Mockito

- Most common use cases for Mockito come when **we want to test an implementation class that depends on some service Interfaces**. We call the implementation class an *SUT* (System Under Testing) and the service interfaces *Dependencies*. Ideally what we would like to do is isolate the interface dependency's implementation from the actual interface method, and only use that within our code. For example:

```
public class MyImplementationClass {
	private IOtherService otherService;

	public void doSomething ()  {
		// some code here
		otherService.doSomeOtherStuff();
		// some other code here
	}
}
```

  In this example, Mockito will help us **ignore the otherService's object implementation**, because that's not what we want to test in this case.

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-all -->
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-all</artifactId>
   <version>1.10.19</version>
   <scope>test</scope>
</dependency>
```

Stubs

- A stub is nothing but a **sample implementation** of the interface method we want to "ignore". It is a class that gives a simple implementation. We have to create it in the test package.
- We should most likely give it a simple, dummy behaviour.
- We are retrieving a fake response using this method.
- Naming convention: *IOtherServiceStub* → Name of the interface + "Stub" at the end.
- **Problem**: Main problem with subbing is that whenever new methods are added to the Interface, we also have to add them as an implementation in the Stub class. So a lot of maintenance is required.

- **Problem:** Another problem using stubs is that if you want the stubbed method to behave differently, you would have to implement another particular class just for that.

Mocks

- Mocking is **creating objects that simulate the behaviour** of real objects. Unlike stubs, mocks can be created dynamically from code - at runtime.
- Mocks offer **more functionality** than stubbing, as you can verify method calls and a lot of other things.
- **Useful classes**:
    - *org.hamcrest.CoreMatchers.**
    - *org.hamcrest.Matchers.**
    - *org.mockito.Mock*
    - *org.mockito.InjectMocks*
- We can achieve a similar behaviour as stubbing by using the ***@Mock* annotation**. If we don't specify any behaviour for a method from a mocked class, it resolves to a default value: For example if the method has to retrieve an ArrayList, it will return an empty list. For int values, it will return 0 and for booleans it will return false.

  ```
  @Mock
  private IOtherService serviceMock;
  ```

- We can also **define specific behaviours for our mocks**. This is called dynamically stubbing the method:

  ```
  when(serviceMock.doSomeOtherStuff()).thenReturn(value1);
  ```

- We can also give different return values to the mocked behaviour by **concatenating multiple thenReturn** statements:

  ```
  when(serviceMock.doSomeOtherStuff()).thenReturn(value1).thenRe
  turn(value2);
  ```

- Mockito also supports **exceptions**:

  ```
  when(serviceMock.get(anyInt()).thenThrow(new
  NullPointerException("Something went wrong"));
  ```

  As in the Junit lesson, we will pass the *expected* parameter to the @Test annotation:

  ```
  @Test(expected=NullPointerException.class)
  ```

- With Mockito we can mock any kind of class method, including **predefined Java methods**:

  ```
  @Mock
  private List listMock;
  ...
  when(listMock.size()).thenReturn(2);
  ```

```
assertEquals(2, listMock.size()); // This test will pass
```

As in the Junit lesson, we will pass the *expected* parameter to the @Test annotation:

```
@Test(expected=NullPointerException.class)
```

- **Verify:** the mockito *verify* method allows us to check if a method has been called:

```
methodToTest(); // inside the methodToTest we would have a call to some list.get(0).
verify(listMock).get(0); // checks if the method .get() has been called with the parameter 0
```

  - **Never:** we can also check that a method is never called:

```
verify(listMock, never()).get(0); // checks if the method .get() has never been called with the parameter 0
```

  - **Times:** we can also check that a method has been called **x** times:

```
verify(listMock, times(2)).get(0); // checks if the method .get() has been called twice with the parameter 0
```

  - **AtLeast:** we can also check that a method has been called **at least x** times:

```
verify(listMock, atLeast(2)).get(0); // checks if the method .get() has been called at least twice with the parameter 0
```

- **Mockito matchers:** We can configure a mocked method in various ways:
  - One is by passing fixed values:

```
when(listMock.get(0)).thenReturn("Something");
```

  - Another one is by passing **argument matchers**:

```
when(listMock.get(anyInt())).thenReturn("Something");
```

  - If the method has more than one arg, **all its arguments** must be argumentMatchers. The example below **is not allowed**.

```
when(listMock.get(anyInt(),5)).thenReturn("Something");
```

  - We **can't use matchers as a return value**. Return values must be exact values.

- **Mockito argument capture:** If we want to capture the argument value with which a method is called within a mockito test
  - 1st we must **declare the argument captor**

```
ArgumentCaptor<Integer> intArgumentCaptor =
ArgumentCaptor.forClass(Integer.class);
```

- 2nd we must **define** the argument captor **on the specific method call** (only works with the BDD syntax)

```
then(listMock).should().get(intArgumentCaptor);
then(listMock).should(times(2)).get(intArgumentCaptor);
// this is if we want to capture arguments on multiple
method calls
```

- 3rd **capture** the argument (getValue / getAllValues)

```
assertThat(stringArgumentCaptor.getValue(), is(0));
assertThat(stringArgumentCaptor.getAllValues(), is(0));
```

Mockito annotations

- **@ExtendWith(MockitoExtension.class)** - Whenever we want to use annotations in a test class, first we have to specify it with this annotation at the class level.
- **@Mock** - Creates a mock of the particular class it annotates
  - @Mock annotation must be used with a class variable that represents an Interface.

```
@Mock

ISomeService someService; // Interface
```

- **@InjectMocks** - It declares the class where the @Mock interface will implement their methods

```
@InjectMock

SomeServiceImpl someServiceImpl; // Implementation class
```

- **@Captor** - It's the same as ArgumentCaptor but in an annotation form

```
@Captor

ArgumentCaptor<Integer> intArgumentCaptor;
```

- **@Spy** - Unlike @Mock, a mockito **@Spy** gets all the logic from a class. It is useful when we want to override a particular functionality but use the rest as it is. It is also called a partial mock. However **we should avoid using spies** in our code, since it's actually using part of the functionality and not really mocking it. In 99% of our testings we can use mocks instead of spies. A scenario where spying is necessary is when using legacy systems.

```
@Spy
```

```
List<String> spiedList = new ArrayList<String>();
…
stub(spiedList.size()).toReturn(5);

assertEquals(5, spiedList.size());
```

Hamcrest

- **Library** that will help us write more readable unit tests.

```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>2.2</version>
    <scope>test</scope>
</dependency>
```

- Usual use: ***assertThat(testObject, matcher)***
  - `assertThat(arrayListObject, hasSize(4));`
  - `assertThat(arrayListObject, hasItems(99,100));`
- It introduces a lot of **useful matchers**:
  - **Core**
    - anything - always matches, useful if you don't care what the object under test is
    - describedAs - decorator to adding custom failure description
    - is - decorator to improve readability
  - **Logical**
    - allOf - matches if all matchers match, short circuits (like Java &&)
    - anyOf - matches if any matchers match, short circuits (like Java ||)
    - not - matches if the wrapped matcher doesn't match and vice versa
  - **Object**
    - equalTo - test object equality using Object.equals
    - hasToString - test Object.toString
    - instanceOf, isCompatibleType - test type
    - notNullValue, nullValue - test for null
    - sameInstance - test object identity
  - **Beans**
    - hasProperty - test JavaBeans properties
  - **Collections**
    - array - test an array's elements against an array of matchers
    - hasEntry, hasKey, hasValue - test a map contains an entry, key or value

        - hasItem, hasItems - test a collection contains elements
        - hasItemInArray - test an array contains an element
    - **Number**
        - closeTo - test floating point values are close to a given value
        - greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo - test ordering
    - **Text**
        - equalToIgnoringCase - test string equality ignoring case
        - equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace
        - containsString, endsWith, startsWith - test string matching

BDD - Business Driven Development with Mockito

- Development methodology that starts with a **user story,** and based on that user story, we create particular **scenarios** in order to further define the **behaviour** of our application.
- Normal scenario structure:
    - **Title** of the scenario
    - **Given** some initial state
    - **When** some action happens
    - **Then** we expect a specific result
- **Mockito perfectly adapts** to this way of working, by implementing the tests that can prove the specific scenarios in the BDD definition:
    - Given → Where we setup ***initial configuration,*** along with the mockito ***when*** statements
        - **Note**: for better readability in BDD, mockito allows us to implement the *when* syntax differently:
        - *when(method()).then(result);* → ***given(method()).willReturn(result);***
    - When → Where we will call the method we want to test
    - Then → Where we add our ***assertion*** statements
        - A very straightforward way to check the *then* part is with the syntax ***then - should****,* specifically for method calls:

```
then(listMock).should().get(0)
then(listMock).should(never()).get(0)
```