Junit

Basic terms:

Test Fixtures: It is a fixed state of a set of objects used as a baseline for running Tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are applicable. (Loading local Database, preparation of input data..)

Unit test, test coverage, unit testing, integration tests.

Behaviour testing: validating inputs and parameters

State Testing: validating results

Following the convention of Maven and Gradle: **src / test / java** should contain the test classes

Using JUnit

- JUnit is a **test** framework which uses annotations to identify methods that specify a test.
- JUnit *test* is a method contained in Test class which is only used for testing.
- To define a certain method is a test method we annotate it with @Test annotation.
- We can use assert statements with meaningful messages which makes it easier for the user to identify and fix the problem.
- **MyClasstest.java**

- While using maven we should use Test suffix for test classes as it automatically includes such classes in its scope.
- The org.junit.runner.JUnitCore class provides the runClasses() method. This method allows you to run one or several tests classes. As a return parameter you receive an object of the type org.junit.runner.Result. This object can be used to retrieve information about the tests.
- For ex: **MyTestRunner.java**

- 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. However, we can annotate the test class with @FixMethodOrder(MethodSorters.NAME_ASCENDING) so they are sorted by method name in lexicographic order.

Annotations:

Several Annotations can be used on methods. List is given below:

- @Test identifies a method as a test method
- @Before/BeforeEach Executed before each test
- @After / AfterEach Executed after each test
- @BeforeClass / BeforeAll Executed once before the start of all tests
- @AfterClass / AfterAll Executed once after all tests have been finished
- @Ignore / Disabled Marks the test should be disabled
- @Tag("<TagName>") *Tests in JUnit 5 can be filtered by tag
- @RepeadedTest(<Number>) *Repeat the test a <Number> of times

Assert Statements:

- JUnit provides static methods to test for certain conditions via the Assert class. These *assert statements* typically start with assert. They allow you to specify the error message, the expected and the actual result.
- An *assertion method* compares the actual value returned by a test to the expected value. It throws an **AssertionException** if the comparison fails
  - fail(["String"])
  - assertTrue ()
  - assertFalse
  - assertEquals..

JUnit Test Suites

- If you have several test classes, you can combine them into a test suite. Running a test suite executes all test classes in that suite in the specified order. A test suite can also contain other test suites.
- If you want to add another test class, you can add it to the @Suite.SuiteClasses statement. Ex: **AllTest.java**

** The @Ignore annotation allow to statically ignore a test. Alternatively you can use Assume.assumeFalse or Assume.assumeTrue to define a condition for the test. Assume.assumeFalse marks the test as invalid, if its condition evaluates to true. Assume.assumeTrue evaluates the test as invalid if its condition evaluates to false.

Parameterized Test

- JUnit allows you to use parameters in a test class. This class can contain one test method and this method is executed with different parameters provided.
- It helps developers save time when executing the same tests which only differ in their inputs and expected outputs.
- We use @RunWith(Parameterized.class) annotation.
- https://www.logicbig.com/tutorials/unit-testing/junit/runner.html

- Test Class must contain a static method annotated with the @Parameters annotation that returns a collection of objects as a test dataset.
- We can use either @Parameter annotation on public fields to get the test values in the test **or** we can use a constructor to initialize the values to store values for each test.
- Ex: **Parameterized***

JUnit Rules

- Through JUnit rules you can add behaviour to each test in a class.
- We can annotate fields of type TestRule(It is an interface and the types like ExpectedException and TemporaryFolder implements this interface) with the @Rule annotation.
- We could, for example, specify which exception message you expect during the execution of your test code. Ex: **RuleExceptionTesterExample.java**
- https://junit.org/junit4/javadoc/4.12/org/junit/rules/ExpectedException.html
- We can also write a custom rule by implementing the TestRule interface. This interface defines the apply method i.e. apply(Statement, Description) which must return an instance of Statement.
    - Statement represents the tests within the JUnit runtime and Statement evaluate() run these.
    - Description describes the individual test.

Categories

- We can also define categories of tests and include or exclude them based on annotations
- We use @Category() to mark a particular test into the particular category.
- We use @IncludeCategory(), @ExcludeCategory() to include nd exclude categories into a particular test suite.

```
public interface FastTests { /* category marker */
}

public interface SlowTests { /* category marker */
}

public class A {
  @Test
  public void a() {
    fail();
  }

  @Category(SlowTests.class)
  @Test
  public void b() {
  }
}

@Category({ SlowTests.class, FastTests.class })
public class B {
  @Test
```

```
    public void c() {
    }
}

@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@SuiteClasses({ A.class, B.class })
// Note that Categories is a kind of Suite
public class SlowTestSuite {
    // Will run A.b and B.c, but not A.a
}

@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@ExcludeCategory(FastTests.class)
@SuiteClasses({ A.class, B.class })
// Note that Categories is a kind of Suite
public class SlowTestSuite {
    // Will run A.b, but not A.a or B.c
}
```