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

1. Prerequisites

This tutorial introduces the usage of Mockito for JUnit tests (https://www.vogella.com/tutorials/JUnit/article.html). If you are not familiar with JUnit, please read first the JUnit Tutorial (https://www.vogella.com/tutorials/JUnit/article.html).

2. Using mocks for unit testing

2.1. Mock objects

In your unit tests, you want to test certain functionality (the class under test) in isolation. Other functionality required to test the class under test, should be controlled to avoid side-effects.

A *mock object* is a dummy implementation for an interface or a class. It allows to define the output of certain method calls. They typically record the interaction with the system and tests can validate that.

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.

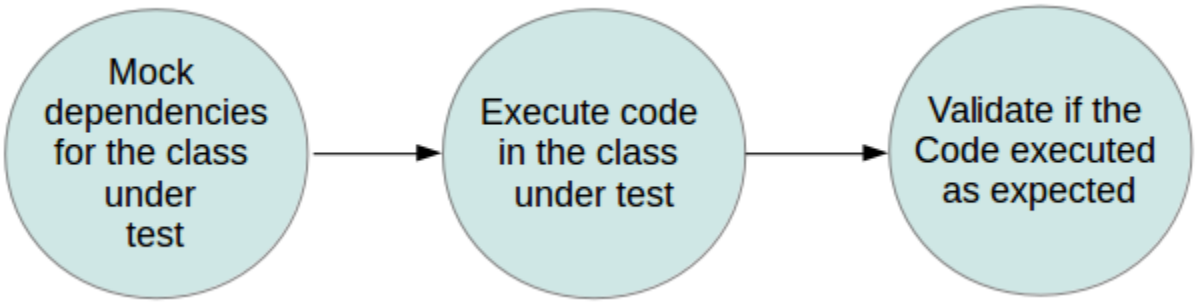
Mocking or mock frameworks allows testing the expected interaction with the mock object. You can, for example, validate that only certain methods have been called on the mock object.

2.2. 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 greatly simplifies the development of tests for classes with external dependencies.

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



3. Adding Mockito to a project

Using the Mockito libraries should be done with a modern dependency system like Maven or Gradle. All modern IDEs (Eclipse, Visual Studio Code, IntelliJ) support both Maven and Gradle.

The following contains detailed descriptions for your environment, pick the one which is relevant for you. The latest version of Mockito can be found via https://search.maven.org/artifact/org.mockito/mockito-core.

▼ Using Maven

To use Mockito in a Maven project you have to add it as dependency to your pom file.

```xml
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-inline</artifactId>
  <version>3.7.7</version>
</dependency>
```

mockito-inline is the Java 16 ready version of mockito-core. It also support mocking of static methods and final classes.

To use the `MockitoExtension` for JUnit 5 also add the following dependencies.

```xml
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.7.7</version>
    <scope>test</scope>
</dependency>
```

- Using Gradle for a Java project
- Using Gradle for an Android project
- OSGi or Eclipse plug-in development

4. Using the Mockito API

4.1. Creating mock objects

Mockito provides several methods to create mock objects:

- Using the static `mock()` method.
- Using the `@Mock` annotation.
- Using the `@ExtendWith(MockitoExtension.class)` extension for JUnit 5

If you use the `@Mock` annotation, you must trigger the initialization of the annotated fields. The `MockitoExtension` does this by calling the static method `MockitoAnnotations.initMocks(this)`.

The usage of the `@Mock` annotation and the `MockitoExtension` extension for JUnit 5 is demonstrated by the following example.

The `Service` class is the one which should be tested and the `Database` class should be mocked.

JAVA

```java
package com.vogella.junit5;

public class Database {

    public boolean isAvailable() {
        // TODO implement the access to the database
        return false;
    }
    public int getUniqueId() {
        return 42;
    }
}
```

JAVA

```java
package com.vogella.junit5;

public class Service {

    private Database database;

    public Service(Database database) {
        this.database = database;
    }

    public boolean query(String query) {
        return database.isAvailable();
    }


    @Override
    public String toString() {
        return "Using database with id: " + String.valueOf(database.getUniqueId());
    }
}
```

```java
package com.vogella.junit5;

import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)                          (1)
class ServiceTest {

    @Mock
    Database databaseMock;                                   (2)

    @Test
    public void testQuery()  {
        assertNotNull(databaseMock);
        when(databaseMock.isAvailable()).thenReturn(true);   (3)
        Service t  = new Service(databaseMock);              (4)
        boolean check = t.query("* from t");                 (5)
        assertTrue(check);
    }
}
```

1. Tells Mockito to create the mocks based on the @Mock annotation, this requires JUnit 5, if you use JUnit 4, call `Mock.init()` in your setup method
2. Tells Mockito to mock the databaseMock instance
3. Configure the Mock to return true when its isAvailable method is called, see later for more options
4. Executes some code of the class under test
5. Asserts that the method call returned true

Static imports

By adding the `org.mockito.Mockito.*;` static import, you can use methods like `mock()` directly in your tests. Static imports allow you to call static members, i.e., methods and fields of a class directly without specifying the class.

Using static imports also greatly improves the readability of your test code.

4.2. Configuring mocks

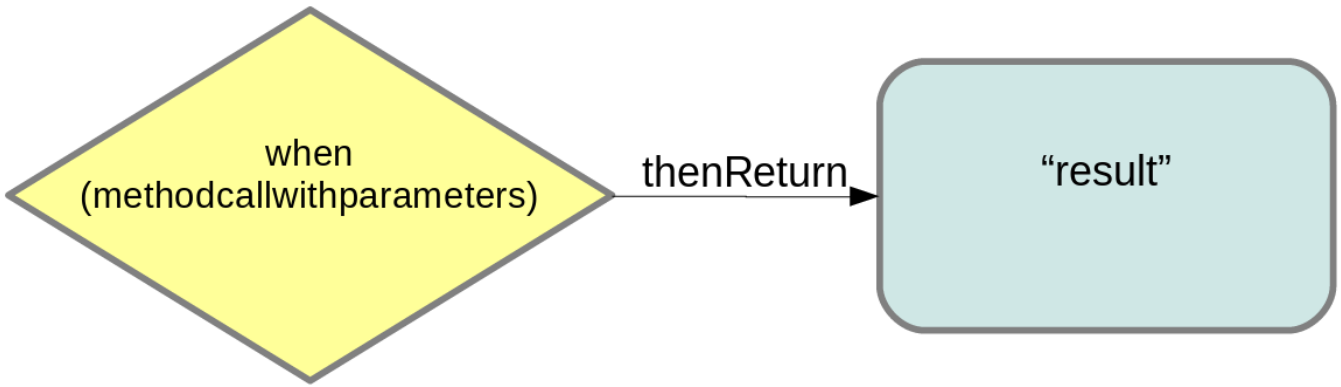
Mockito allows to configure the return values of its mocks via a fluent API. Unspecified method calls return "empty" values:

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

The following assert statements are only for demonstration purposes, a real test would use the mocks to unit test another functionality.

4.2.1. Using `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 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.

The following demonstrates the usage of `when(….).thenReturn(….)`.

```java
package com.vogella.junit5;

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

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class ServiceDatabaseIdTest {

    @Mock
    Database databaseMock;

    @Test
    public void ensureMockitoReturnsTheConfiguredValue() {

        // define return value for method getUniqueId()
        when(databaseMock.getUniqueId()).thenReturn(42);

        Service service = new Service(databaseMock);
        // use mock in test....
        assertEquals(service.toString(), "Using database with id: 42");
    }

}
```

Other examples which demonstrates the configuration of Mockito are in the following listing. These are not real test, the test only validating the Mockito configuration.

```java
package com.vogella.junit5;

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

import java.util.Iterator;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;


@ExtendWith(MockitoExtension.class)
class MockitoWhenExampleTest {

    @Mock
    Iterator<String> i;

    Comparable<String> c;

    // demonstrates the return of multiple values
    @Test
    public void testMoreThanOneReturnValue() {
        when(i.next()).thenReturn("Mockito").thenReturn("rocks");
        String result = i.next() + " " + i.next();
        // assert
        assertEquals("Mockito rocks", result);
    }

    // this test demonstrates how to return values based on the input
    // and that @Mock can also be used for a method parameter
    @Test
    public void testReturnValueDependentOnMethodParameter(@Mock Comparable<String> c)  {
            when(c.compareTo("Mockito")).thenReturn(1);
            when(c.compareTo("Eclipse")).thenReturn(2);
            //assert
            assertEquals(1, c.compareTo("Mockito"));
            assertEquals(2, c.compareTo("Eclipse"));
    }

    // return a value based on the type of the provide parameter

    @Test
    public void testReturnValueInDependentOnMethodParameter2(@Mock Comparable<Integer> c )  {
            when(c.compareTo(isA(Integer.class))).thenReturn(0);
```

```
            //assert
            assertEquals(0, c.compareTo(Integer.valueOf(4)));
    }

}
```

The `when(….).thenReturn(….)` method chain can be used to throw an exception.

JAVA

```
package com.vogella.junit5;

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

import java.util.Properties;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;


class MockitoThrowsTest {

    // demonstrates the configuration of a throws with Mockito
    // not a read test, just "testing" Mockito behavior
    @Test
    public void testMockitoThrows() {
        Properties properties = Mockito.mock(Properties.class);

        when(properties.get(Mockito.anyString())).thenThrow(new IllegalArgumentException("Stuff"));

        Throwable exception = assertThrows(IllegalArgumentException.class, () -> properties.get("A"));

        assertEquals("Stuff", exception.getMessage());
    }
}
```

4.2.2. "doReturn when" and "doThrow when"

The `doReturn(…).when(…).methodCall` call chain works similar to `when(….).thenReturn(….)`. It is useful for mocking methods which give an exception during a call, e.g., if you use use functionality like Wrapping Java objects with Spy.

```java
package com.vogella.junit5;

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

import java.util.Properties;

import org.junit.jupiter.api.Test;


class MockitoSpyTest {

    // demonstrates of the spy function
    @Test
    public void testMockitoThrows() {
        Properties properties = new Properties();

        Properties spyProperties = spy(properties);

        doReturn("42").when(spyProperties).get("shoeSize");

        String value = (String) spyProperties.get("shoeSize");

        assertEquals("42", value);
    }
}
```

The `doThrow` variant can be used for methods which return `void` to throw an exception. This usage is demonstrated by the following code snippet.

```java
package com.vogella.junit5;

import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.mockito.Mockito.doThrow;
import static org.mockito.Mockito.mock;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;

import org.junit.jupiter.api.Test;


class MockitoThrowsTest {

    @Test
    public void testForIOException() throws IOException {
        // create an configure mock
        OutputStream mockStream = mock(OutputStream.class);
        doThrow(new IOException()).when(mockStream).close();

        // use mock
        OutputStreamWriter streamWriter = new OutputStreamWriter(mockStream);

        assertThrows(IOException.class, () -> streamWriter.close());
    }


}
```

4.3. Wrapping Java objects with Spy

@Spy or the `spy()` method can be used to wrap a real object. Every call, unless specified otherwise, is delegated to the object.

```java
package com.vogella.junit5;

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

import java.util.LinkedList;
import java.util.List;

import org.junit.jupiter.api.Test;

public class MockitoSpyTest {

    @Test
    public void testLinkedListSpyCorrect() {
        // Lets mock a LinkedList
        List<String> list = new LinkedList<>();
        List<String> spy = spy(list);

        // this would not work as delegate it called so spy.get(0)
        // throws IndexOutOfBoundsException (list is still empty)
        // when(spy.get(0)).thenReturn("foo");

        // you have to use doReturn() for stubbing
        doReturn("foo").when(spy).get(0);

        assertEquals("foo", spy.get(0));
    }

}
```

4.4. 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.

```java
package com.vogella.junit5;

import static org.mockito.Mockito.atLeast;
import static org.mockito.Mockito.atLeastOnce;
import static org.mockito.Mockito.never;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentMatchers;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class MockitoVerifyTest {


    @Test
    public void testVerify(@Mock Database database)  {
        // create and configure mock
        when(database.getUniqueId()).thenReturn(43);


        // call method testing on the mock with parameter 12
        database.setUniqueId(12);
        database.getUniqueId();
        database.getUniqueId();


        // now check if method testing was called with the parameter 12
        verify(database).setUniqueId(ArgumentMatchers.eq(12));

        // was the method called twice?
        verify(database, times(2)).getUniqueId();

        // other alternatives for verifiying the number of method calls for a method
        verify(database, never()).isAvailable();
        verify(database, never()).setUniqueId(13);
        verify(database, atLeastOnce()).setUniqueId(12);
        verify(database, atLeast(2)).getUniqueId();

        // more options are
        // times(numberOfTimes)
        // atMost(numberOfTimes)
```

```
        // This let's you check that no other methods where called on this object.
        // You call it after you have verified the expected method calls.
        verifyNoMoreInteractions(database);
    }

}
```

In case you do not care about the value, use the `anyX`, e.g., `anyInt`, `anyString()`, or `any(YourClass.class)` methods.

4.5. 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. For example, assume that you have the following classes.

JAVA

```
package com.vogella.junit5;

public class User {

}
```

JAVA

```
package com.vogella.junit5;

public class ArticleListener {

}
```

JAVA

```
package com.vogella.junit5;

public class ArticleDatabase {

    public void addListener(ArticleListener articleListener) {
        // TODO Auto-generated method stub

    }

}
```

```java
package com.vogella.junit5;
public class ArticleManager {
    private User user;
    private ArticleDatabase database;

    public ArticleManager(User user, ArticleDatabase database) {
        super();
        this.user = user;
        this.database = database;
    }

    public void initialize() {
        database.addListener(new ArticleListener());
    }
}
```

This class can be constructed via Mockito and its dependencies can be fulfilled with mock objects as demonstrated by the following code snippet.

```java
package com.vogella.junit5;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class ArticleManagerTest {

    @Mock ArticleDatabase database;
    @Mock User user;

    @InjectMocks
    private ArticleManager manager; (1)

    @Test public void shouldDoSomething() {
       // calls addListener with an instance of ArticleListener
       manager.initialize();

        // validate that addListener was called
       Mockito.verify(database).addListener(Mockito.any(ArticleListener.class));
    }
}
```

(1) creates an instance of `ArticleManager` and injects the mocks into it

Mockito can inject mocks either via constructor injection, setter injection, or property injection and in this order. So if `ArticleManager` would have a constructor that would only take `User` and setters for both fields, only the mock for `User` would be injected.

4.6. 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.

To run this example you need to add hamcrest-library (https://mvnrepository.com/artifact/org.hamcrest/hamcrest-library) to your project.

```java
package com.vogella.junit5;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasItem;
import static org.mockito.Mockito.verify;

import java.util.Arrays;
import java.util.List;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class MockitoArgumentCaptureTest {

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

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

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

4.7. Using Answers for complex mocks

`thenReturn` returns a predefined value every time. With an `Answer` object you can calculate a response based on the arguments given to your stubbed method.

This can be useful if your stubbed method is supposed to call a function on one of the arguments or if your method is supposed to return the first argument to allow method chaining. There exists a static method for the latter. Also note that there a different ways to configure an answer:

```java
import static org.mockito.AdditionalAnswers.returnsFirstArg;

@Test
public final void answerTest() {
    // with doAnswer():
    doAnswer(returnsFirstArg()).when(list).add(anyString());
    // with thenAnswer():
    when(list.add(anyString())).thenAnswer(returnsFirstArg());
    // with then() alias:
    when(list.add(anyString())).then(returnsFirstArg());
}
```

Or if you need to do a callback on your argument:

```java
@Test
public final void callbackTest() {
    ApiService service = mock(ApiService.class);
    when(service.login(any(Callback.class))).thenAnswer(i -> {
        Callback callback = i.getArgument(0);
        callback.notify("Success");
        return null;
    });
}
```

It is even possible to mock a persistence service like an DAO, but you should consider creating a fake class instead of a mock if your Answers become too complex.

```java
List<User> userMap = new ArrayList<>();
UserDao dao = mock(UserDao.class);
when(dao.save(any(User.class))).thenAnswer(i -> {
    User user = i.getArgument(0);
    userMap.add(user.getId(), user);
    return null;
});
when(dao.find(any(Integer.class))).thenAnswer(i -> {
    int id = i.getArgument(0);
    return userMap.get(id);
});
```

5. More on Mockito

5.1. Mocking final classes and static methods

Since the `mockito-inline` library replaced the `mockito-core` library it is possible to mock final classes and static methods.

For example, if you have the following final class:

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

You can mock it via the following code:

```java
package com.vogella.junit5;

import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.MockedStatic;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class MockitoMockFinal {


    @Test
    public void testMockFinal(@Mock FinalClass finalMocked)  {
        assertNotNull(finalMocked);
    }

    @Test
    public void testMockFinalViaMockStatic()  {
        MockedStatic<FinalClass> mockStatic = Mockito.mockStatic(FinalClass.class);
        assertNotNull(mockStatic);
    }
}
```

Mockito also allows to mock static methods.

```java
package com.vogella.junit5;

public class Utility {

    public static String getDatabaseConnection(String url) {
        return "http:///production/" + url;
    }
}
```

```java
package com.vogella.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;

class MyUtilsTest {

    @Test
    void shouldMockStaticMethod() {
        MockedStatic<FinalClass> mockStatic = Mockito.mockStatic(FinalClass.class);
        try (MockedStatic<Utility> mockedStatic = Mockito.mockStatic(Utility.class)) {

            mockedStatic.when(() ->
Utility.getDatabaseConnection(Mockito.eq("test"))).thenReturn("testing");
            mockedStatic.when(() ->
Utility.getDatabaseConnection(Mockito.eq("prod"))).thenReturn("production");

            String result1 = Utility.getDatabaseConnection("test");

            assertEquals("testing", result1);
            String result2 = Utility.getDatabaseConnection("prod");
            assertEquals("production", result2);

        }

    }
}
```

5.2. 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

```java
@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");
}
```

```java
// activate the strict subs rule
@Rule public MockitoRule rule = MockitoJUnit.rule().strictness(Strictness.STRICT_STUBS);

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

    when(deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything")).thenReturn(42);
    // this fails now with an UnnecessaryStubbingException since it is never called in the test
    when(deepThought.otherMethod("some mundane thing")).thenReturn(null);

    // this will now throw a PotentialStubbingProblem Exception since we usually don't want to call methods on mocks without configured behavior
    deepThought.someMethod();

    assertEquals(42, deepThought.getAnswerFor("Ultimate Question of Life, The Universe, and Everything"));
    // verifyNoMoreInteractions now automatically verifies that all stubbed methods have been called as well
    verifyNoMoreInteractions(deepThought);
}
```

5.3. Limitations

Mockito has certain limitations, for example, you cannot mock private methods (https://github.com/mockito/mockito/wiki/Mockito-And-Private-Methods).

See FAQ for Mockito limitations for the details (https://github.com/mockito/mockito/wiki/FAQ#what-are-the-limitations-of-mockito)

5.4. Behavior testing vrs. state testing

Mockito puts a focus on behavior testing, vrs. result testing. This is not always correct, for example, if you are testing a sort algorithm, you should test the result not the internal behavior.

```java
// state testing
testSort() {
    testList = [1, 7, 3, 8, 2]
    MySorter.sort(testList)

    assert testList equals [1, 2, 3, 7, 8]
}


// incorrect would be behavior testing
// the following tests internal of the implementation
testSort() {
    testList = [1, 7, 3, 8, 2]
    MySorter.sort(testList)

    assert that compare(1, 2) was called once
    assert that compare(1, 3) was not called
    assert that compare(2, 3) was called once
    ....
}
```

6. Exercise: Creating mock objects using Mockito

In this exercise you create an simple API. You use Mockito to mock this API in your tests.

6.1. Create a sample Twitter API

Implement a `TwitterClient`, which works with `ITweet` instances. But imagine these `ITweet` instances are pretty cumbersome to get, e.g., by using a complex service, which would have to be started.

```java
package com.vogella.unittest.twitter;

public interface ITweet {
    String getMessage();
}
```

```java
package com.vogella.unittest.twitter;

public class TwitterClient {

    public void sendTweet(ITweet tweet) {
        String message = tweet.getMessage();

        // TODO send the message to Twitter
        System.out.println(message);
    }
}
```

6.2. Mocking ITweet instances

In order to avoid starting up a complex service to get `ITweet` instances, they can also be mocked by Mockito.

```java
package com.vogella.unittest.twitter;

import static org.mockito.Mockito.atLeastOnce;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class TwitterClientTest {

    @Mock
    ITweet tweet;

    @Test
    public void testSendingTweet() {
        TwitterClient twitterClient = new TwitterClient();

        ITweet iTweet = mock(ITweet.class);

        when(iTweet.getMessage()).thenReturn("Using mockito is great");

        twitterClient.sendTweet(iTweet);


    }
}
```

Now the `TwitterClient` can make use of a mocked `ITweet` instance and will get "Using Mockito is great" as message when calling `getMessage()` on the mocked `ITweet`.

6.3. Verify method invocation

In your test, ensure that getMessage() is at least called once.

```java
verify(iTweet, atLeastOnce()).getMessage();
```

6.4. Validate

Run the test and validate that it is successful.

7. Exercise: Testing an API with Mockito

7.1. Create an AudioManager API

Create the following classes which will be used in the following tests.

```java
package com.vogella.unittest.audio;

public enum RINGER_MODE {
    RINGER_MODE_NORMAL, RINGER_MODE_SILENT
}
```

```java
package com.vogella.unittest.audio;

public class AudioManager {
    private int volume = 50;
    private RINGER_MODE mode = RINGER_MODE.RINGER_MODE_SILENT;

    public RINGER_MODE getRingerMode() {
        return mode;
    }
    public int getStreamMaxVolume() {
        return volume;
    }
    public void setStreamVolume(int max) {
        volume = max;
    }
    public void makeReallyLoad() {
        if (mode.equals(RINGER_MODE.RINGER_MODE_NORMAL)) {
            setStreamVolume(100);
        }
    }

}
```

```java
package com.vogella.unittest.audio;

public class ConfigureThreadingUtil {
    public static void configureThreadPool(MyApplication app){
        int numberOfThreads = app.getNumberOfThreads();
        // TODO use information to configure the thread pool


    }
}
```

```java
package com.vogella.unittest.audio;

public class MyApplication {

    public int getNumberOfThreads() {
        return 5;
    }

}
```

```java
package com.vogella.unittest.audio;

public class VolumeUtil {
    public static void maximizeVolume(AudioManager audioManager) {
        if (audioManager.getRingerMode() != RINGER_MODE.RINGER_MODE_SILENT) {
            int max = audioManager.getStreamMaxVolume();
            audioManager.setStreamVolume(max);
        }
    }
}
```

7.2. Testing VolumeUtil

We want to test VolumeUtil using Mockito.

```java
package com.vogella.unittest.audio;


import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

public class VolumeUtilTests {
    @Test
    public void testNormalRingerIsMaximized(){
        // 1.) mock AudioManager
        // 2.) configure Audiomanager to return RINGER_MODE_NORMAL if getRinderMode is called
        // 3.) configure Audiomanager to return 100 if getStreamMaxVolume() is called
        // 4.) call VolumeUtil.maximizeVolume with Audiomanager -> code under test
        // 5.) verify that setStreamVolume(STREAM_RING, 100, 0) was called on audioManager

    }

    @Test
    public void testSilentRingerIsNotDisturbed() {
        // 1. Prepare AudioManager mock
        // 2.) configure audiomanager to return "RINGER_MODE_SILENT" if getRingerMode is called
        // 3.) call VolumeUtil.maximizeVolume with audio manager
        // 4.) verify that getRingerMode()
        // 5.) Ensure that nothing more was called


    }
}
```

▶ Open the drop-down to see the solution

7.3. Test

Write a new test which implements the following test comments.

```java
public class ConfigureThreadingUtilTests {
    @Test
    public void testApplication (){
        // mock MyApplication
        // call ConfigureThreadingUtil.configureThreadPool
        // verify that getNumberOfThreads was called on app
    }
}
```

▶ Open the drop-down to see the solution

8. Mockito resources

Mockito home page (http://site.mockito.org)

Dzone reference card (https://dzone.com/refcardz/mockito)

Mockito project hosting page (https://github.com/mockito/mockito)

Mockito release notes (https://github.com/mockito/mockito/blob/master/doc/release-notes/official.md)

Martin Fowler about Mocks, Stubs etc. (http://martinfowler.com/articles/mocksArentStubs.html)

Chiu-Ki Chan Advanced Android Espresso presentation (http://chiuki.github.io/advanced-android-espresso/)