the input type a certain value should be returned.

If you specify more than one value, they are returned in the order of specification, until the last one is used. Afterwards the last specified value is returned.

4.3. Verify the calls on the mock objects:

Mockito keeps track of all the method calls and their parameters to the mock object. You can use the verify() method on the mock object to verify that the specified conditions are met. For example, you can verify that a method has been called with certain parameters. This kind of testing is sometimes called *behavior testing*. Behavior testing does not check the result of a method call, but it checks that a method is called with the right parameters.

4.4. Using @InjectMocks for dependency injection via Mockito:

You also have the @InjectMocks annotation which tries to do constructor, method or field dependency injection based on the type.

Mockito can inject mocks either via constructor injection, setter injection, or property injection and in this order.

4.5. Capturing the arguments:

The ArgumentCaptor class allows to access the arguments of method calls during the verification. This allows to capture these arguments of method calls and to use them for tests.

SAMPLE CODE:

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

   @Captor
   private ArgumentCaptor<List<String>> captor;


   @Test
   public final void shouldContainCertainListItem() {
      List<String> asList = Arrays.asList("someElement_test", "someElement");
      final List<String> mockedList = mock(List.class);
      mockedList.addAll(asList);

      verify(mockedList).addAll(captor.capture());
      final List<String> capturedArgument = captor.getValue();
      assertThat(capturedArgument, hasItem("someElement"));
   }
}
```

4.6. Mocking final classes:

Since Mockito v2 it is possible to mock final classes. This feature is incubating and is deactivated by default. To activate the mocking of final classes create the file org.mockito.plugins.MockMaker in either src/test/resources/mockito-extensions/ or src/mockito-extensions/. Add this line to the file: *mock-maker-inline*. With this modification we now can mock a final class.

SAMPLE CODE:

```
final class FinalClass {
   public final String finalMethod() { return "something"; }
}

@Test
public final void mockFinalClassTest() {
    FinalClass instance = new FinalClass();

    FinalClass mock = mock(FinalClass.class);
    when(mock.finalMethod()).thenReturn("that other thing");

    assertNotEquals(mock.finalMethod(), instance.finalMethod());
}
```

4.7. Clean test code with the help of the strict stubs rule:

The strict stubs rule helps you to keep your test code clean and checks for common oversights. It adds the following:

- test fails early when a stubbed method gets called with different arguments than what it was configured for (with PotentialStubbingProblem exception).
- test fails when a stubbed method isn't called (with UnnecessaryStubbingException exception).
- org.mockito.Mockito.verifyNoMoreInteractions(Object) also verifies that all stubbed methods have been called during the test

SAMPLE CODE:

```
@Test
public void withoutStrictStubsTest() throws Exception {
   DeepThought deepThought = mock(DeepThought.class);

        when(deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything")).thenReturn(42);
        when(deepThought.otherMethod("some mundane thing")).thenReturn(null);

        System.out.println(deepThought.getAnswerFor("Six by nine"));

        assertEquals(42, deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything"));
        verify(deepThought, times(1)).getAnswerFor("Ultimate Question of Life, The Universe, and Everything");
}
```

5. JUNIT:

5.1. Junit Annotations:

1. **@Test:** The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded**.**

2. **@Before:** When writing tests, it is common to find that several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before the Test method. The @Before methods of super classes will be run before those of the current class.

3. **@After:** If you allocate external resources in a Before method you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method. All @After methods are guaranteed to run even if a Before or Test method throws an exception. The @After methods declared in superclasses will be run after those of the current class.

4. **@BeforeClass:** Sometimes several tests need to share computationally expensive setup (like logging into a database). While this can compromise the independence of tests, sometimes it is a necessary optimization. Annotating a public static void no-arg method with @BeforeClass causes it to be run once before any of the test methods in the class. The @BeforeClass methods of superclasses will be run before those the current class.The annotations @BeforeClass and @Before are same in functionality. The only difference is the method annotated with @BeforeClass will be called once per test class based, and the method annotated with @Before will be called once per test based.

5. **@AfterClass:** If you allocate expensive external resources in a BeforeClass method you need to release them after all the tests in the class have run. Annotating a public static void method with @AfterClass causes that method to be run after all the tests in the class have been run. All @AfterClass methods are guaranteed to run even if a BeforeClass method throws an exception. The @AfterClass methods declared in superclasses will be run after those of the current class.The annotations @AfterClass and @After are same in functionality. The only difference is the method annotated with @AfterClass will be called once per test class based, and the method annotated with @After will be called once per test based.

6. **@Ignore:** Sometimes you want to temporarily disable a test or a group of tests. Methods annotated with Test that are also annotated with @Ignore will not be executed as tests. Also, you can annotate a class containing test methods with @Ignore and none of the containing tests will be executed. Native JUnit 4 test runners should report the number of ignored tests along with the number of tests that ran and the number of tests that failed.

5.2. Junit Methods:

1. **assertArrayEquals():**The assertArrayEquals() method will test whether two arrays are equal to each other. In other words, if the two arrays contain the same number of elements, and if all the elements in the array are equal to each other. To check for element equality, the elements in the array are compared using their equals() method. More specifically, the elements of each array are compared one by one using their equals() method. That means, that it is not enough that the two arrays contain the same elements. They must also be present in the same order.

2. **AssertEquals():**The assertEquals() method compares two objects for equality, using their equals() method.

3. **assertTrue() + assertFalse():**The assertTrue() and assertFalse() methods tests a single variable to see if its value is either true, or false.

4. **assertNull()+assertNotNull():**The assertNull() and assertNotNull() methods test a single variable to see if it is null or not null.

5. **assertSame() and assertNotSame():**The assertSame() and assertNotSame() methods tests if two object references point to the same object or not. It is not enough that the two objects pointed to are equals according to their equals() methods. It must be exactly the same object pointed to.

6. **AssertThat():**The assertThat() method compares an object to an org.hamcrest.Matcher to see if the given object matches whatever the Matcher requires it to match.

6.A. MOCKING EXAMPLE – 1:(Mocking Database Operations)

6.A.1. Testing Controller:

```
package com.app.bankexample;
import static org.mockito.Matchers.anyLong;
import static org.mockito.Matchers.anyObject;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import java.util.ArrayList;
import java.util.List;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
import com.app.bankexample.controller.BankUserController;
import com.app.bankexample.model.BankUser;
import com.app.bankexample.repository.BankUserRepository;
import com.app.bankexample.service.BankService;


//@RunWith attaches a runner with the test class to initialize the test data
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class BankControllerTests {

		private MockMvc mockMvc;

		@Autowired
		private WebApplicationContext wac;

		private final String apiPrefix = "/Bank";

		@Before
		public void setup() {
			this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
		}

//@MockBean annotation is used to create the mock object to be injected and using Mockito methods directly in the test class
		@MockBean
		BankService service;
```

//@MockBean annotation is used to create the mock object to be injected and using Mockito methods directly in the test class

```
@MockBean
BankUserRepository repo;

@Autowired
BankUserController controller;

// creating JSON for Bank Object
private static String createBankInJson(String firstName, String lastName, Float
balance, String acType)
{
	return "{ \"firstName\": \"" + firstName + "\", " +
	 "\"lastName\":\"" + lastName + "\", " +
	  "\"balance\":\"" + balance + "\", " +
	  "\"acType\":\"" + acType + "\"}";
}

// creating JSON for Bank Object
private static String createBank2InJson(String firstName, String lastName, Float
balance, String acType, Long acNo)
{
	return "{ \"firstName\": \"" + firstName + "\", " +
	   "\"lastName\":\"" + lastName + "\", " +
	   "\"balance\":\"" + balance + "\", " +
	   "\"acType\":\"" + acType + "\", " +
	   "\"acNo\":\"" + acNo + "\"}";
}


@Test
public void verifySuccessfullCreate() throws Exception
{
BankUser bankUser1=newBankUser(
1l,"TestUserFirstName1","TestUserLastName1", 10000f,"TestAccountType");
when(service.getFindByFirstName(anyObject())).thenReturn(bankUser1);
mockMvc.perform(MockMvcRequestBuilders.post(apiPrefix + "/AddBankUser")
	.contentType(MediaType.APPLICATION_JSON)
	.content(createBankInJson("TestUserFirstName","TestUserLastName",
	10000f,"TestAccountType"))
	.accept(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
	.andDo(print());
}



@Test
public void verifyGetMyDetails() throws Exception
{
	BankUser bankUser=new BankUser(
	"TestUserFirstName","TestUserLastName", 10000f,"TestAccountType");
	when(repo.findOne(anyLong())).thenReturn(bankUser);
```

```
mockMvc.perform(MockMvcRequestBuilders.get(apiPrefix+"/GetMyDetails/{acNo}",anyLong()))
                .andExpect(status().isOk()).andDo(print());
        }


        @Test
        public void verifyShowAll() throws Exception
        {
                List<BankUser> list=new ArrayList<>();
                BankUser bankUser1=new BankUser(
                "TestUserFirstName1","TestUserLastName1", 10000f,"TestAccountType");
                BankUser bankUser2=new BankUser(
                "TestUserFirstName2","TestUserLastName2", 10000f,"TestAccountType");
                list.add(bankUser1);
                list.add(bankUser2);
                when(repo.findAll()).thenReturn(list);
        mockMvc.perform(MockMvcRequestBuilders.get(apiPrefix + "/AllBankUsers"))
                .andExpect(status().isOk()).andDo(print());
        }

        @Test
        public void verifyWithdraw() throws Exception
        {
                BankUser bankUser=new BankUser(
                "TestUserFirstName","TestUserLastName", 10000f,"TestAccountType");
                when(repo.findOne(anyLong())).thenReturn(bankUser);
                when(repo.save(bankUser)).thenReturn(bankUser);
                mockMvc.perform(MockMvcRequestBuilders.post(apiPrefix + "/Withdraw")
                .contentType(MediaType.APPLICATION_JSON)
                .content(createBank2InJson("TestUserFirstName","TestUserLastName",
                3000f,"TestAccountType", 3l))
                .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isAccepted())
                .andDo(print());
        }

        @Test
        public void verifyCredit() throws Exception
        {
                BankUser bankUser=new BankUser(
                "TestUserFirstName","TestUserLastName", 10000f,"TestAccountType");
                when(repo.findOne(anyLong())).thenReturn(bankUser);
                mockMvc.perform(MockMvcRequestBuilders.post(apiPrefix + "/Credit")
                .contentType(MediaType.APPLICATION_JSON)
                .content(createBank2InJson("TestUserFirstName","TestUserLastName",
                2000f,"TestAccountType",3l))
                .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isAccepted()).andDo(print());
        }

}
```

6.A.2. Testing Service class:

```
package com.app.bankexample;

import static org.junit.Assert.*;
import static org.mockito.Matchers.anyString;
import static org.mockito.Mockito.when;
import java.util.ArrayList;
import java.util.List;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.app.bankexample.model.BankUser;
import com.app.bankexample.repository.BankUserRepository;
import com.app.bankexample.service.BankServiceImpl;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class BankServiceTests
{

	@MockBean
	BankUserRepository repo;

	@Autowired
	BankServiceImpl service ;

	@Test
	public void verifyGetFindByFirstName()
	{
		BankUser bankuser=new BankUser(
		"TestUserFirstName1","TestUserLastName1", 10000f,"TestAccountType");

		BankUser bankuser2=new BankUser(
		"TestUserFirstName2","TestUserLastName2", 10000f,"TestAccountType");

		BankUser bankuser1=new BankUser(
		"TestUserFirstName3","TestUserLastName3", 10000f,"TestAccountType");

		List<BankUser> list=new ArrayList<>();
		list.add(bankuser1);
		list.add(bankuser2);
		when(repo.findByFirstName(anyString())).thenReturn(list);
		assertNotNull(service.getFindByFirstName(bankuser));
	}

}
```

6.B. MOCKING EXAMPLE – 2:(Mocking External REST Call)

6.B.1. Testing Controller:

```
package com.app.bankexample;

import static org.mockito.Matchers.anyString;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
import com.app.bankexample.controller.GeoController;
import com.app.bankexample.model.Location;
import com.app.bankexample.service.GeoService;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class TestGeocoderApplicationTests {
	private MockMvc mockMvc;
	@Autowired
	private WebApplicationContext wac;
	@Before
	public void setup() {
		this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
	}

	//@MockBean annotation is used to create the mock object to be injected
	@MockBean
	GeoService service
	@Autowired
	GeoController controller;
	@Test
	public void getLocationTest() throws Exception {
		Location location=new Location();
		when(service.getLocation(anyString())).thenReturn(location);
		mockMvc.perform(MockMvcRequestBuilders.get("/getLocation/TestLocation")
		.contentType(MediaType.APPLICATION_JSON)
	.accept(MediaType.APPLICATION_JSON)).andExpect(status().isOk()).andDo(print());
	}

}
```

7. LIMITATIONS, DONT'S AND REFERENCES:

7.1. Limitations:

Mockito has certain limitations. For example, you cannot mock static methods and private methods.

7.2.Dont's:

- Do not mock types you don't own.
- Don't mock value objects.
- Don't mock everything.

7.3. REFERENCES:

- http://site.mockito.org/
- http://www.baeldung.com/injecting-mocks-in-spring
- http://www.vogella.com/tutorials/Mockito/article.html
- https://www.tutorialspoint.com/mockito/index.htm
- http://tutorials.jenkov.com/java-unit-testing/asserts.html