JUnit 5 (and assertj) and Mockito

**TODO:** Overview of all the Headers

A screenshot of a cell phone

Description automatically generated**JUnit** and **TestNG** are undoubtedly the two most popular unit-testing frameworks in the Java ecosystem. But JUnit is the defacto standard when it comes to testing in java apps. Junit 5 is the newest iteration

A picture containing indoor, table, bottle, photo

Description automatically generated

Unit test can be classified as regression tests. We write the tests in order to constantly control if our code is working as intended. If at any point in time something changes in the code or the overall behaviour of the program, the test will fail and let us know that we broke a part.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated**Why use a testing framrwork?**

Let’s say we have a basic calculator class. Our goal is here to verify that the **actual** **value** we get equals to the **expected** **value**. Those 2 terms are really often used in the testing context.

This example shows just one test case out of many. For instance, what about negative numbers? We need to create a new test and pass negative numbers. When you have many of these tests, the developer has to look to the outputs to see how the tests went. So there are certain steps that a developer needs to do if not using a framework.

A screenshot of a cell phone

Description automatically generatedEach highlighted step will be different for each new test case. But what about the others? The other steps are common for every test case.

This is where Junit and other testing frameworks provides us value. They provide the 3 un-highlighted steps. It will run the tests, verify the results and provide a handy report back to the developer.

A picture containing table, paper, bird

Description automatically generated

**What was the problem with JUnit 4?**

A screenshot of a cell phone

Description automatically generatedIt was created many years ago and it couldn’t keep up with the changing technologies and industry. Another big problem was that it has a monolithic architecture, it has one jar file. When you need junit, you get the jar and don’t have the option to just pick a part of it that you actually need. Junit is an open-source project, so people work on it on their free time. Many bugs and features piled up.

The developers actually started a crowd funding campaign. They said “we have work to do and we need money to do this work”. This was the basis of Junit 5.

A close up of a sign

Description automatically generated**JUnit 5 Architecture**

As we said JUnit 4 was just one jar.

Plattform is the core what composes the library of JUnit, which is involved running the tests, providing a way for you to call the test runner, provides an execution context etc. This core platform is not something that developers directy interact with!

What we interact with is the **JUnit API**, which is called **Jupiter**. The API is called Jupiter since they didn’t want to name it Junit, because Junit is the name of the whole framework. (Jupiter is the 5th planet in the solar system).

JUnit 5 is not backward compatible out of the box. So they created another set of APIs to provide backwards compatibility for older tests. It is called **Vintage**. You can have a project where you have Junit 4 and Junit 5 tests both run in parallel.

You also have a library which let you **extend Junit**. Lets say you don’t like the Junit API. You can come up with your own APIs and still use the underline test engine (Platform).

* The test engine is what runs your tests and Jupiter is the API which informs the engine what to run, how to run it etc.
* You can also extend Jupiter itself. Not writing your own API.
* So you need 2 Maven dependencies to write Junit 5 tests, **Jupiter** and **Platform**.
* A lot of IDEs provide integrated support for Junit 5.

**Example Maven Project**

A screenshot of a cell phone

Description automatically generated

A picture containing bird

Description automatically generated

We are adding the 2 dependencies Junit and Jupiter.

* **<scope> test </scope>** This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive.
* Whatever is in your test scope won’t be included in your final build. (when you do mvn install/deploy)

Text

Description automatically generated

* Contains all the Junit 5 dependencies (I guess)

**Why do we unit test?**

We do unit-testing in order to:

* Make sure that our logic really does what we think it does.
* **IMPORTANT**: Have the ability to refactor our logic at a later time. Without testing, we are afraid to touch our code. The creator fears its creation! Tests gives us the ability to be confident that our code will work even if you refactor production-code someone else wrote after 3 years!

So, unit tests are basically regression tests. They make sure with every build that all the logic that we already have did not change its functionality and still works.

Some points to keep in mind when testing (extract this maybe to “clean code” documents):

Victor Rentea - Unit Testing like a Pro: The Circle of Purity - <https://www.youtube.com/watch?v=1Z_h55jMe-M>

* If you find a bug in your code, don’t just fix it and be done with it. Instead, first write a unit test which reproduces that bug and then fix it to make the test green.

**Creating a Test**

Creating a test is as easy as adding a **@Test** annotation **(org.junit.jupiter.api.Test)**. This way you tell Junit to run this method as a test. Your IDE will have the option to run all the methods which has that annotation and shows a list of all of them.

* If nothing failed (or exceptions thrown) in a method/test, Junit will count it as ‘passed’. “No failures” means success, so if nothing happens in your method, junit will see it as “passed”.
* Tests having the same package as the application code is because that way the package visibility are the same

**Expectation vs Reality**

1. Create an instance of the class under test: In order to call a method, we need an instance of the object.
2. Setup inputs: We need to pass some arguments to the method under test.
3. Execute the code you want to test
4. Verify the results what you expect

After we executed the method, we want to check if the output is actually the same as what we expected. To do that we will use assertions.

* All the assertion methods in JUnit are in the **Assertions** class.
* The methods are static, so we can import them all: **import static org.junit.jupiter.api.Assertions.\*;**

|  |
| --- |
| @Test public void **test**() {  *// we are testing the String.format() method* String actual = String.*format*(**"%s - %s"**, **"test"**, **"123"**);  String expected = **"test - 123"**;   */\*  -> assertEquals(Object expected, Object actual)  -> Assert that expected and actual are equal.  -> If both are null, they are considered equal.  -> invokes the equals() method of passed objects  \*/  assertEquals*(expected, actual); } |

* JUnit 4 **@Test** import is**: import org.junit.Test;**

|  |
| --- |
| *// the actual code which assertEquals() calls* static boolean **objectsAreEqual**(Object obj1,   Object obj2) {  if (obj1 == null) {  return (obj2 == null);  }  return obj1.equals(obj2); } |
|  |

**Some other Assert methods**

|  |
| --- |
| *assertEquals*(expected, actual, **"message to provide extra information on fail"**);  *assertNull*(expected); *assertTrue*(expected == **""**); *assertFalse*(expected == **""**); *assertArrayEquals*(arr1, arr2); *// also checks order* assertIterableEquals(...);  *fail*(**"message"**); |

**Maven Surefire Plugin Integration**

Using the build in IDE Test overview is fine for development but when the application goes to for instance to a CI/CD pipeline, the build process won’t install Eclipse and right click -> run all tests. We need a command line way of doing this.

This is where we’ll use the maven way. In order to do that we need the **Maven Surefire Plugin**.

* **mvn test** -> will not run any tests, even if we have @Test annotations

|  |
| --- |
| *<!-- MAVEN SUREFIRE PLUGIN -->* <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.22.2</version> </plugin> |

Now when we do a **mvn test**, all of our test will run.

**Asserting Exceptions with assertThrows()**

Until now we just tested one possible path of execution in our methods, which is the happy path.

* **Happy-path** testing is a type of software testing that uses known input and produces an expected output.   
  Also referred to as golden-path or sunny-day testing

But there is another path which will be taken when an exception is thrown for instance. We often wan’t that our method throws and exception if something happens and we also want to test that.

* Asserting exceptions was more difficult to do in Junit 4 but thanks to lambdas, we can do it much more easily.

|  |
| --- |
| */\*  This test will fail for 2 reasons:   1. If the method does not throw any exceptions  2. If the method throws a different exception   \*/* @Test public void **testException**() {  *assertThrows*(ArithmeticException.class, () -> *divide*(4, 0)); } |

**Lifecycle and Test Antipatterns to Avoid**

A Test-Lifecycle is the process in which the test instance is created, managed and destroyed. Unlike other Classes in our codebase, when we run our tests, who is initializing our …Test class? Junit is doing the **new …Test().**

* Junit gives us hooks to access the different Lifecycle-Phases.
* There are no saying what the order of test method execution is going to be! It is essentially random. Junit has an **@Order** annotation but if you need to use that then you have a serious problem. Your tests shoud not depend on other tests or any execution order.
* **Junit creates a new class instance for each test method run**! Don’t try share instance variables in your test methods, meaning do not define a field and let one method change it and then check in the other method if it actually changed! Since all test methods will be executed on different objects, they all have separate state!

**Test Class and Test Method**

Test Class: any top-level class, static member class, or **@Nested** class that contains at least one test method. Test classes must not be abstract and must have a single constructor.

Test Method: any instance method that is directly annotated or meta-annotated with

* @Test
* @RepeatedTest
* @ParameterizedTest
* @TestFactory
* @TestTemplate

**JUnit Lifecycle Hook Annotations**

Lifecycle Method: any method that is directly annotated or meta-annotated with**:**

* @BeforeAll
* @AfterAll
* @BeforeEach
* @AfterEach

Test methods and lifecycle methods:

* must not be abstract (because they can be subclassed)
* must not return a value
* must not be private (because they can be subclassed)

Table

Description automatically generated

Text

Description automatically generated with medium confidence

**@BeforeAll** and **@AfterAll** will execute before the instance is created / after the instance is terminated and only once. Therefore it also needs to be a static method (unless **Lifecycle.PER\_CLASS** because then there is only one instance and JUnit can run those as instance methods)

**Changing the default TestInstance behavior**

The@TestInstance annotation is used to configure the test instance lifecycle for the annotated test class.

Test Instance Lifecycle: In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each test method (see Test Classes and Methods). This "per-method" test instance lifecycle is the default behavior in JUnit Jupiter and is analogous to all previous versions of JUnit.

Text

Description automatically generated

If you would prefer that JUnit Jupiter execute all test methods on the same test instance, annotate your **test class** with

@TestInstance(Lifecycle.***PER\_CLASS***). When using this mode, a new test instance will be created once per test class. Thus, if your test methods rely on state stored in instance variables, you may need to reset that state in **@BeforeEach** or **@AfterEach** methods.

* Lifecycle.***PER\_METHOD*** is the default

**Other Annotations**

**@DisplayName**

Test classes and test methods can declare custom display names via **@DisplayName** — with spaces, special characters, and even emojis — that will be displayed in test reports and by test runners and IDEs. In some projects it is required

Text

Description automatically generatedBy default the tests names are the names of your class and methods. The @DisplayName can change that.

@DisplayName(**"--- Testing DropboxClient ---"**)



**@DisplayNameGeneration**

JUnit provides different name generators to automatically change the name of your methods/classes which are displayed on the test reports.

* **@DisplayName** annotation always takes precedence over any display name generator.

One built-in generator is @DisplayNameGeneration(**DisplayNameGenerator**.ReplaceUnderscores.class) which will replace the underscores in the name with spaces.

* Standard: Matches the standard display name generation behavior in place since JUnit Jupiter 5.0 was released.
* Simple: Removes trailing parentheses for methods with no parameters.
* ReplaceUnderscores: Replaces underscores with spaces.
* IndicativeSentences: Generates complete sentences by concatenating the names of the test and the enclosing classes.

You can also give different separators to be used or even write your own name generator.

**@Disabled**

Used to disable a test class or test method; analogous to JUnit 4’s @Ignore. Such annotations are not inherited.

It is a better alternative to deleting the @Test annotation, which can lead to some confusion in the future since a non-annotated method won’t show up in the test reports. Using **@Disabled** will still show the method name but it will be marked as skipped/disabled in the test reports.

Text

Description automatically generatedText

Description automatically generated

**@Timeout**

Used to fail a test, test factory, test template, or lifecycle method if its execution exceeds a given duration. Such annotations are inherited.

**@TempDir (Experimental)**

Used to supply a temporary directory via field injection or parameter injection in a lifecycle method or test method; located in the org.junit.jupiter.api.io package.

**Conditional Execution**

* @EnabledOnOs(OS.LINUX)
* @EnabledOnJre(JRE.JAVA\_11)
* @EnabledIf
* @EnabledIfSystemProperty
* @EnabledIfEnvironmentVariable

If the conditions won’t match then the test will be disabled.

There are also other ways we can handle conditional execution which depends on external factors:

* **assumeThat(boolean)**: if the boolean condition is false, then then the method will be skipped.

This gives you programatic control over if you want to execute a method or not. You can have logic to calculate the boolean value, depending on external factors. For instance your test requires a server to be up and in the context it will not be a failiure condition, it’s sometimes down and we don’t want our test to fail if it’s down. So if the test will fail, you will know for sure that it was not because the server was down.

**Using assertAll()**

A picture containing text, person

Description automatically generatedThis is usefull when you have a bunch of assertions which can be grouped together. If any of them fails then the assertion will fail. It’s like an AND condition for your asserts.

It takes multiple **Executable** instances (Lambdas can be used) and runs them all.

**Lazy Assert Messages**

You can pass lambdas to any assertion for the message, which will be displayed if the test fails.

If creating a message involves and expensive process which takses up some resources, then it is not the best option to create that message for each time its used, for every test case. Most of the time it will not even be displayed because most tests will pass.

* You can give a **Supplier** lambda as an argument. It only executes that lambda if the test fails.  
  So, we can pass: **() -> “Failed because: “ + calculatedString**

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application

Description automatically generated

**Tagging**

Lets you tag your tests with certain words. Maybe you want to separate your unit tests and integration tests, or you maybe want to have an “important” tag for certain methods and only run them in certain scenarios.

* Text

  Description automatically generatedGraphical user interface, text, application, email

  Description automatically generatedUse **@Tag(“myTag”)** on methods or classes.

You can then have the ability to control which methods with which tags will be triggered.

IDE’s also have run configurations where you can set profiles which should run only certain tags.

**Nested Classes**

Sometimes if you put all your test methods in one test class, that class becomes really large. Nested classes help you organize your code if you have larger classes, a lot of methods.

Graphical user interface, text, application, email

Description automatically generatedYou can either create an inner class and annotate it with @Nested, which is a way to group similar test cases. It has the benefit to be grouped in tests reports.

Graphical user interface, text, application, chat or text message

Description automatically generated

**Typical use case**

Very often, developer teams define a test class by class to test. That is a shared good practice but it also may make your test class very big and to count several hundred of lines. You can indeed have classes to test with multiple methods to test, multiple scenarios for each one and also some initialization steps required in the unit test methods to test the scenarios.All of these will naturally increase the test class size.Above a threshold (maybe 500 lines or about), it becomes legitimate to ask yourself whether a refactoring is needed.

A big class (test class or not), even well organized is harder to read, maintain than multiple classes grouping things with high cohesion/relationship between.In the unit tests cases, it can be sometime still worse because you may not find a test scenario and write a new one while it existed but you didn't manage to find it because the test class is big.

**The solution: @Nested**

**@Nested** addresses this issue by giving the possibility to group multiple test methods inside multiple nested classes of a main(outer) test class.The test methods of all nested classes defined in the main(outer) test class are handled as any test methods. So **@BeforeEach, @AfterEach, @ExtendWith**... are applied for all the them.

The single exception is @BeforeAll and @AfterAll :

* Only non-static nested classes (i.e. inner classes) can serve as **@Nested** test classes. Nesting can be arbitrarily deep, and those inner classes are considered to be full members of the test class family with one exception: **@BeforeAll** and **@AfterAll** methods do not work by default. The reason is that Java does not allow static members in inner classes. However, this restriction can be circumvented by annotating a **@Nested** test class with **@TestInstance(Lifecycle.PER\_CLASS)** (see Test Instance Lifecycle).

Text

Description automatically generated

This is also usefull when the goal is to craft full real sentences for test cases and/or when using **@DisplayNameGeneration** to make the names more readable and structured.

**TODO What I don’t understand**: inheritance? Just create another class?

**Repeated Test**

Takes the place of @Test. Takes an argument: how many times you want the test to be repeated.

A picture containing logo

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated

You can also get hold of the repetition information. JUnit can pass that as a parameter to your method.

Lets say you want to do something different for only the second repetition.

Graphical user interface, text, application, chat or text message

Description automatically generated

**Parameterized Tests with @ParameterizedTest**

TODO - <https://www.baeldung.com/parameterized-tests-junit-5>

**Dynamic Tests with @TestFactory**

TODO - <https://www.baeldung.com/junit5-dynamic-tests>

**Test Templates with @TestTemplate**

TODO - <https://www.baeldung.com/junit5-test-templates>

Graphical user interface, text

Description automatically generated

**Using TestInfo and TestReporter**

TODO

**Tips / Best Practices**

* Place verify() statements before the assert() statements.

**TODO, testing if files exist**





**Assertj**

The AssertJ project provides fluent assertion statements for Java. These assert statements are typically used with Java JUnit tests. AssertJ is a fork of the Fest assert library, as Fest is not actively maintained anymore.

AssertJ is a library for simplifying the writing of assert statements in tests. It also improves the readability of asserts statements. It has a fluent interface for assertions, which makes it easy for your code completion to help your write them. The base method for AssertJ assertions is the assertThat method followed by the assertion.

<https://www.vogella.com/tutorials/AssertJ/article.html>

**Effective Unit Testing by Eliotte Rusty Harold**

<https://www.youtube.com/watch?v=fr1E9aVnBxw>

**Write awesome tests by Jeroen Mols**

<https://www.youtube.com/watch?v=F8Gc8Nwf0yk>

**Mockito Best Practices**

**@ExtendWith @RunWith difference?**

* Junit version < 5: use **@RunWith(SpringRunner.class)** **or @RunWith(MockitoJUnitRunner.class)**
* Junit version = 5: use **@ExtendWith(SpringExtension.class)** **or @ExtendWith(MockitoExtension.class)**

In JUnit 5, the **@RunWith** annotation has been replaced by the more powerful @ExtendWith annotation.

However, the @RunWith annotation can still be used in JUnit5 for the sake of the backward compatibility.

Graphical user interface, text, application, email

Description automatically generated

**Rules to keep in mind while writing tests**

* Keep the testing code compact and readable
* Cover as much of the range as possible to show positive cases and especially erroneous code paths.
* Avoid coding a tautology
* Don't mock everything, it's an anti-pattern
* Don’t mock value objects
* Do not mock types you don’t own
* Write unit tests for a bug before fixing it.
* Always see your unit tests failing, to make sure you really cover the lines you think you do.

**Keep the testing code compact and readable**

To do that, apply merciless refactoring, just as one should do for production code. Otherwise letting things rot will be like creating the dreaded legacy code on the test side. If tests cannot be easily refactored, then the production code will be hard to refactor as well, leading to legacy production code. Always follow the route of the brave refactorer.

**Avoid coding a tautology**

Where e.g. the test code generates things using the very same regexp that is used in the parser.Generally speaking one does not want to duplicate the logic between tests and code. So replicating a regexp or something else in the test is not an option. In this case, thinking about testing input stimulus / output result helps (f(input) -> output), for example if the code is supposed to process a template, don't add values. Instead, test against a computed result.

|  |
| --- |
| *// use* Assertions.assertThat(processTemplate(**"param1"**, **"param2"**)).isEqualTo(**"this is 'param1', and this is 'param2'"**));  *// instead of* Assertions.assertThat(processTemplate(**"param1"**, **"param2"**)).isEqualTo(String.format(**"this is '%s', and this is '%s'"**, param1, param2)); |

**Don't mock everything, it's an anti-pattern**

If everything is mocked, are we really testing the production code? Don't hesitate to not mock!

**Don't mock value objects**

Why one would even want to do that? Because instantiating the object is too painful !? => not a valid reason.

If it's too difficult to create new fixtures, it is a sign the code may need some serious refactoring. An alternative is to create builders for your value objects -- there are tools for that, including IDE plugins, Lombok, and others. One can also create meaningful factory methods in the test classpath.

|  |
| --- |
| abstract class CustomerCreations {  public static Customer **customer\_with\_a\_single\_item\_in\_the\_basket**() {  *// long init sequence* } } |

Mockito is about focusing on interactions between objects, which is the most essential part of Object Oriented Programming (or messaging).

**Don't mock a type you don't own! (Not sure about this)**

This is not a hard line, but crossing this line may have repercussions! (it most likely will)

TDD is just as much about design as it is about test, when mocking an external API the test cannot be used to drive the design, the API belongs to someone else ; this third party can and will change the signature and behaviour of the API.

1. Imagine code that mocks a third party lib. After a particular upgrade of a third library, the logic might change a bit, but the test suite will execute just fine, because it's mocked. So later on, thinking everything is good to go, the build-wall is green after all, the software is deployed and... Boom
2. It may be a sign that the current design is not decoupled enough from this third party library.
3. Also another issue is that the third party lib might be complex and require a lot of mocks to even work properly. That leads to overly specified tests and complex fixtures, which in itself compromises the compact and readable goal. Or to tests which do not cover the code enough, because of the complexity to mock the external system.

Instead, the most common way is to create wrappers around the external lib/system, though one should be aware of the risk of abstraction leakage, where too much low level API, concepts or exceptions, goes beyond the boundary of the wrapper. In order to verify integration with the third party library, write integration tests, and make them as compact and readable as possible as well.

Other people have already written on the matter and experienced pain when mocking a type they didn't own:

* <http://davesquared.net/2011/04/dont-mock-types-you-dont-own.html>
* <http://www.markhneedham.com/blog/2009/12/13/tdd-only-mock-types-you-own>
* <http://blog.8thlight.com/eric-smith/2011/10/27/thats-not-yours.html>
* <http://stackoverflow.com/questions/1906344/should-you-only-mock-types-you-own>

Arguments for/against against “Don’t mock a foreign type”

* <https://testing.googleblog.com/2020/07/testing-on-toilet-dont-mock-types-you.html>

**Why Mockito doesn't mock private methods?**

Firstly, we are not dogmatic about mocking private methods. We just don't care about private methods because from the standpoint of testing, private methods don't exist. Here are a couple of reasons Mockito doesn't mock private methods:

1. It requires hacking of classloaders that is never bullet proof and it changes the API (you must use custom test runner, annotate the class, etc.).
2. It is very easy to work around - just change the visibility of method from private to package-protected (or protected).
3. It requires the team to spend time implementing & maintaining it. And it does not make sense given point (2) and a fact that it is already implemented in different tool (powermock).
4. Finally... Mocking private methods is a hint that there is something wrong with Object Oriented understanding. In OO you want objects (or roles) to collaborate, not methods. Forget about pascal & procedural code. Think in objects.

**What are the limitations of Mockito?** - *Mockito 3.x specific limitations*

* Requires Java 8+
* Cannot mock constructors

**Mockito Usage**

Docs:

* <https://www.javadoc.io/doc/org.mockito/mockito-core/1.10.19/org/mockito/Mockito.html>

**Mock vs Stub vs Fake …?**

**What is the difference between mocking and spying when using Mockito?**

<https://stackoverflow.com/questions/15052984/what-is-the-difference-between-mocking-and-spying-when-using-mockito>

**Use Mockito to mock some methods but not others (NICE!!!)**

<https://stackoverflow.com/questions/14970516/use-mockito-to-mock-some-methods-but-not-others>

**Mockito: Trying to spy on method is calling the original method**

<https://stackoverflow.com/questions/11620103/mockito-trying-to-spy-on-method-is-calling-the-original-method>

**Private field dependency injection**

<https://www.linkedin.com/pulse/avoid-private-field-dependency-injection-here-why-m%C3%A1rio-j%C3%BAnior>

**Testing made sweet with a Mockito by Jeroen Mols**

<https://www.youtube.com/watch?v=DJDBl0vURD4>

**Different ways to use Mockito**

**Manually Creating mocks**

Manually with **Mockito::mock** works regardless of the JUnit version (or test framework for that matter).

|  |
| --- |
| GoogleClient googleClient = *mock*(GoogleClient.class);  DbxClientV2 dropboxService = *mock*(DbxClientV2.class, ***RETURNS\_DEEP\_STUBS***); |

**Annotation Based**

Using the **@Mock**-annotation and the corresponding call to **MockitoAnnotations::initMocks** to create mocks works regardless of the JUnit version (or test framework for that matter but Java 9 could interfere here, depending on whether the test code ends up in a module or not).

|  |
| --- |
| @Mock(answer = Answers.***RETURNS\_DEEP\_STUBS***) private DbxClientV2 dropboxService;   @BeforeEach public void **before**() {  *// .initMocks() is deprecated*  MockitoAnnotations.*~~initMocks~~*(this); } |

**Mockito Extension**

JUnit 5 has a powerful extension model and Mockito recently published one under the group / artifact ID org.mockito : mockito-junit-jupiter.

You can apply the extension by adding **@ExtendWith(MockitoExtension.class)** to the test class and annotating mocked fields with **@Mock**.

The MockitoExtension documentation describes other ways to instantiate mocks, for example with constructor injection (if you pefer final fields in test classes).

|  |
| --- |
| @ExtendWith(MockitoExtension.class) public class DropboxClientTest {   @Mock(answer = Answers.***RETURNS\_DEEP\_STUBS***)  private DbxClientV2 dropboxService;  ...  } |

asd

**13. Spying on real objects**

You can create spies of real objects. When you use the spy then the real methods are called (unless a method was stubbed). Real spies should be used carefully and occasionally, for example when dealing with legacy code.

Spying on real objects can be associated with "**partial mocking**" concept. Before the release 1.8, Mockito spies were not real partial mocks. The reason was we thought partial mock is a code smell. At some point we found legitimate use cases for partial mocks (3rd party interfaces, interim refactoring of legacy code, the full article is here)

**Important gotcha on spying real objects!**

1. Sometimes it's impossible or impractical to use **when(Object)** for stubbing spies. Therefore when using spies please consider **doReturn|Answer|Throw()** family of methods for stubbing. Example:

|  |
| --- |
| **List** list = new LinkedList(); **List** spy = *spy*(list);  *// Impossible: real method is called so spy.get(0)*  *// throws IndexOutOfBoundsException (the list is yet empty) when*(spy.get(0)).thenReturn(**"foo"**);  *// You have to use doReturn() for stubbing doReturn*(**"foo"**).when(spy).get(0); |

1. Mockito \***does not**\* delegate calls to the passed real instance, instead it actually creates a copy of it. So if you keep the real instance and interact with it, don't expect the spied to be aware of those interaction and their effect on real instance state. The corollary is that when an \***unstubbed**\* method is called \***on the spy**\* but \***not on the real instance**\*, you won't see any effects on the real instance.
2. Watch out for final methods. Mockito doesn't mock final methods so the bottom line is: **when you spy on real objects + you try to stub a final method = trouble**. Also you won't be able to verify those method as well.

* verifyNoInteractions does not acceps a spy object:   
  **org.mockito.exceptions.misusing.NotAMockException:Argument(s) passed is not a mock!**

**Important!**

Invocations and interactions inside the test method will be also tracked, so if you do

When(myMock.doStuff()).thenReturn

And if myMock.doStuff() is also called inside the method which is tested, then the verify will say there was 2 interactions!

**Questions**

1. Difference??:
   * when(mock.isOk()).thenThrow(exception);
   * doThrow(exception).when(mock).someVoidMethod();
2. When to test if an exception is thrown? Only when we throw it with throw new … or also if the 3rd part library throws it?
3. Is clearInvocations() really needed?

**Example for bad testibility**:

|  |
| --- |
| @Slf4j public class DropboxClient {  public static final String ***KEY\_DROPBOX\_ACCESS\_TOKEN*** = **"DROPBOX\_ACCESS\_TOKEN"**;  private final String dropboxAccessToken;  private DbxClientV2 dbxClient;   DropboxClient(**Dotenv** dotenv) {  dropboxAccessToken = dotenv.get(***KEY\_DROPBOX\_ACCESS\_TOKEN***);if (StringUtils.*isBlank*(dropboxAccessToken)) {  ***log***.info(**"Cannot instantiate instance because {} is empty"**, ***KEY\_DROPBOX\_ACCESS\_TOKEN***);  } else {  DbxRequestConfig config = DbxRequestConfig.*newBuilder*(**"dropbox/notion-backup"**).build();  dbxClient = new DbxClientV2(config, dropboxAccessToken);  }  }   public void **upload**(File fileToUpload) {  if (StringUtils.*isBlank*(dropboxAccessToken)) {  ***log***.info(**"Skipping upload to Dropbox because {} is empty"**, ***KEY\_DROPBOX\_ACCESS\_TOKEN***);  return;  }  *// This does not override if it's the same file with the same name and silently executes  // Throws an UploadErrorException if we try to upload a different file with an already existing name* ***log***.info(**"Uploading file '{}' to Dropbox..."**, fileToUpload.getName());  try (InputStream in = new FileInputStream(fileToUpload)) {  *// without slash: IllegalArgumentException: String 'path' does not match pattern* dbxClient.files().uploadBuilder(**"/"** + fileToUpload.getName()).uploadAndFinish(in);   if (doesFileExist(fileToUpload.getName())) {  ***log***.info(**"Successfully uploaded '{}' to Dropbox"**, fileToUpload.getName());  } else {  ***log***.warn(**"Could not upload '{}' to Dropbox"**, fileToUpload.getName());  }  } catch (IOException | DbxException e) {  ***log***.warn(**"Exception during upload of file '{}'"**, fileToUpload.getName(), e);  }  }   public boolean **doesFileExist**(String fileName) throws DbxException {  System.***out***.println(**" -> doesFileExist()"**);  ListFolderResult result = dbxClient.files().listFolder(**""**);  return result.getEntries().stream()  .anyMatch(entry -> StringUtils.*equalsIgnoreCase*(entry.getName(), fileName));  } } |

|  |
| --- |
| *// given* **Dotenv** dotenvMock = *mock*(**Dotenv**.class); FileMetadata fileMetadataMock = *mock*(FileMetadata.class); DbxClientV2 dbxClientMock = *mock*(DbxClientV2.class, ***RETURNS\_DEEP\_STUBS***); *// Mock a fluent API object  // when when*(dotenvMock.get(*any*())).thenReturn(**"some string"**); *// fluent API object call when*(dbxClientMock.files().uploadBuilder(*anyString*()).uploadAndFinish(*any*())).thenReturn(fileMetadataMock); DropboxClient testee = new DropboxClient(dotenvMock); FieldUtils.*writeField*(testee, **"dbxClient"**, dbxClientMock, true); testee = *spy*(testee); *// when(testee.doesFileExist(anyString())).thenReturn(true); // Will call the actual .doesFileExist() method! doReturn*(true).when(testee).doesFileExist(*anyString*()); *// You have to use doReturn() for stubbing  // Mocking a file is not a good idea* File testFileToUpload = new File(**"src/test/resources/testFileToUpload.txt"**); if (!testFileToUpload.exists()) {  System.***out***.println(**"Test file does not exist..."**); } testee.upload(testFileToUpload);  *// then verify*(dotenvMock).get(DropboxClient.***KEY\_DROPBOX\_ACCESS\_TOKEN***); *verify*(dbxClientMock.files().uploadBuilder(**"/testFileToUpload.txt"**)).uploadAndFinish(*any*(FileInputStream.class)); *verify*(testee).doesFileExist(**"testFileToUpload.txt"**); |