## INTRODUCTION TO UNIT TESTING

A unit test is a test related to a single responsibility of a single class, often referred to as the System Under Test (SUT). The purpose of unit tests is to verify that the code in an SUT works. A tested object usually talks to other objects known as collaborators. These collaborators need to be created so the tested object can be assigned to them in the test. To make unit testing simpler and allow control of all aspects of the execution context, it is useful to replace the real cooperating objects with their fake replacements called test doubles. They look like the originals, but do not have any dependencies to other objects. Test doubles can also be easily programmed with specific expectations, such as recording any interactions they’ve had.

To make it clearer, try to imagine code for a typical enterprise system. Now here’s a service with some logic that needs two classes to fulfill its responsibility. Both classes then require a few other classes. One of these other classes could be a DAO, which needs access to a database, while yet another requires a message queue. It would be quite an effort to create that hierarchy and provide required resources. There could also be problems while running that kind of test, e.g., long startup times or the inability to test multiple developer stations simultaneously. Using Mocks, though, the same test could be much cleaner and faster.

Test doubles can be divided into a few groups:

- **Dummy** - an empty object passed in an invocation (usually only to satisfy a compiler when a method argument is required)
- **Fake** - an object having a functional implementation, but usually in a simplified form, just to satisfy the test (e.g., an in-memory database)
- **Stub** - an object with hardcoded behavior suitable for a given test (or a group of tests)
- **Mock** - an object with the ability to a) have a programmed expected behavior, and b) verify the interactions occurring in its lifetime (this object is usually created with the help of mocking framework)
- **Spy** - a mock created as a proxy to an existing real object; some methods can be stubbed, while the un-stubbed ones are forwarded to the covered object

Mockito is a mocking framework helpful in creating mocks and spies in a simple and intuitive way, while at the same time providing great control of the whole process.

## CONFIGURING MOCKITO IN A PROJECT

Mockito artifacts are available in the Maven Central Repository (MCR). The easiest way to make MCR available in your project is to put the following configuration in your dependency manager:

Maven:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>1.9.0</version>
  <scope>test</scope>
</dependency>
```

Gradle:

```
testCompile 'org.mockito:mockito-core:1.9.0'
```

Ivy:

```
<dependency org="org.mockito" name="mockito-core" rev="1.9.0" conf="test->default"/>
```

It will add JAR with Mockito classes as well as all required dependencies. Change 1.9.0 with the latest released version of Mockito.

This Refcard is based on the latest stable version 1.9.0. Some things are about to change in the further Mockito versions.

## CREATING MOCK

A mock can be created with the help of a static method mock():

```
Flower flowerMock = Mockito.mock(Flower.class);
```

But there’s another option: use of @Mock annotation:

```
@Mock
private Flower flowerMock;
```

**Warning:** If you want to use @Mock or any other Mockito annotations, it is required to call MockitoAnnotations.initMocks(testClass) OR MockitoJUnit4Runner as a JUnit runner (see the annotation section below for more information).

### Hot Tip

You can use Mockito to create mocks of a regular (not final) class not only of an interface.
MOCKITO calls real method when given method is called. This process of defining how a given mock method should behave is called stubbing.

**Warning:** Note that the examples in this Refcard were created to demonstrate behaviors of Mockito in a specific context. Of course, when writing the test for your codebase, there is no need to ensure that mocks are stubbed correctly.

Mockito provides a family of functions for requesting specific behaviors.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thenReturn(T value)</td>
<td>returns given value</td>
</tr>
<tr>
<td>thenThrow(Throwable toBeThrown)</td>
<td>throws given exception</td>
</tr>
<tr>
<td>then(Answer answer)</td>
<td>uses user created code to answer</td>
</tr>
<tr>
<td>thenCallRealMethod()</td>
<td>calls real method when working with partial mock/spy</td>
</tr>
</tbody>
</table>

Non-void methods return by default an "empty" value appropriate for its type (e.g.: null, 0, false, empty collection).

**Non void method's returned value**

One of the basic functions of mocking frameworks is an ability to return a given value when a specific method is called. It can be done using Mockito.when() in conjunction with thenReturn(). This process of defining how a given mock method should behave is called stubbing.

In this example, Mockito is stubbing the getNumberOfLeafs method of the Flower class to return the value 5.

```java
import static org.mockito.Mockito.when;

public class SimpleStubbingTest {
    private static final int TEST_NUMBER_OF_LEAFS = 5;

    public void shouldReturnGivenValue() {
        Flower flowerMock = mock(Flower.class);
        when(flowerMock.getNumberOfLeafs()).thenReturn(TEST_NUMBER_OF_LEAFS);
        int numberOfLeafs = flowerMock.getNumberOfLeafs();
        assertEquals(numberOfLeafs, TEST_NUMBER_OF_LEAFS);
    }
}
```

**ARGUMENT MATCHING**

Mockito, by default, compares arguments using equals() methods. Sometimes it's convenient to know exactly what parameter the method will be called with.

```java
import static org.mockito.Mockito.when;

public class SimpleStubbingTest {
    private static final int TEST_NUMBER_OF_LEAFS = 5;

    public void shouldReturnGivenValueUsingBDDSemantics() {
        Flower flowerMock = mock(Flower.class);
        //given
        given(flowerMock.getNumberOfLeafs()).willReturn(TEST_NUMBER_OF_LEAFS);
        //when
        int numberOfLeafs = flowerMock.getNumberOfLeafs();
        //then
        assertEquals(numberOfLeafs, TEST_NUMBER_OF_LEAFS);
    }
}
```

**Tip**

If an argument matcher is used for at least one argument, all arguments must be provided by matchers.

```java
import static org.mockito.Mockito.when;

public class SimpleStubbingTest {
    private static final int TEST_NUMBER_OF_LEAFS = 5;

    public void shouldReturnGivenValueUsingBDDSemantics() {
        Flower flowerMock = mock(Flower.class);
        //given
        given(flowerMock.getNumberOfLeafs()).willReturn(TEST_NUMBER_OF_LEAFS);
        //when
        int numberOfLeafs = flowerMock.getNumberOfLeafs();
        //then
        assertEquals(numberOfLeafs, TEST_NUMBER_OF_LEAFS);
    }
}
```

**Tip**

The methods from the any() family don't do any type checks. Various variants were created to avoid casting. To perform type checks, method isA(Class) should be used.

```java
import static org.mockito.Mockito.any;

public class SimpleStubbingTest {
    private static final int TEST_NUMBER_OF_LEAFS = 5;

    public void shouldReturnGivenValueUsingBDDSemantics() {
        Flower flowerMock = mock(Flower.class);
        //given
        given(flowerMock.getNumberOfLeafs()).willReturn(TEST_NUMBER_OF_LEAFS);
        //when
        int numberOfLeafs = flowerMock.getNumberOfLeafs();
        //then
        assertEquals(numberOfLeafs, TEST_NUMBER_OF_LEAFS);
    }
}
```

**Tip**

It is also possible to create a custom matcher by extending the ArgumentMatcher class and using it together with argThat(). For more information, see javadoc for Matchers class.
### Returns given value (not for void methods)

Any object that is equal to the given value.

### Any object that implements the same interface allowing the implementation of a callback method and providing a generic friendly way to avoid casting.

### Stubbing multiple calls to the same method

Sometimes you want to return different values for subsequent calls of the same method. Returned values can be mixed with exceptions. The last value/behavior used is for all following calls.

```java
given(schedulerMock.getNumberOfPlantsScheduledOnDate
given(flowerMock).getNumberOfLeafs();
//only runtime exception

//with the util method to create a matcher
private ArgumentMatcher<Date> haveHourFieldEqualTo(final int hour) {
    return new ArgumentMatcher<Date>() {
        @Override
        public boolean matches(Object argument) {
            return new ArgumentMatcher<Date>() {
                private ArgumentMatcher<Date> haveHourFieldEqualTo(final int hour) {
                    //with the util method to create a matcher
                    argThat(haveHourFieldEqualTo(7)).willReturn(1);
                }
            }
        }
    }
}
```

### Stubbing void methods

As we’ve seen before, the stubbed method is passed as a parameter to a given / when method. This obviously means that you cannot use this construct for void methods. Instead, you should use willXXX.given or doXXX.when. See here:

```java
given(waterSource.getWaterPressure()).willReturn(3, 5);
```

### do/willXXX methods family:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doThrow(Throwables toBeThrown) doThrow(Class extends Throwable) toBeThrown</td>
<td>Throws given exception</td>
</tr>
<tr>
<td>doAnswer(Answer answer)</td>
<td>Uses user-created code to answer</td>
</tr>
<tr>
<td>doCallRealMethod()</td>
<td>Working with spy</td>
</tr>
<tr>
<td>doNothing()</td>
<td>Does nothing</td>
</tr>
<tr>
<td>doReturn(Object toBeReturned)</td>
<td>Returns given value (not for void methods)</td>
</tr>
</tbody>
</table>

do/willXXX methods are also handy when working with spy objects, as will be seen in the following section.

A given / when construct allows Mockito to internally use the type returned by a stubbed method to provide a typed argument in the will/thenReturn methods. Void is not a valid type of it causes a compilation error. Will/doReturn does not use that trick. Will/doReturn can be also used for stubbing non-void methods, though it is not recommended because it cannot detect wrong return method types at a compilation time (only an exception at runtime will be thrown).

### Stubbing with a custom answer

In some rare cases it can be useful to implement a custom logic, later used on a stubbed method invocation. Mockito contains a generic Answer interface allowing the implementation of a callback method and providing access to invocation parameters (used arguments, a called method, and a mock instance).

### STUBBING WITH A CUSTOM ANSWER

It is not recommended to use do/willReturn for stubbing non-void methods.

```java
//compilation error - int expected, not boolean
waterSourceMock.getNumberOfPlantsScheduledOnDate().willReturn(true);

//only runtime exception
willReturn(true).given(browntock.getNumberOfLeafs());
```
**VERIFYING BEHAVIOR**

Once created, a mock remembers all operations performed on it. Important from the SUT perspective, these operations can easily be verified. In the basic form, use Mockito. verify (T mock) on the mocked method.

```java
WaterSource waterSourceMock = mock(WaterSource.class);
waterSourceMock.doSelfCheck();
verify(waterSourceMock).doSelfCheck();
```

By default, Mockito checks if a given method (with given arguments) was called once and only once. This can be modified using a VerificationMode. Mockito provides the number of very meaningful verification modes. It is also possible to create a custom verification mode.

### Verification Modes

- **atLeast(int minNumberOfInvocations)**: called at least n times
- **atMost(int maxNumberOfInvocations)**: called at most n times
- **atLeastOnce()**: called at least once
- **never()**: never called
- **times(int wantedNumberOfInvocations)**: called exactly n times
- **times(InvokedAtLeastOncePredicate invocationPredicate)**: called exactly n times satisfying the given predicate
- **times(InvokedAtLeastOncePredicate invocationPredicate, int maxNumberOfInvocations)**: called between n and at most m times satisfying the given predicate
- **times(InvokedAtLeastOncePredicate invocationPredicate, int minNumberOfInvocations, int maxNumberOfInvocations)**: called at least n times and at most m times satisfying the given predicate
- **always()**: always called

### Example

```java
verify(waterSourceMock).doSelfCheck();
```

**Warning**: The need to use a custom answer may indicate that tested code is too complicated and should be re-factored.

### Arguement matching

If you want to search or check for specific arguments, you can use ArgumentCaptor for this.

```java
ArgumentCaptor<SearchCriteria> captor = ArgumentCaptor.forClass(SearchCriteria.class);
```

### Timeouts

Mockito lets you verify interactions within a specified time frame. It causes a verify() method to wait for a specified period of time for a requested interaction rather than fail immediately if that had not already happened. It can be useful while testing multi-threaded systems.

---

**Mockito does not automatically verify calls**

Mockito enables you to verify if interactions with a mock were performed in a given order using the InOrder API. It is possible to create a group of mocks and verify the call order of all calls within that group.

```java
Mockito.inOrder(waterSource1, waterSource2).verify()
```

**Warning**: The need to use a custom answer may indicate that tested code is too complicated and should be re-factored.

### Verifying with Argument Matching

During a verification of an interaction, Mockito uses equals () methods on the passed arguments. This is usually enough. It is also possible to use the standard matchers, described earlier about stubbing, as well as custom matchers. However, in some situations it may be helpful to keep the actual argument value to make custom assertions on it. Mockito offers an ArgumentCaptor class, enabling us to retrieve the argument passed to a mock.

```java
ArgumentCaptor<SearchCriteria> captor = ArgumentCaptor.forClass(SearchCriteria.class);
assertThat(captor.getValue().getSearchCriteria().getSearchCriteria()).isNotNull();
```

**Warning**: It is recommended to use ArgumentCaptor with verification, but not with stubbing. Creating and using a captor in two different test blocks can decrease test readability. In addition to a situation when a stubbed method is not called, no argument is captured, which can be confusing.

### Verifying with Timeout

As an alternative to never (), which works only for the specified call, verifyNoInteraction ( Object ... mocks) method can be used to verify no interaction with any method of the given mock(s). Additionally, there is one more method available, called verifyNoMoreInteractions ( Object ... mocks), which allows to ensure that no more interaction (except the already verified ones) was performed with the mock(s). verifyNoMoreInteractions can be useful in some cases, but shouldn’t be overused by using on all mocks in every test. Unlike other mocking frameworks, Mockito does not automatically verify all stubbed calls. It is possible to do it manually, but usually it is just redundant. The tested code should mainly care about values returned by stubbed methods. If a stubbed method was not called, while being important from the test perspective, something else should break in a test. Mockito’s philosophy allows the test writer to focus on interesting behaviors in the test for the SUT and his collaborators.

---

**Mockito does not automatically verify calls**

Mockito ensures you to verify if interactions with a mock were performed in a given order using the InOrder API. It is possible to create a group of mocks and verify the call order of all calls within that group.

```java
Mockito.inOrder(waterSource1, waterSource2).verify()
```

**Warning**: The need to use a custom answer may indicate that tested code is too complicated and should be re-factored.

### Verifying with Argument Matching

During a verification of an interaction, Mockito uses equals () methods on the passed arguments. This is usually enough. It is also possible to use the standard matchers, described earlier about stubbing, as well as custom matchers. However, in some situations it may be helpful to keep the actual argument value to make custom assertions on it. Mockito offers an ArgumentCaptor class, enabling us to retrieve the argument passed to a mock.

```java
ArgumentCaptor<SearchCriteria> captor = ArgumentCaptor.forClass(SearchCriteria.class);
assertThat(captor.getValue().getSearchCriteria().getSearchCriteria()).isNotNull();
```

**Warning**: It is recommended to use ArgumentCaptor with verification, but not with stubbing. Creating and using a captor in two different test blocks can decrease test readability. In addition to a situation when a stubbed method is not called, no argument is captured, which can be confusing.

### Verifying with Timeout

As an alternative to never (), which works only for the specified call, verifyNoInteraction ( Object ... mocks) method can be used to verify no interaction with any method of the given mock(s). Additionally, there is one more method available, called verifyNoMoreInteractions ( Object ... mocks), which allows to ensure that no more interaction (except the already verified ones) was performed with the mock(s). verifyNoMoreInteractions can be useful in some cases, but shouldn’t be overused by using on all mocks in every test. Unlike other mocking frameworks, Mockito does not automatically verify all stubbed calls. It is possible to do it manually, but usually it is just redundant. The tested code should mainly care about values returned by stubbed methods. If a stubbed method was not called, while being important from the test perspective, something else should break in a test. Mockito’s philosophy allows the test writer to focus on interesting behaviors in the test for the SUT and his collaborators.
@Mock
Creates a mock of a given type

@Spy
Creates a spy of a given object

@Captor
Creates an argument captor of a given type

@InjectMocks
Creates an object of a given type and injects mocks and spies existing in a test

SPYING ON REAL OBJECTS

With Mockito, you can use real objects instead of mocks by replacing only some of their methods with the stubbed ones. Usually there is no reason to spy on real objects, and it can be a sign of a code smell, but in some situations (like working with legacy code and IoC containers) it allows us to test things impossible to test with pure mocks.

```
@Test
public void shouldStubMethodAndCallRealNotStubbedMethod() {
    Flower realFlower = new Flower();
    realFlower.setNumberOfLeafs(ORIGINAL_NUMBER_OF_LEAFS);
    Flower flowerSpy = spy(realFlower);
    willDoNothing().given(flowerSpy).setNumberOfLeafs(anyInt());
    flowerSpy.setNumberOfLeafs(NEW_NUMBER_OF_LEAFS); //stubbed - should do nothing
    verify(flowerSpy).setNumberOfLeafs(NEW_NUMBER_OF_LEAFS);
    assertEquals(flowerSpy.getNumberOfLeafs(), ORIGINAL_NUMBER_OF_LEAFS); //value was not changed
}
```

CHANGING THE MOCK DEFAULT RETURN VALUE

Mockito enables us to redefine a default value returned from non-stubbed methods.

```
Default Answer          Description
----------------------------------------------------------------
RETURNS_DEFAULTS       Returns a default "empty" value (e.g., null, 0, false, empty collection) - used by default
RETURNS_SMART_NULLS    Creates a spy of a given object
RETURNS_MOCKS          Returns a default "empty" value, but a mock instead of null
RETURNS_DEEP_STUBS     Allows for a simple deep stubbing (e.g., Given(ourMock.getObject().getValue()).willReturn(s))
CALLS_REAL_METHODS     Call a real method of spied object
```

Warning: The last three default answers should not be needed when working with well-crafted, testable code. The behavior can be configured per mock during its creation or globally for all tests using GlobalConfiguration mechanism (it helps to use RETURNS_SMART_NULLS by default).

```
PlantWaterer plantWatererMock =
    mock(PlantWaterer.class, Mockito.RETURNS_DEEP_STUBS);
given(plantWatererMock.getWaterSource().getWaterPressure()).willReturn(5);
@Mock(answer = Answers.RETURNS_SMART_NULLS)
private PlantWaterer plantWatererMock;
```

Sample verbose exception received using SmartNull:

```
org.mockito.exceptions.verification.SmartNullPointerException:
You have a NullPointerException here:
-> at PlantWaterer.generateNPE(PlantWaterer.java:24)
because this method call was "not" stubbed correctly:
-> at PlantWaterer.generateNPE(PlantWaterer.java:24)
wateringScheduler.returnNull();
at PlantWaterer.generateNPE(PlantWaterer.java:24)
at DefaultValuesTest.shouldReturnNicerErrorMessageOnNPE(DefaultValuesTest.java:64)
```
MOCKITO

**Resetting Mock**

In some rare cases (like using a mock as a bean in an IoC container) you may need to reset a mock using the Mockito `reset(T ... mocks)` static method. This causes the mock to forget all previous behavior and interactions.

**Warning:** In most cases, using the reset method in a test is a code smell and should be avoided. Splitting a large test into smaller ones with mocks can be a better alternative.

**Limitations**

Mockito has a few limitations worth remembering. They are generally technical restrictions, but Mockito authors believe using hacks to work around them would encourage people to write poorly testable code. Mockito cannot:

- mock final classes
- mock enums
- mock final methods
- mock static methods
- mock private methods
- mock `hashCode()` and `equals()`
- mock `toString()` method

**Important Note:**

In most cases, using the reset method in a test is a code smell and should be avoided. Splitting a large test into smaller ones with mocks can be a better alternative.

**Recommended Book**

Practical Unit Testing with Test NG and Mockito

This book explains in detail how to implement unit tests using two very popular open source Java technologies: TestNG and Mockito. It presents a range of techniques necessary to write high quality unit tests - e.g. mocks, parameterized tests and matchers. It also discusses trade-offs related to the choices we have to make when dealing with some real-life code issues.

http://practicalunittesting.com/

**Further Reading**

- [http://blog.solidsoft.info/mockito-docs/](http://blog.solidsoft.info/mockito-docs/) - official Mockito documentation (redirect to the current version)