**What is Test Driven Development (TDD)?**

* TDD Stands for Test Driven Development.
* One reason to write the tests first is to have a better understanding of the actual code before you write it. This is the main benefit of test driven development. When you write the test cases first, you think more critically about the corner cases. It's then easier to address them when you write the code and ensure that they're accurate.
* TDD starts with designing and developing tests for every small functionality of an application. In TDD approach, first, the test is developed which specifies and validates what the code will do.

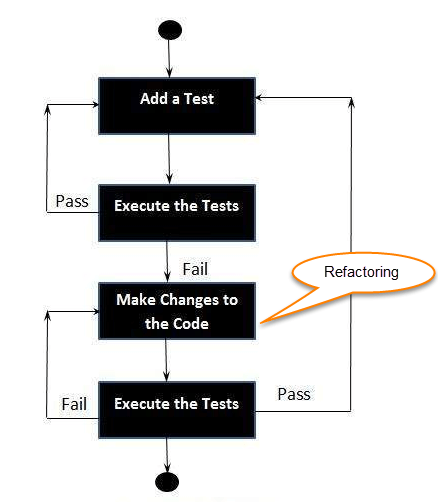
**Note:**

In the normal Software Testing process, we first generate the code and then test. But it TDD the tests are written first before the actual code. In Short, Test-Driven development is a process of developing and running automated test before actual development of the application.

**How to perform TDD Test**

Following steps define how to perform TDD test,

1. Add a test.
2. Run all tests and see if any new test fails.
3. Write some code.
4. Run tests and Refactor code.
5. Repeat.



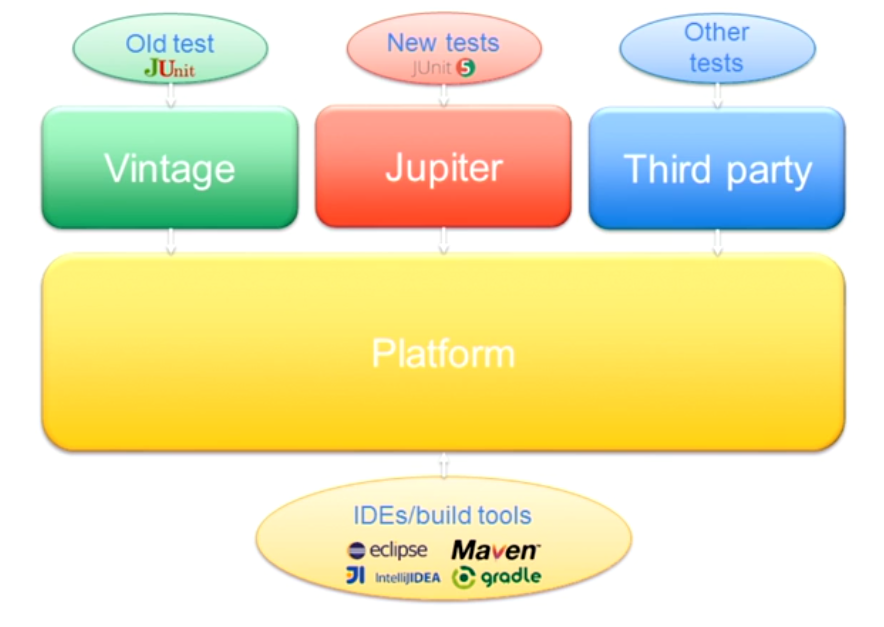
**What is Unit Testing?**

**UNIT TESTING** is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/ super class, abstract class or derived/ child class.

**Why Unit Tests Are Important?**

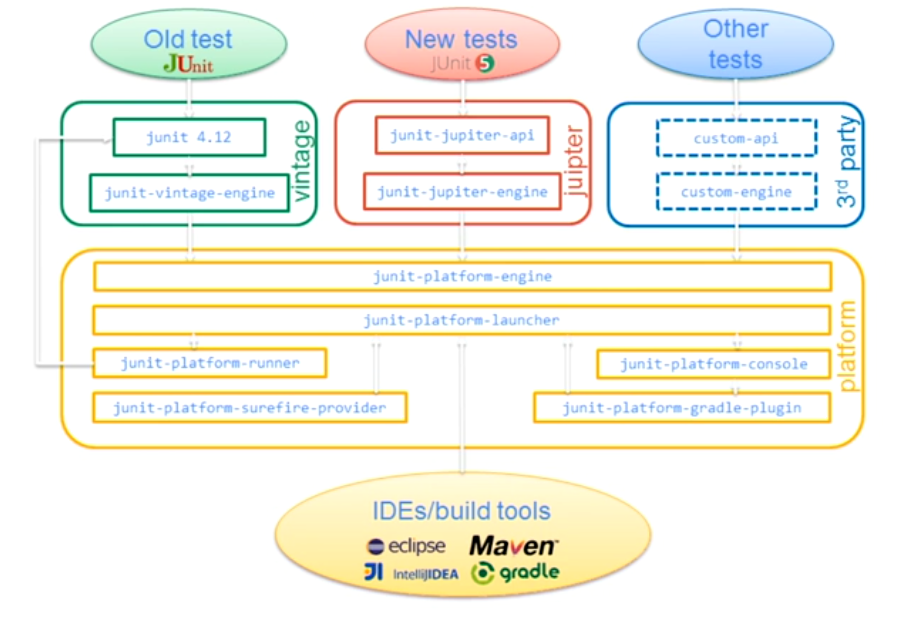
Developers should be focused on unit testing before the deployment of product so that the testers can test builds on time. When developers do not pay attention to unit testing and hand over the build to the tester for testing, this situation produces lot of issues for tester.

**JUnit Architecture**

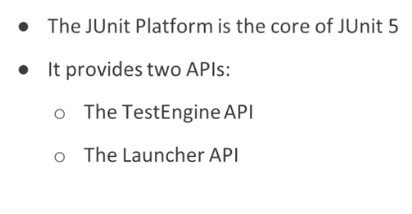
****

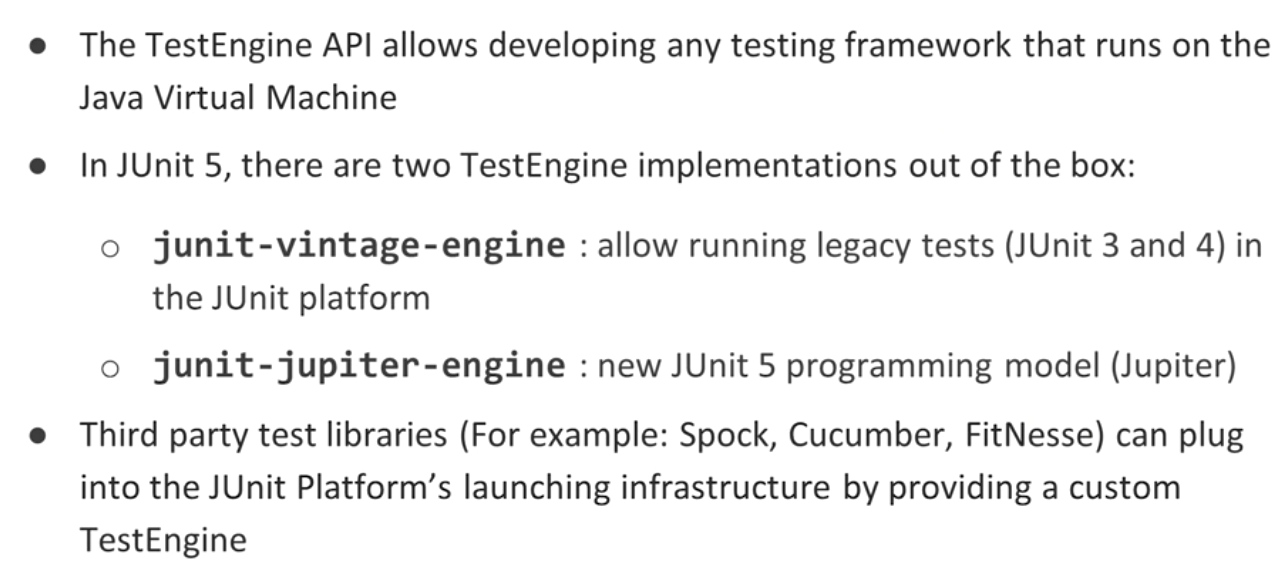
* The JUnit framework is composed by three major components, Platform, Jupiter, and Vintage.
* In the core of JUnit 5, we find the **Platform**. This component is completely new, and it is aimed to be the foundation for any testing framework that is going to be executed in the Java virtual machine. For legacy JUnit 4 tests, to third party tests.
* The component Jupiter provides the brand new programming, and extension model of JUnit 5.
* On the other hand, the Vintage component allows running all tests based on version 3 and 4 of JUnit.
* Finally, third party frameworks can produce the JUnit Platform to include new testing in JUnit. For example, Cucumber, Spock, Weakness and so on.

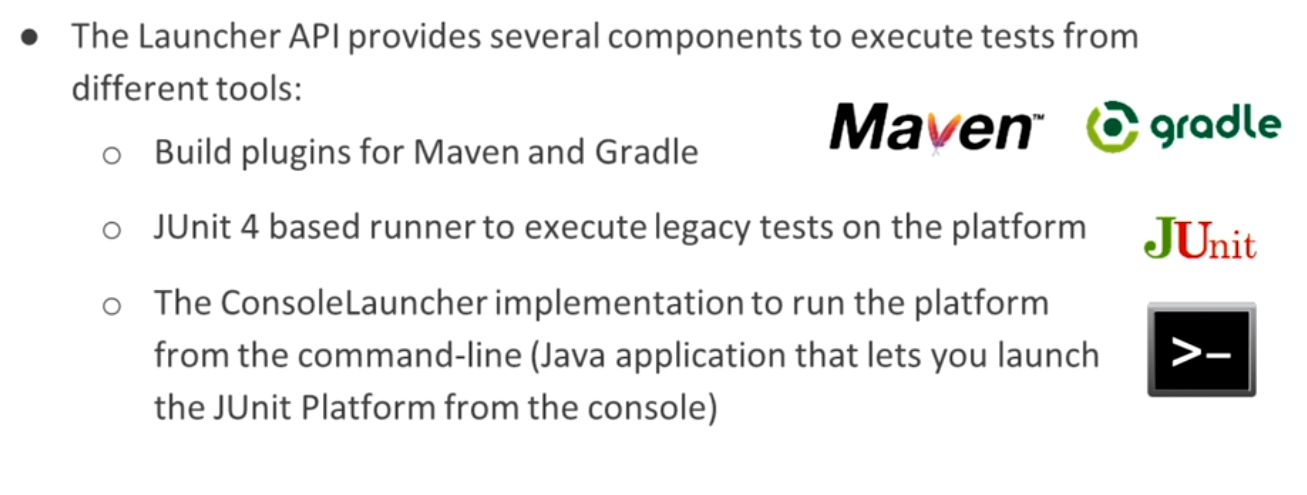
**Let's take a closer look to the internal details of each component.**

****

****

****

****

****

**Setting up JUnit with Maven**

**pom.xml**

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.dev.test</groupId>

<artifactId>MyJunit5</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>MyJunit5</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<junit.jupiter.version>5.4.1</junit.jupiter.version>

</properties>

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>${junit.jupiter.version}</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.0</version>

</plugin>

<!-- JUnit 5 requires Surefire version 2.22.1 or higher -->

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.1</version>

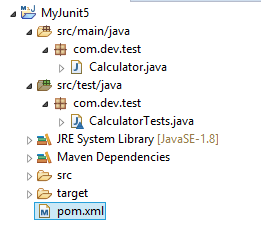
</plugin>

</plugins>

</build>

</project>

**Sample Project Structure**

****

***Calculator.java***

**package** com.dev.test;

**public** **class** Calculator {

**public** **int** add(**int** a, **int** b) {

**return** a + b;

}

}

***CalculatorTests.java***

**package** com.dev.test;

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

**import** org.junit.jupiter.api.DisplayName;

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

**import** org.junit.jupiter.params.ParameterizedTest;

**import** org.junit.jupiter.params.provider.CsvSource;

**class** CalculatorTests {

@Test

@DisplayName("1 + 1 = 2")

**void** addsTwoNumbers() {

Calculator calculator = **new** Calculator();

assertEquals(2, calculator.add(1, 1), "1 + 1 should equal 2");

}

@ParameterizedTest(name = "{0} + {1} = {2}")

@CsvSource({

"0, 1, 1",

"1, 2, 3",

"49, 51, 100",

"1, 100, 101"

})

**void** add(**int** first, **int** second, **int** expectedResult) {

Calculator calculator = **new** Calculator();

assertEquals(expectedResult, calculator.add(first, second),

() -> first + " + " + second + " should equal " + expectedResult);

}

**Lifecycle Methods**

In JUnit 5, test lifecycle is driven by 4 primary annotations i.e. [@BeforeAll](https://howtodoinjava.com/junit-5/before-all-annotation-example/), [@BeforeEach](https://howtodoinjava.com/junit-5/before-each-annotation-example/), [@AfterEach](https://howtodoinjava.com/junit-5/after-each-annotation-example/) and [@AfterAll](https://howtodoinjava.com/junit-5/after-all-annotation-example/). Along with it, each test method must be marked with @Test annotation.

 *@BeforeEach –* denotes that the annotated method will be executed before each test method (previously *@Before*)

 *@AfterEach* – denotes that the annotated method will be executed after each test method (previously *@After*)

 *@BeforeAll* – denotes that the annotated method will be executed before all test methods in the current class (previously *@BeforeClass*)

 *@AfterAll* – denotes that the annotated method will be executed after all test methods in the current class (previously *@AfterClass*)

| **Annotation** | **Description** |
| --- | --- |
| @Test | Denotes that a method is a test method. Unlike JUnit 4’s @Test annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are *inherited* unless they are *overridden*. |
| @ParameterizedTest | Denotes that a method is a [parameterized test](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests). Such methods are *inherited* unless they are *overridden*. |
| @RepeatedTest | Denotes that a method is a test template for a [repeated test](https://junit.org/junit5/docs/current/user-guide/#writing-tests-repeated-tests). Such methods are *inherited* unless they are *overridden*. |
| @TestFactory | Denotes that a method is a test factory for [dynamic tests](https://junit.org/junit5/docs/current/user-guide/#writing-tests-dynamic-tests). Such methods are *inherited* unless they are *overridden*. |
| @TestTemplate | Denotes that a method is a [template for test cases](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-templates) designed to be invoked multiple times depending on the number of invocation contexts returned by the registered [providers](https://junit.org/junit5/docs/current/user-guide/#extensions-test-templates). Such methods are *inherited* unless they are *overridden*. |
| @TestMethodOrder | Used to configure the [test method execution order](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order) for the annotated test class; similar to JUnit 4’s @FixMethodOrder. Such annotations are *inherited*. |
| @TestInstance | Used to configure the [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) for the annotated test class. Such annotations are *inherited*. |
| @DisplayName | Declares a custom [display name](https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-names) for the test class or test method. Such annotations are not *inherited*. |
| @DisplayNameGeneration | Declares a custom [display name generator](https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-name-generator) for the test class. Such annotations are *inherited*. |
| @BeforeEach | Denotes that the annotated method should be executed *before* **each** @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4’s @Before. Such methods are *inherited* unless they are *overridden*. |
| @AfterEach | Denotes that the annotated method should be executed *after* **each** @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4’s @After. Such methods are *inherited* unless they are *overridden*. |
| @BeforeAll | Denotes that the annotated method should be executed *before* **all** @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4’s @BeforeClass. Such methods are *inherited* (unless they are *hidden* or *overridden*) and must be static (unless the "per-class" [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) is used). |
| @AfterAll | Denotes that the annotated method should be executed *after* **all** @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4’s @AfterClass. Such methods are *inherited* (unless they are *hidden* or *overridden*) and must be static (unless the "per-class" [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) is used). |
| @Nested | Denotes that the annotated class is a non-static [nested test class](https://junit.org/junit5/docs/current/user-guide/#writing-tests-nested). @BeforeAll and @AfterAll methods cannot be used directly in a @Nested test class unless the "per-class" [test instance lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle) is used. Such annotations are not *inherited*. |
| @Tag | Used to declare [tags for filtering tests](https://junit.org/junit5/docs/current/user-guide/#writing-tests-tagging-and-filtering), either at the class or method level; analogous to test groups in TestNG or Categories in JUnit 4. Such annotations are *inherited* at the class level but not at the method level. |
| @Disabled | Used to [disable](https://junit.org/junit5/docs/current/user-guide/#writing-tests-disabling) a test class or test method; analogous to JUnit 4’s @Ignore. Such annotations are not *inherited*. |
| @ExtendWith | Used to [register extensions declaratively](https://junit.org/junit5/docs/current/user-guide/#extensions-registration-declarative). Such annotations are *inherited*. |
| @RegisterExtension | Used to [register extensions programmatically](https://junit.org/junit5/docs/current/user-guide/#extensions-registration-programmatic) via fields. Such fields are *inherited* unless they are *shadowed*. |
| @TempDir | Used to supply a [temporary directory](https://junit.org/junit5/docs/current/user-guide/#writing-tests-built-in-extensions-TempDirectory) via field injection or parameter injection in a lifecycle method or test method; located in the org.junit.jupiter.api.io package. |