This tutorial explains unit testing with JUnit with the JUnit 5 framework (JUnit Jupiter). It explains the creation of JUnit 5 tests with the Maven and Gradle build system. It also demonstrates the usage of the Eclipse IDE for developing software tests with JUnit 5.

1. Overview

This tutorial gives an introduction into unit testing with the JUnit framework using JUnit 5.

It starts with a introduction about https://www.vogella.com/tutorials/JUnit/article.html#testintroductiontesting] in general. Afterwards it introduces JUnit (https://www.vogella.com/tutorials/JUnit/article.html#junitintroduction). The setup to use JUnit in your project with Maven or Gradle is described in Setting up the usage of JUnit (https://www.vogella.com/tutorials/JUnit/article.html#junitsetup).

The usage of the Eclipse IDE is described in

And finally you get a full exercise how to create a new project in the Eclipse IDE and use JUnit 5 in it in the Using JUnit 5 with Maven and the Eclipse IDE (https://www.vogella.com/tutorials/JUnit/article.html#junitmaveneclipse).

I hope you have fun while going through the tutorial.

2. Introduction into software testing

The following text introduces the purpose and base technology of software testing, if you interested in the usage of the JUnit 5 test framework, skip to the next section.

3. Introduction to testing

3.1. The purpose of software tests

A software test is a piece of software, which executes another piece of software asserting that it behaves in a certain way.

With software tests you ensure that certain parts of our software work as expected. These tests are typically executed automatically via the build system and therefore help the developer to avoid breaking existing code during development activities.

Running tests automatically helps to identify software regressions introduced by changes in the source code. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

You should write software tests for the critical and complex parts of your application. If you introduce new features a solid test suite also protects you against regression in existing code.

Some developers believe every statement in your code should be tested but in general it is safe to ignore trivial code. For example, it is typical useless to write tests for getter and setter methods which simply assign values to fields. Writing tests for these statements is time consuming and pointless, as you would be testing the Java virtual machine. The JVM itself already has test cases for this. If you are developing end user applications you are safe to assume that a field assignment works in Java.

If you start developing tests for an existing code base without any tests, it is good practice to start writing tests for code in which most of the errors happened in the past. This way you can focus on the critical parts of your application.

3.2. Testing terminology

The code which is tested is typically called the *code under test*. If you are testing an application, this is called the *application under test*.

A *test fixture* is a fixed state of a set of objects which are used as a baseline for running tests. Another way to describe this is a test precondition.

For example, a test fixture might be a a fixed string, which is used as input for a method. The test would validate if the method behaves correctly with this input.

3.2.1. Unit, integration and performance tests

In testing you distinguish between unit, integration tests and performance tests:

- A *unit test* is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state. The percentage of code which is tested by unit tests is typically called *test coverage*. A unit test targets a small unit of code, e.g., a method or a class. External dependencies should be removed from unit tests, e.g., by replacing the dependency with a test implementation or a (mock) object created by a test framework. Unit tests are not suitable for testing complex user interfaces or component interaction. For this, you should develop integration tests.

- An *integration test* aims to test the behavior of a component or the integration between a set of components. The term *functional test* is sometimes used as synonym for integration test. Integration tests check that the whole system works as intended, therefore they are reducing the need for intensive manual tests. These kinds of tests allow you to translate your user stories into a test suite. The test would resemble an expected user interaction with the application.
- Performance tests are used to benchmark software components repeatedly. Their purpose is to ensure that the code under test runs fast enough even if it's under high load.

3.2.2. Behavior vs. state testing

A test is a behavior test (also called interaction test) if it checks if certain methods were called with the correct input parameters. A behavior test does not validate the result of a method call.

State testing is about validating the result. Behavior testing is about testing the behavior of the application under test.

If you are testing algorithms or system functionality, in most cases you may want to test state and not interactions. A typical test setup uses mocks or stubs of related classes to abstract the interactions with these other classes away Afterwards you test the state or the behavior depending on your need.

4. Setup for using JUnit 5

This section describes the project setup to add the JUnit 5 libraries to your project If you project is already configured to use JUnit 5 you can skip this section.

▶ Usage of JUnit 5 with Maven

▶ Usage of JUnit 5 with Gradle

▶ Not using Gradle or Maven

5. Introduction into JUnit

JUnit (http://junit.org/) is a Java test framework and an open source project hosted at Github (https://github.com/junit-team/junit). JUnit 5 (also known as Jupiter) is the latest major release of JUnit. It consists of a number of discrete components:

- JUnit Platform - foundation layer which enables different testing frameworks to be launched on the JVM
- Junit Jupiter - is the JUnit 5 test framework which is launched by the JUnit Platform

- JUnit Vintage - legacy TestEngine which runs older tests

As the usage of JUnit 5 requires multiple libraries to be present, you typically use it with a build system like Maven or Gradle. JUnit 5 needs at least Java 8 to run.

5.1. How to define a test in JUnit?

A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*. To define that a certain method is a test method, annotate it with the `@Test` annotation.

This method executes the code under test. You use an *assert* method, provided by JUnit or another assert framework, to check an expected result versus the actual result. These calls are typically called *asserts* or *assert statements*.

Assert statements typically allow to define messages which are shown if the test fails. You should provide here meaningful messages to make it easier for the user to identify and fix the problem. This is especially true if someone looks at the problem, who did not write the code under test or the test code.

5.2. Example JUnit 5 test

The following example demonstrates the usage of a JUnit test.

Assume you have the following class which you want to test.

```java
package com.vogella.junit5;

public class Calculator {

    public int multiply(int a, int b) {
        return a * b;
    }
}
```

A test class for the above class could look like the following.

```java
package com.vogella.junit5;

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

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;

public class CalculatorTest {

    private Calculator calculator;

    @BeforeEach                                                 (1)
    public void setUp() throws Exception {
        calculator = new Calculator();
    }

    @Test                                                       (2)
    @DisplayName("Simple multiplication should work")          (3)
    public void testMultiply() {
        assertEquals(20, calculator.multiply(4,5),              (4)
        "Regular multiplication should work");                  (5)
    }

    @RepeatedTest(5)                                            (6)
    @DisplayName("Ensure correct handling of zero")
    public void testMultiplyWithZero() {
        assertEquals(0, calculator.multiply(0,5), "Multiple with zero should be zero");
        assertEquals(0, calculator.multiply(5,0), "Multiple with zero should be zero");
    }
}
```

1. Runs before each test
2. Defines a test method
3. Name of the test to be displayed by the test runner
4. assert statement validating expected and actual value
5. error message in case the test fails
6. test which run multiple times, in this example 5 times

5.3. Test class and test methods naming conventions

There are several potential naming conventions for JUnit tests. A widely-used solution for classes is to use the *Test* suffix at the end of test classes names.

As a general rule, a method name for a name should explain what the test does. If that is done correctly, reading the actual implementation can be avoided.

One possible convention is to use the "should" in the test method name. For example, `ordersShouldBeCreated` or `menuShouldGetActive`. This gives a hint what should happen if the test method is executed.

Another approach is to use `given[ExplainYourInput]When[WhatIsDone]Then[ExpectedResult]` for the display name of the test method.

With JUnit5 naming the test method is less important, as you can use the `@DisplayName` annotation to add a description for the test method.

5.4. Important JUnit annotations

JUnit uses annotations to mark methods as test methods and to configure them. The following table gives an overview of the most important annotations in JUnit 5. All these annotations are part of the `org.junit.jupiter.api` package.

Table 1. Annotations

Annotation	Description
`@Test`	Identifies a method as a test method.
`@Disabled("reason")`	Disables a test method with an option reason.
`@BeforeEach`	Executed before each test. Used to prepare the test environment, e.g., initialize the fields in the test class, configure the environment, etc.
`@AfterEach`	Executed after each test. Used to cleanup the test environment, e.g., delete temporary data, restore defaults, cleanup expensive memory structures.
`@DisplayName("<Name>")`	<Name> that will be displayed by the test runner. In contrast to method names the name can contain spaces to improve readability.
`@RepeatedTest(<Number>)`	Similar to `@Test` but repeats the test a <Number> of times

Annotation	Description
`@BeforeAll`	Annotates a method which is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
`@AfterAll`	Annotates a method which is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
`@TestFactory`	Annotates a method which is a Factory for creating dynamic tests
`@Nested`	Lets you nest inner test classes to force a certain execution order
`@Tag("<TagName>")`	Tags a test method, tests in JUnit 5 can be filtered by tag. E.g., run only tests tagged with "fast".
`@ExtendWith`	Lets you register an Extension class that adds functionality to the tests

5.5. Flexible disabling of tests

The `@Disabled` or `@Disabled("Why disabled")` annotation marks a test to be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted of if the test demonstrates an incorrect behavior in the code which has not yet been fixed. It is best practice to provide the optional description, why the test is disabled.

Alternatively you can use `Assumptions.assumeFalse` or `Assumptions.assumeTrue` to define a condition for test deactivation. `Assumptions.assumeFalse` marks the test as invalid, if its condition evaluates to true. `Assumptions.assumeTrue` evaluates the test as invalid if its condition evaluates to false. For example, the following disables a test on Linux:

```java
Assumptions.assumeFalse(System.getProperty("os.name").contains("Linux"));
```

This gives `TestAbortedException` which the test runners evaluate as skipped tests.

For example the following `testMultiplyWithZero` is skipped if executed on Linux.

```java
package com.vogella.junit5;

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

import org.junit.jupiter.api.Assumptions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;

public class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void setUp() throws Exception {
        calculator = new Calculator();
    }

    @RepeatedTest(5)
    @DisplayName("Ensure correct handling of zero")
    void testMultiplyWithZero() {
        Assumptions.assumeFalse(System.getProperty("os.name").contains("Linux"));

        assertEquals(calculator.multiply(0,5), 0, "Multiple with zero should be zero");
        assertEquals(calculator.multiply(5,0), 0, "Multiple with zero should be zero");
    }
}
```

You can also write an extension for @ExtendWith which defines conditions under which a test should run.

5.6. Static imports and unit testing

JUnit tests frequently use static imports to make the test short and easy to read. Static imports are a Java feature that allows fields and methods defined in a class as `public static` to be used without specifying the class in which the field is defined.

JUnit assert statements are typically defined as `public static` to allow the developer to write short test statements. The following snippet demonstrates an assert statement with and without static imports.

```java
// without static imports you have to write the following statement
import org.junit.jupiter.api.Assertions;
// more code
Assert.assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));


// alternatively define assertEquals as static import
import static org.junit.jupiter.api.Assertions.assertEquals;
// more code
// use assertEquals directly because of the static import
assertEquals(calculator.multiply(4,5), 20, "Regular multiplication should work");
```

5.7. Expecting Exceptions

Exception is handling with `org.junit.jupiter.api.Assertions.expectThrows()`. You define the expected Exception class and provide code that should throw the exception.

```java
import static org.junit.jupiter.api.Assertions.assertThrows;

@Test
void exceptionTesting() {
    // set up user
    Throwable exception = assertThrows(IllegalArgumentException.class, () -> user.setAge("23"));
    assertEquals("Age must be an Integer.", exception.getMessage());
}
```

This lets you define which part of the test should throw the exception. The test will still fail if an exception is thrown outside of this scope.

5.8. Grouped assertions with assertAll

If an assert fails in a test, JUnit will stop executing the test and additional asserts are not checked. In case you want to ensure that all asserts are checked you can `assertAll`.

In this grouped assertion all assertions are executed, even after a failure. The error messages get also grouped together.

```java
@Test
void groupedAssertions() {
    Address address = new Address();
    assertAll("address name",
        () -> assertEquals("John", address.getFirstName()),
        () -> assertEquals("User", address.getLastName())
    );
}
```

If these tests fail, the result looks like the following:

```
=> org.opentest4j.MultipleFailuresError: address name (2 failures)
expected: <John> but was: <null>
expected: <User> but was: <null>
```

5.9. Timeout tests

If you want to ensure that a test fails, if it isn't done in a certain amount of time you can use the `assertTimeout()` method. This assert fails the method if the timeout is exceeded.

```java
import static org.junit.jupiter.api.Assertions.assertTimeout;
import static java.time.Duration.ofSeconds;
import static java.time.Duration.ofMinutes;

@Test
void timeoutNotExceeded() {
    assertTimeout(ofMinutes(1), () -> service.doBackup());
}

// if you have to check a return value
@Test
void timeoutNotExceededWithResult() {
    String actualResult = assertTimeout(ofSeconds(1), () -> {
        return restService.request(request);
    });
    assertEquals(200, request.getStatus());
}
```

```java
=> org.opentest4j.AssertionFailedError: execution exceeded timeout of 1000 ms by 212 ms
```

If you want your tests to cancel after the timeout period is passed you can use the `assertTimeoutPreemptively()` method.

```java
@Test
void timeoutNotExceededWithResult() {
    String actualResult = assertTimeoutPreemptively(ofSeconds(1), () -> {
        return restService.request(request);
    });
    assertEquals(200, request.getStatus());
}
```

```java
=> org.opentest4j.AssertionFailedError: execution timed out after 1000 ms
```

5.10. Running the same test repeatedly on a data set

Sometimes we want to be able to run the same test on a data set. Holding the data set in a Collection and iterating over it with the assertion in the loop body has the problem that the first assertion failure will stop the test execution. JUnit 5 offers multiple ways to overcome this limitation.

5.10.1. Using Dynamic Tests

JUnit 5 offers the possibility to define dynamic tests. We can use this to rewrite our example. Dynamic test methods are annotated with `@TestFactory`. They can return the following datastructure with the `DynamicTest` type:

- an Iterable
- a Collection
- a Stream

JUnit then runs all dynamic tests during test execution.

`@BeforeEach` and `@AfterEach` methods will not be called for dynamic tests. This means that you can't use them to reset the test object if you change it's state in the lambda expression for a dynamic test.

In the following example we define a method to return a Stream of `DynamicTest` instances.

```java
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

import java.util.Arrays;
import java.util.stream.Stream;

import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

public class DynamicTestCreationTest {

    @TestFactory
    public Stream<DynamicTest> testDifferentMultiplyOperations() {
        MyClass tester = new MyClass();
        int[][] data = new int[][] { { 1, 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.stream(data).map(entry -> {
            int m1 = entry[0];
            int m2 = entry[1];
            int expected = entry[2];
            return dynamicTest(m1 + " * " + m2 + " = " + expected, () -> {
                assertEquals(expected, tester.multiply(m1, m2));
            });
        });
    }

    // class to be tested
    class MyClass {
        public int multiply(int i, int j) {
            return i * j;
        }
    }
}
```

5.10.2. Using Parameterized Tests

Junit5 also supports parameterized tests. To use them you have to add the `junit-jupiter-params` package as a test dependencies.

▶ Adding `junit-jupiter-params` dependency for a Maven build

▶ Adding `junit-jupiter-params` dependency for a Gradle build

For this example we use the `@MethodSource` annotation.

We give it the name of the function(s) we want it to call to get it's test data. The function has to be static and must return either a Collection, an Iterator, a Stream or an Array. On execution the test method gets called once for every entry in the data source. In contrast to Dynamic Tests `@BeforeEach` and `@AfterEach` methods will be called for parameterized tests.

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

public class DynamicTestCreationTest {

    public static int[][] data() {
        return new int[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
    }

    @ParameterizedTest
    @MethodSource(value =  "data")
    void testWithStringParameter(int[] data) {
        MyClass tester = new MyClass();
        int m1 = data[0];
        int m2 = data[1];
        int expected = data[2];
        assertEquals(expected, tester.multiply(m1, m2));
    }

    // class to be tested
    class MyClass {
        public int multiply(int i, int j) {
            return i * j;
        }
    }
}
```

Data sources

The following table gives an overview of all possible test data sources for parameterized tests.

Table 2. Table Parameterized Tests Data Sources

Annotation	Description

Annotation	Description
JAVA `@ValueSource(ints = { 1, 2, 3 })`	Lets you define an array of test values. Permissible types are `String`, `int`, `long`, or `double`.
JAVA `@EnumSource(value = Months.class, names = {"JANUARY", "FEBRUARY"})`	Lets you pass Enum constants as test class. With the optional attribute `names` you can choose which constants should be used. Otherwise all attributes are used.
JAVA `@MethodSource(names = "genTestData")`	The result of the named method is passed as argument to the test.
JAVA `@CsvSource({ "foo, 1", "'baz, qux', 3" })` `void testMethod(String first, int second) {`	Expects strings to be parsed as Csv. The delimiter is `','`.
JAVA `@ArgumentsSource(MyArgumentsProvider.class)`	Specifies a class that provides the test data. The referenced class has to implement the `ArgumentsProvider` interface.

Argument conversion

JUnit tries to automatically convert the source strings to match the expected arguments of the test method.

If you need explicit conversion you can specify a converter with the `@ConvertWith` annotation. To define your own converter you have to implement the `ArgumentConverter` interface. In the following example we use the abstract `SimpleArgumentConverter` base class.

```java
@ParameterizedTest
@ValueSource(ints = {1, 12, 42})
void testWithExplicitArgumentConversion(@ConvertWith(ToOctalStringArgumentConverter.class) String
argument) {
    System.err.println(argument);
    assertNotNull(argument);
}

static class ToOctalStringArgumentConverter extends SimpleArgumentConverter {
    @Override
    protected Object convert(Object source, Class<?> targetType) {
        assertEquals(Integer.class, source.getClass(), "Can only convert from Integers.");
        assertEquals(String.class, targetType, "Can only convert to String");
        return Integer.toOctalString((Integer) source);
    }
}
```

5.11. JUnit naming conventions for Maven

If you are using the Maven build system, all classes following the following pattern are automatically included into the test run:

```
**/Test*.java          1
**/*Test.java          2
**/*Tests.java         3
**/*TestCase.java      4
```

1. includes all of its subdirectories and all Java filenames that start with "Test".
2. includes all of its subdirectories and all Java filenames that end with "Test".
3. includes all of its subdirectories and all Java filenames that end with "Tests".
4. includes all of its subdirectories and all Java filenames that end with "TestCase".

5.12. Where should the test be located?

Typical, unit tests are created in a separate source folder to keep the test code separate from the real code. The standard convention from the Maven and Gradle build tools is to use:

- src/main/java - for Java classes
- src/test/java - for test classes

6. Eclipse support for JUnit

6.1. Creating JUnit tests

You can write the JUnit tests manually, but Eclipse supports the creation of JUnit tests via wizards.

You can select the class which you want to test in the editor, can press Ctrl+1 and select `Create new JUnit test`.

Alternatively, you can right-click on your class, select this class in the *Project Explorer* or *Package Explorer* view, right-click on it and select New › JUnit Test Case. Or, you can also use the JUnit wizards available under File › New › Other… › Java › JUnit.

6.2. Running JUnit tests

The Eclipse IDE also provides support for executing your tests interactively.

To run a test, select the test class, right-click on it and select Run-as › JUnit Test. This starts JUnit and executes all test methods in this class.

Eclipse provides the `Alt`+`Shift`+`X`, `T` shortcut to run the test in the selected class.

To run only the selected test, position the cursor on the test method name and use the shortcut.

To see the result of a JUnit test, Eclipse uses the *JUnit* view which shows the results of the tests. You can also select individual unit tests in this view, right-click on them and select *Run* to execute them again.

Eclipse creates run configurations for tests. You can see and modify these via the Run › Run Configurations… menu.

6.3. Compare results

For some assertions, you can open a dialog to compare the expected and the actual value. Right-click on the test result, and select Compare Result or double-click on the line.

6.4. Show only failed tests

By default, this view shows all tests. You can also configure, that it only shows failing tests.

You can also define that the view is only activated if you have a failing test.

6.5. Extracting the failed tests and stacktraces

To get the list of failed tests, right click on the tests result and select *Copy Failure List*. This copies the failed tests and the stack traces into the clipboard.

6.6. Using code minings

The Eclipse IDE supports adding additional information to text editors to enrich the editor content without modifying the original source. This can be especially useful for unit tests, as you can activate the method parameters.

You can activate that via Windows › Preference › Java › Editor › Code Minings.

6.7. Setting Eclipse up for using JUnits static imports

The Eclipse IDE cannot always create the corresponding `static import` statements automatically.

You can configure the Eclipse IDE to use code completion to insert typical JUnit method calls and to add the static import automatically. For this open the Preferences via Window › Preferences and select Java › Editor › Content Assist › Favorites.

Use the `New Type` button to add the following entries to it:

- `org.junit.Assert`

- `org.hamcrest.CoreMatchers`
- `org.hamcrest.Matchers`

This makes, for example, the `assertTrue`, `assertFalse` and `assertEquals` methods directly available in the *Content Assists*.

You can now use *Content Assists* (shortcut: `Ctrl`+`Space`) to add the method and the import.

6.8. Wizard for creating test suites

You can create a test suite via Eclipse. For this, select the test classes which should be included in suite in the *Package Explorer* view, right-click on them and select New > Other... > JUnit > JUnit Test Suite.

6.9. JUnit Plug-in Test

JUnit Plug-in tests are used to write unit tests for your plug-ins. These tests are executed by a special test runner that launches another Eclipse instance in a separate VM. The test methods are executed within that instance.

7. Additional information about JUnit 5 usage

7.1. Test execution order

JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests.

As of JUnit 4.11 the default is to use a deterministic, but not predictable, order for the execution of the tests.

You can use an annotation to define that the test methods are sorted by method name, in lexicographic order. To activate this feature, annotate your test class with the `@FixMethodOrder(MethodSorters.NAME_ASCENDING)` annotation. You can also explicitly set the default by using the `MethodSorters.DEFAULT` parameter in this annotation. You can also use `MethodSorters.JVM` which uses the JVM defaults, which may vary from run to run.

7.2. Using the @TempDir annotation to create temporary files and paths

The `@TempDir` annotations allows to annotate non-private fields or method parameters in a test method of type `Path` or `File. JUnit 5 has registered a `ParameterResolutionException` for this annotation and will create temporary files and paths for the tests. It will also remove the temporary files are each test.

```java
    @Test
    @DisplayName("Ensure that two temporary directories with same files names and content have same
hash")
    void hashTwoDynamicDirectoryWhichHaveSameContent(@TempDir Path tempDir, @TempDir Path tempDir2)
throws IOException {

        Path file1 = tempDir.resolve("myfile.txt");

        List<String> input = Arrays.asList("input1", "input2", "input3");
        Files.write(file1, input);

        assertTrue(Files.exists(file1), "File should exist");

        Path file2 = tempDir2.resolve("myfile.txt");

        Files.write(file2, input);
        assertTrue(Files.exists(file2), "File should exist");

    }
```

7.3. Mocking frameworks

Mocking frameworks help to mock classes requried for a test. See Mockito tutorial (https://www.vogella.com/tutorials/Mockito/article.html).

7.4. Assert frameworks

While JUnit provides simple asserts for common cases, additional frameworks provide even more options.

See Hamcrest Tutorial (https://www.vogella.com/tutorials/Hamcrest/article.html) and AssertJ Tutorial (https://www.vogella.com/tutorials/AssertJ/article.html) for introductions into these frameworks.

7.5. Test Suites

Currently, JUnit 5 does not support natively the grouping of tests. See https://github.com/junit-team/junit5/pull/2416 for the work in progress.

8. Exercise: Writing a JUnit 5 test with Maven and Eclipse in 5 mins

8.1. Project creation

Create a new Maven project with the File > New > Other > Maven > Maven Project entry.

Select to create a simple project (skip archetype selection).

Name the artifact *com.vogella.junit.first*.

8.2. Package creation

Create a package named *com.vogella.junit.first* in the *src/main/java* and *src/main/test* folder.

8.3. Update your pom file to use JUnit

Add the following to your pom file, after the groupId, artifactID and version entry:

```
JAVA
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <!--1 -->
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>
        </plugins>
    </build>

    <!--2 -->
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.7.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.7.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

8.4. Update Maven

Right-click your pom file, select Maven › Update Project and select your project. This triggers an update of your project settings and dependencies.

8.5. Create a Java class

In the *src* folder, create the following class in the `com.vogella.junit.first` package.

```java
package com.vogella.junit.first;

public class MyClass {
    // the following is just an example
  public int multiply(int x, int y) {
    if (x > 999) {
      throw new IllegalArgumentException("X should be less than 1000");
    }
    return x / y;
  }
}
```

8.6. Create a JUnit test

Position the cursor on the `MyClass` in the Java editor and press Ctrl+1. Select that you want to create a new JUnit test from the list.

Alternatively you can right-click on your new class in the :_Project_Explorer_ or *Package Explorer* view and select New > Other > Java > JUnit Test Case.

In the following wizard ensure that the *New JUnit Jupiter test* flag is selected. The source folder should select the *test* directory.

Press the `Next` button and select the methods that you want to test.

Create a test with the following code.

```java
package com.vogella.junit.first;

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

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class MyClassTest {

    @Test
    public void testExceptionIsThrown() {
        MyClass tester = new MyClass();
        assertThrows(IllegalArgumentException.class, () -> tester.multiply(1000, 5));
    }

    @Test
    public void testMultiply() {
        MyClass tester = new MyClass();
        assertEquals(50, tester.multiply(10, 5), "10 x 5 must be 50");
    }
}
```

8.7. Run your test in Eclipse

Right-click on your new test class and select Run-As › JUnit Test.

The result of the tests are displayed in the JUnit view. In our example one test should be successful and one test should show an error. This error is indicated by a red bar.

You discovered a bug in the tested code!

8.8. Fix the bug and re-run your tests

The test is failing, because our multiplier class is currently not working correctly. It does a division instead of multiplication. Fix the bug and re-run the test to get a green bar.

8.9. Conclusion

After a few minutes you should have created a new project, a new class and a new unit test. Congratulations! If you feel like it, lets improve the tests a bit and write one grouped test.

9. Optional Exercise: Cleaning up tests and writing a grouped test

9.1. Improve tests

The initialization of `MyClass` happens in every test, move the initialization to a `@BeforeEach` method.

▶ Solution

9.2. Define a group check with assertAll

Define a new test method which checks both condition at the same time with `assertAll` statement. Change the condition to make both tests fail, run the test and ensure that both are executed.

▶ Solution

10. Comparison of annotations between JUnit 4 and 5

JUnit 4 is the main release version before JUnit 5. In the section we list the different kind of annotations side by side so that you can compare them.

JUnit 5	JUnit 4	Description
`import org.junit.jupiter.api.*`	`import org.junit.*`	Import statement for using the following annotations.
`@Test`	`@Test`	Identifies a method as a test method.
`@BeforeEach`	`@Before`	Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).

JUnit 5	JUnit 4	Description
`@AfterEach`	`@After`	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
`@BeforeAll`	`@BeforeClass`	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
`@AfterAll`	`@AfterClass`	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
`@Disabled` or `@Disabled("Why disabled")`	`@Ignore` or `@Ignore("Why disabled")`	Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.
Not available, is replaced by `org.junit.jupiter.api.Assertions.expectThrows()`	`@Test (expected = Exception.class)`	Fails if the method does not throw the named exception.
Not available, is replaced by `AssertTimeout.assertTimeout()` and `AssertTimeout.assertTimeoutPreemptively()`	`@Test(timeout=100)`	Fails if the method takes longer than 100 milliseconds.