Contents

[JUNIT 5 2](#_Toc83067220)

[ARCHITECTURE 2](#_Toc83067221)

[PLATFORM 2](#_Toc83067222)

[JUPITER 2](#_Toc83067223)

[VINTAGE 2](#_Toc83067224)

[EXTERNAL (3RD PARTY) 2](#_Toc83067225)

[SAMPLE POM.XML/ DEPENDENCIES 3](#_Toc83067226)

[MAVEN SUREFIRE PLUGIN - INTEGRATION 4](#_Toc83067227)

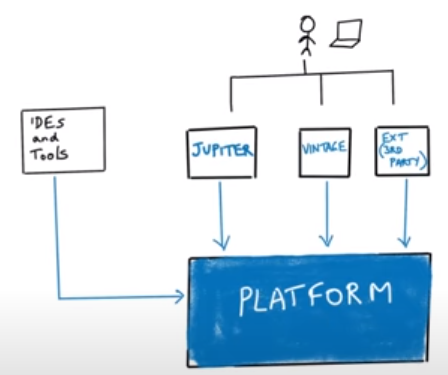
[JUNIT TEST LIFE CYCLE 5](#_Toc83067228)

[LIFE CYCLE HOOKS 6](#_Toc83067229)

[TAGGING A TEST 6](#_Toc83067230)

# JUNIT 5

## ARCHITECTURE



### PLATFORM

* Test engine of Junit which contains core the library of Junit
* Developer while writing the Junit - don’t interact with the platform

### JUPITER

* This is where Junit API resides
* Developer code interact with Jupiter which in turn interact with Junit Platform. Example @Test , Asset – they belong to Jupiter.

### VINTAGE

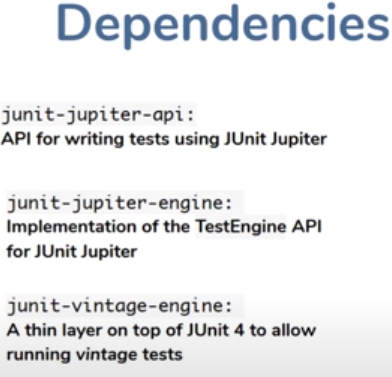
* Junit -5 is not backward compatible so to support the test cases written in the legacy version of Junit we need an extra dependency called vintage.

### EXTERNAL (3RD PARTY)

Note:

* Developer writing the test case on Junit-5 will use Jupiter and the Junit test case written in V4 will use Vintage.
* From maven standpoint, for Junit -5 standpoint we need to add dependencies for,
  + Dependencies for core junit platform
  + Dependencies for Jupiter
  + If we have Junit -4 as well we need vintage dependencies too.

## SAMPLE POM.XML/ DEPENDENCIES



|  |  |
| --- | --- |
| <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>io.javabrains</groupId>  <artifactId>junit-5-basics</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>junit-5-basics</name>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>${maven.compiler.source}</maven.compiler.target>  <junit.jupiter.version>5.5.2</junit.jupiter.version> 🡨 Jupiter Dependency  <junit.platform.version>1.5.2</junit.platform.version>🡨 Junit Platform Dependency  </properties>  <dependencies>  <dependency>  <groupId>**org.junit.jupiter**</groupId>  <artifactId>junit-jupiter-engine</artifactId>  <version>${junit.jupiter.version}</version>  <scope>test</scope> 🡨 This make sure that unit jars are not part of the final build  </dependency>  <dependency>  <groupId>**org.junit.platform**</groupId>  <artifactId>junit-platform-runner</artifactId>  <version>${junit.platform.version}</version>  <scope>test</scope>  </dependency>  </dependencies>  </project > | |
|  | **PROJECT HEIRARACHY**   * **MathUtils.java – Java Class under test**   public class MathUtils {  public int add(int a, int b) {  return a + b;  }  }   * **MathUtilsTest.java – Junit Test class**   import static org.junit.jupiter.api.Assertions.\*;  import org.junit.jupiter.api.Test;  class MathUtilsTest {  @Test  void test() {  MathUtils mathUtils = new MathUtils();  int expected = 2;  int actual = mathUtils.add(1, 1);  assertEquals(expected, actual);  }  }  **RUN THE TEST CASE**   * **Right click of Test class 🡪 Run As 🡪 Junit test case** |

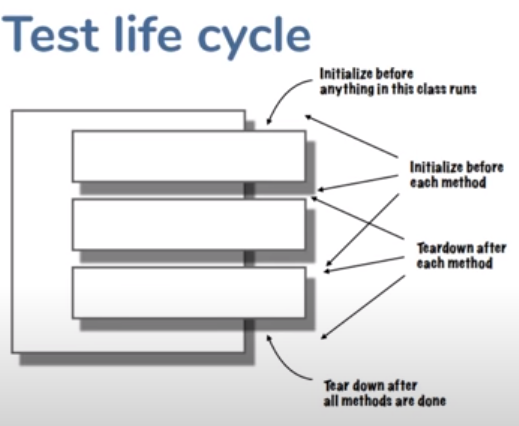
* All the Junit test case are marked with @Test Annotation.
* Junit User Guide : <https://junit.org/junit5/docs/current/user-guide/>

## MAVEN SUREFIRE PLUGIN - INTEGRATION

* Till now we were executing the unit test case using IDE, but what if we want to run the test cases in CI / CD pipeline. The Maven surefire plugin help is achieving the same.

|  |
| --- |
| **ENABLING SUREFIRE PLUG-IN** |
| <project>  .....  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.22.1</version>  </plugin>  </plugins>  </build>  ...  </project>  **RUNINNG THE PROJECT WITH SUREFIRE PLUGIN**   * Right Click on project 🡪 Run As 🡪 Maven 🡪 Maven test |

## JUNIT TEST LIFE CYCLE



* It’s a bad practice to use a shared instance variable across multiple test cases.

|  |  |
| --- | --- |
| TEST CLASS – WITHOUT LIFE CYCLE HOOK | |
| **import** **static** org.junit.jupiter.api.Assertions.\*;  **import** org.junit.jupiter.api.Test;  **class** MathUtilsTest {  @Test  **void** testAdd() {  MathUtils mathUtils = **new** MathUtils();  **int** expected = 2;  **int** actual = mathUtils.add(1, 1);  *assertEquals*(expected, actual);  }  @Test  **void** testSubstract() {  MathUtils mathUtils = **new** MathUtils();  **int** expected = 2;  **int** actual = mathUtils.substract(4, 2);  *assertEquals*(expected, actual);  }  } | * When we run the Junit, test cases all the test case marked with @Test annotation runs in a random order. * To execute the test cases (test function), **the test engine creates an exclusive object of test class for each test function and start calling the test functions on that object** * In the test functions written here is creating an object of class under test(MathUtils.java). This common code is getting executed for each test function. * This can be achieved in more cleaner way using life cycle hooks. |

### LIFE CYCLE HOOKS

|  |  |  |
| --- | --- | --- |
| **LIFE CYCLE HOOKS** | **USABILITY** | |
| **@BeforeAll** | The method annotated with this annotation executed before any test case run | |
| **@AfterAll** | The method annotated with this annotation executed after all test case finish execution | |
| **@BeforeEach** | The method annotated with this annotation executed before each test case run | |
| **@AfterEach** | The method annotated with this annotation executed after each test case run | |
| **TEST CLASS – WITH LIFE CYCLE HOOK** | | |
| **import** **static** org.junit.jupiter.api.Assertions.\*;  **import** org.junit.jupiter.api.BeforeEach;  **import** org.junit.jupiter.api.Test;  **class** MathUtilsTest {  MathUtils mathUtils = **null**;  @BeforeEach  **public** **void** init() {  mathUtils = **new** MathUtils();  }  @Test  **void** testAdd() {  **int** expected = 2;  **int** actual = mathUtils.add(1, 1);  *assertEquals*(expected, actual);  }  @Test  **void** testSubstract() {  **int** expected = 2;  **int** actual = mathUtils.substract(4, 2);  *assertEquals*(expected, actual);  }  } | | * The init() will be executed right before every test case |

### TAGGING A TEST

* Tagging a test helps in managing the test and running the test selectively. Junit uses @Tag to do so.

TAGGING A TEST

TO RUN THE SELECTIVE TEST IN ECLIPSE