de-facto Unit Testing framework for Android App

https://junit.org/junit5/docs/current/user-guide/

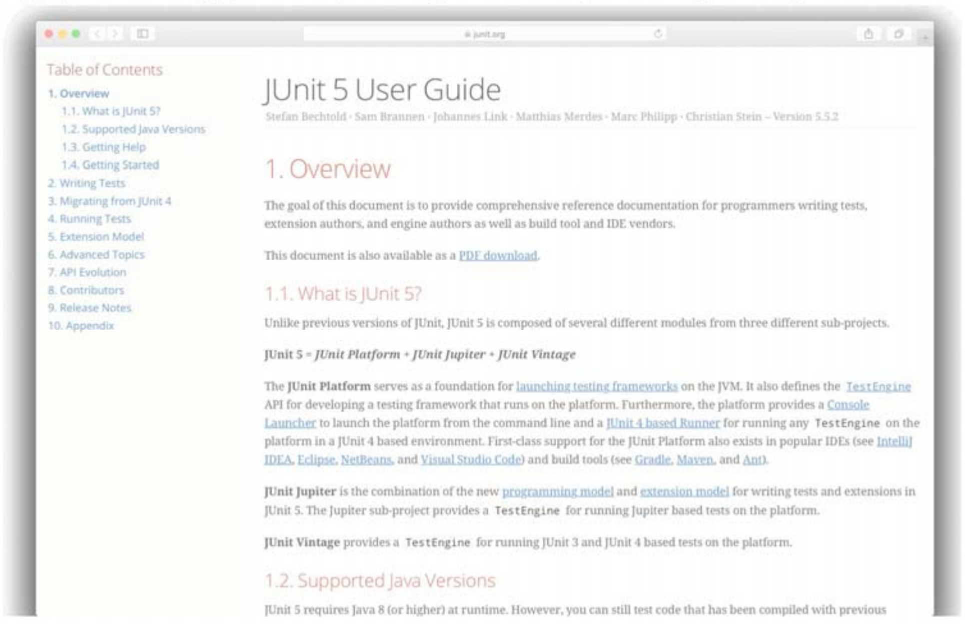

junit.org

Table of Contents

JUnit 5 User Guide

Stefan Bechtold · Sam Brannen · Johannes Link · Matthias Merdes · Marc Philipp · Christian Stein – Version 5.5.2

1. Overview

The goal of this document is to provide comprehensive reference documentation for programmers writing tests, extension authors, and engine authors as well as build tool and IDE vendors.

This document is also available as a PDF download.

1.1. What is JUnit 5?

Unlike previous versions of JUnit, JUnit 5 is composed of several different modules from three different sub-projects.

JUnit 5 = ***JUnit Platform*** + ***JUnit Jupiter*** + ***JUnit Vintage***

The **JUnit Platform** serves as a foundation for launching testing frameworks on the JVM. It also defines the `TestEngine` API for developing a testing framework that runs on the platform. Furthermore, the platform provides a Console Launcher to launch the platform from the command line and a JUnit 4 based Runner for running any `TestEngine` on the platform in a JUnit 4 based environment. First-class support for the JUnit Platform also exists in popular IDEs (see IntelliJ IDEA, Eclipse, NetBeans, and Visual Studio Code) and build tools (see Gradle, Maven, and Ant).

JUnit Jupiter is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a `TestEngine` for running Jupiter based tests on the platform.

JUnit Vintage provides a `TestEngine` for running JUnit 3 and JUnit 4 based tests on the platform.

1.2. Supported Java Versions

JUnit 5 requires Java 8 (or higher) at runtime. However, you can still test code that has been compiled with previous

JUnit

- Kent Beck and Erich Gamma developed a unit testing framework for Java programs called **JUnit**.

  https://junit.org/junit5/

- JUnit 4.0 introduced annotations in the **org.junit** package for marking test code.
  - **@Test, @Before**, **@After**, **@BeforeClass**, **@AfterClass**, **@Ignore**, **@Test** etc.
- In 2017, JUnit 5 was announced.

Terminology

- A **unit test** is a test of a *single* class (in general)
- A **test fixture** is a fixed state of a set of objects used as a baseline for running tests.
  - The purpose is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.
- A **test case** tests the response of a single method to a particular set of inputs.
- A **test suite** is a collection of test cases.

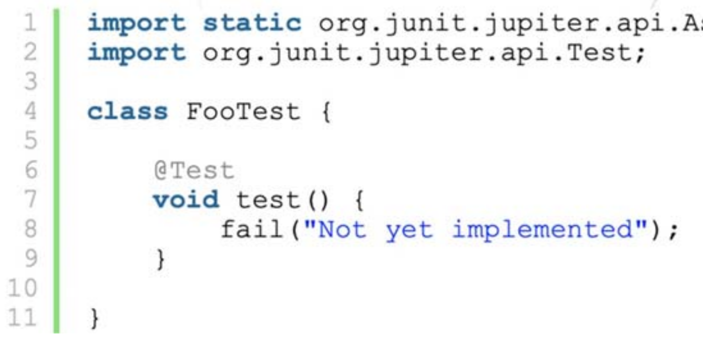
Structure of a JUnit test class

- To test a class named **Foo**
- Create a test class **FooTest**

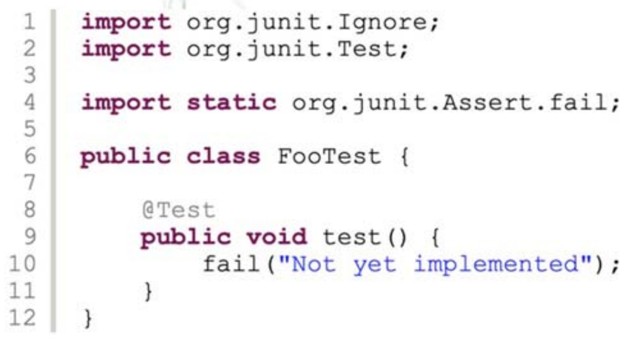
```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class FooTest {

        @Test
        void test() {
                fail("Not yet implemented");
        }

}
```

Structure of a JUnit test class

- To test a class named **Foo**, create a test class **FooTest**



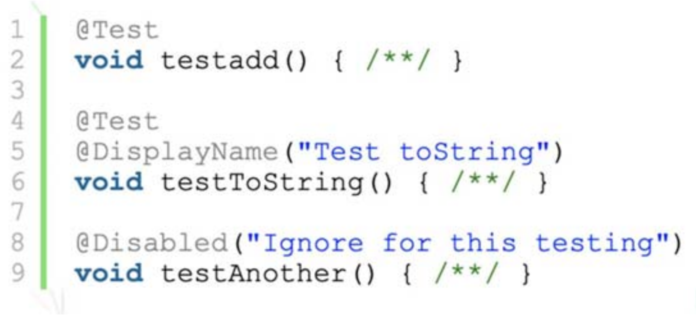
```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class FooTest {

    @Test
    void test() {
        fail("Not yet implemented");
    }

}
```



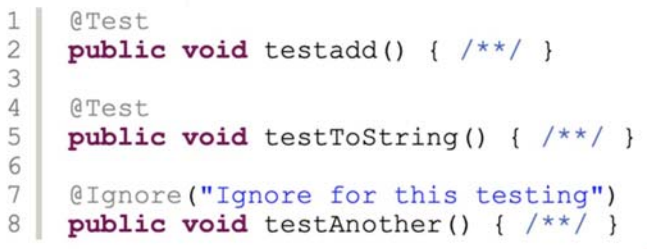
```
import org.junit.Ignore;
import org.junit.Test;

import static org.junit.Assert.fail;

public class FooTest {

    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```

Test Cases

- Methods annotated with **@Test** are test cases:
  - Their order of execution is not specified



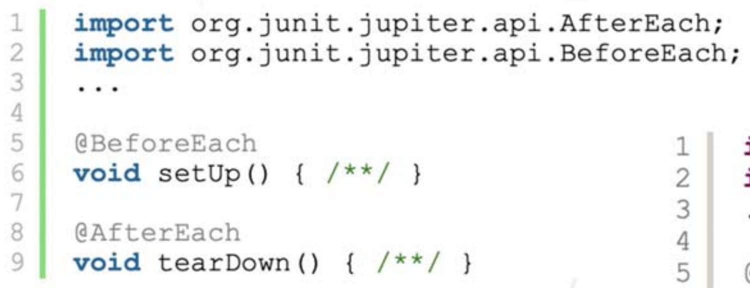
```
@Test
void testadd() { /**/ }

@Test
@DisplayName("Test toString")
void testToString() { /**/ }

@Disabled("Ignore for this testing")
void testAnother() { /**/ }
```



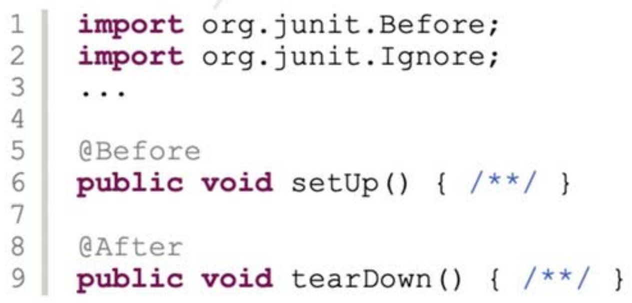
```
@Test
public void testadd() { /**/ }

@Test
public void testToString() { /**/ }

@Ignore("Ignore for this testing")
public void testAnother() { /**/ }
```

Test Fixtures

- Test cases with **@BeforeEach** will execute before every test case.
- Test cases with **@AfterEach** will execute after every test case



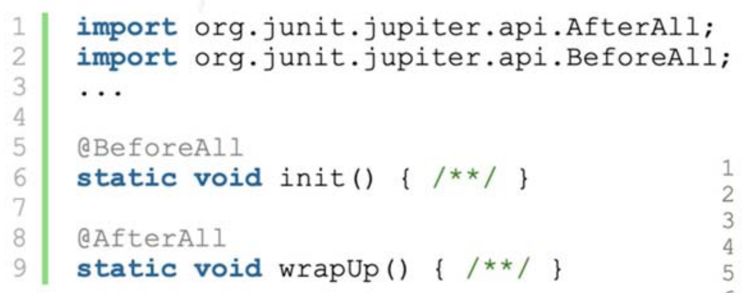
```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
...

@BeforeEach
void setUp() { /**/ }

@AfterEach
void tearDown() { /**/ }
```



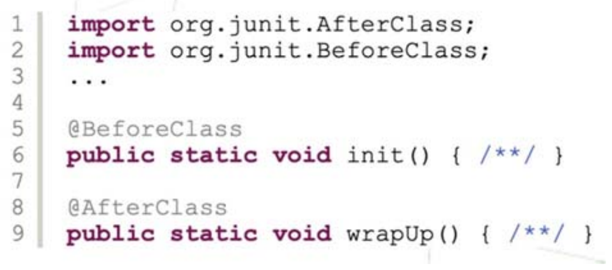
```
import org.junit.Before;
import org.junit.Ignore;
...

@Before
public void setUp() { /**/ }

@After
public void tearDown() { /**/ }
```

Class Test fixtures

- Test cases with **@BeforeAll** will execute once *before* all test cases.
- Test cases with **@AfterAll** will execute once *after* all test cases.
  - Useful to allocate and release expensive resources once



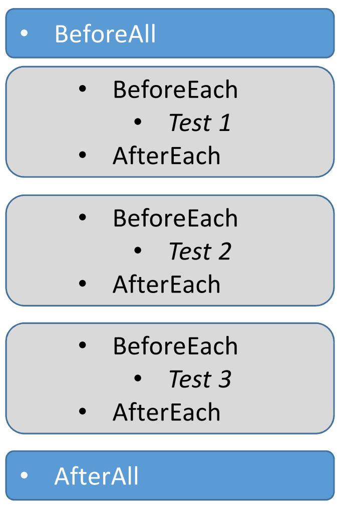
```java
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
...

@BeforeAll
static void init() { /**/ }

@AfterAll
static void wrapUp() { /**/ }
```



```java
import org.junit.AfterClass;
import org.junit.BeforeClass;
...

@BeforeClass
public static void init() { /**/ }

@AfterClass
public static void wrapUp() { /**/ }
```

What JUnit does

- For *each* test case **t**:
  - JUnit executes all **@BeforeEach** methods
  - JUnit executes **t**
    - Any exceptions during its execution are logged
  - JUnit executes all **@AfterEach** methods
- Report for all test cases is presented

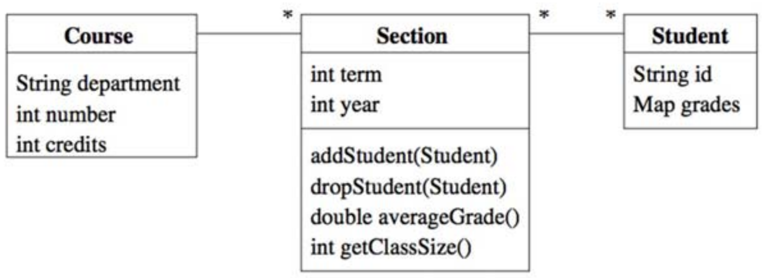


Within a test case

- Call the methods of the class being tested.
- Assert what the correct result should be with one of the provided **assert** methods.
  - `assertEquals(expected, actual);`
- These steps can be repeated as many times as necessary.
- An **assert** method is a **JUnit** method that performs a test, and throws an **AssertionError** if the test fails.
  - **JUnit** catches these exceptions and shows you the results.

Example Classes

- To demonstrate writing unit tests, we are going to develop some classes for modeling **Student**s that are enrolled in a **Section** of a **Course**.



Writing a simple test case

```
class SectionTest {
  @Test
  void testAddStudent() {

    Student student = new Student("123-45-6789");
    Course course = new Course("CS", 410, 4);
    Section section =
      new Section(course, Section.SPRING, 2001);

    section.addStudent(student);

    assertEquals(1, section.getClassSize());
  }
}
```

The left class tests that adding a **Student** increases the enrollment by one

Given

When

Then

The **assertEquals** method is imported from the **Assertions** class. If its arguments are not equal, then the test fails.

Testing Error Conditions

```
@Test
void testDropStudentNotEnrolled() {
    Student student = new Student("123-45-6789");
    Course course = new Course("CS", 410, 4);
    Section section =
      new Section(course, Section.SPRING, 2001);

    assertThrows(IllegalArgumentException.class,
      () -> section.dropStudent(student));
}
```

- Making sure that your program fails in a well-understood fashion is very important.
- To test that the **dropStudent** method throws an **IllegalArgumentException**

Testing Error Conditions (JUnit 4)

```
@Test(expected = IllegalArgumentException.class)
  public void testDropStudentNotEnrolled() {
    Student student = new Student("123-45-6789");
    Course course = new Course("CS", 410, 4);
    Section section =
      new Section(course, Section.SPRING, 2001);

    section.dropStudent(student);
}
```

The Assertions class

- The **Assertions** contains methods for validating that certain conditions are true.
  - **assertEquals**: Two entities (objects, ints, etc.) should be equal
    - (compares objects using **equals()**)
  - **assertNotNull**: A value should not be null
  - **assertSame**: Two object references should be the same
    - (compare objects using ==)
  - **assertTrue**: A boolean expression should be true
  - **assertFalse**: A boolean expression should be false
  - **fail**: The test should fail

The Assertions class

- When an assertion evaluates to false, the test fails.
- Each **assert** method is overloaded to have a **String** message.

```java
assertEquals(1, section.getClassSize());
             ↑              ↑
          expected        actual

assertEquals(
  1, section.getClassSize(), "Wrong number of students"
);
```

How to organize Tests (JUnit Java files)

- The better way is to place the tests in a separate parallel directory structure with package alignment.

```
─── pom.xml
└─── src
     ├── main
     │    ├── java
     │    │    └── com
     │    │         └── javadevelopersguide
     │    │              └── junit
     │    │                   └── Calculator.java
     │    ├──── resources
     └── test
          ├── java
          │    └── com
          │         └── javadevelopersguide
          │              └── junit
          │                   └──── CalculatorTest.java
          └── resources
```

More readable assertions

JUnit provides some basic methods for validating the state of your tests (assertions), but the code and the failure messages can be hard to read

```
assertTrue(myString.contains("Hello"));
```

When the above fails, all you get is an "*expected true, but got false*" error message.

The **Hamcrest** assertion framework provides powerful "matchers" that provide readable assertion statements with detailed and specific failure messages:

http://hamcrest.org/JavaHamcrest

Hamcrest assertion statements

Hamcrest provides an **assertThat** method that asserts that some value "matches" a "matcher" .

Each "matcher" has a static factory method.

Matchers are composed to form complex assertions.

The matcher is syntactic sugar that aids readability.

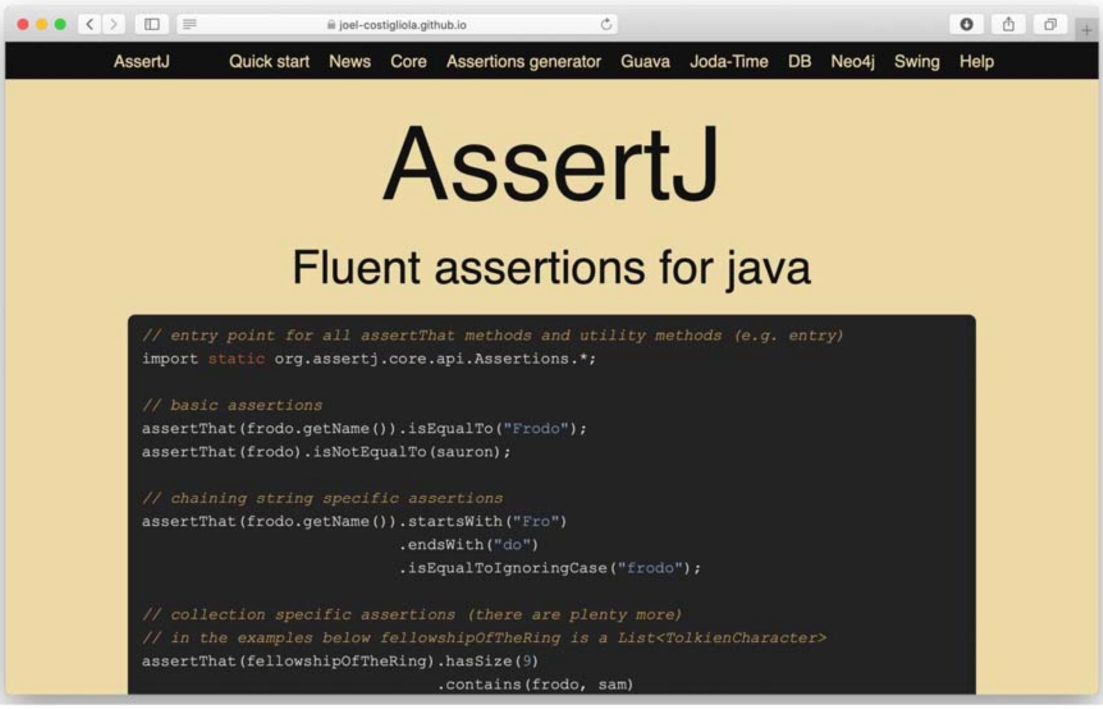
```
import org.junit.Jupiter.api.Test;
import static org.hamcrest.Matchers.*;
import static org.hamcrest.MatcherAssert.assertThat;
class HamcrestMatchersTest {

  @Test
  void isEqualTo() {
    Integer int1 = new Integer("123");
    Integer int2 = new Integer("123");

    assertThat(int1, is(equalTo(int2)));
  }
```

Examples of Hamcrest assertions

```
@Test
void isNullValue() {
  assertThat(null, is(nullValue()));
}

@Test
void isSameInstance() {
  Object o = new Object();
  assertThat(o, is(sameInstance(o)));
}

@Test
public strings() {
String s = "Hamcrest is awesome";

  assertThat(s, startsWith("Hamcrest"));
  assertThat(s, endsWith("awesome"));
  assertThat(s, containsString("is"));
  assertThat(s, is(not(isEmptyString())));
  assertThat(s, is(equalToIgnoringCase("HAMCREST IS AWESOME")));
}
```

Alternative to Hamcrest framework
https://assertj.github.io/doc/

AssertJ
Fluent assertions for java



```
// entry point for all assertThat methods and utility methods (e.g. entry)
import static org.assertj.core.api.Assertions.*;

// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron);

// chaining string specific assertions
assertThat(frodo.getName()).startsWith("Fro")
                           .endsWith("do")
                           .isEqualToIgnoringCase("frodo");

// collection specific assertions (there are plenty more)
// in the examples below fellowshipOfTheRing is a List<TolkienCharacter>
assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
```

Truth - Fluent assertions for Java and Android

https://truth.dev

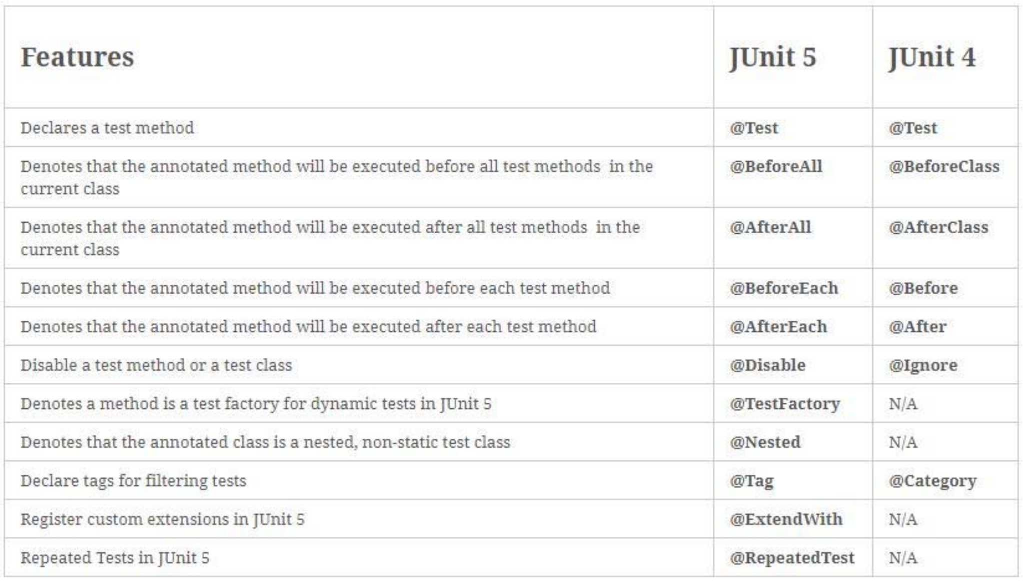
- **Truth** is a library for performing fluent assertions in tests:

```
assertThat(notificationText).contains("testuser@google.com");
```

- Gradle

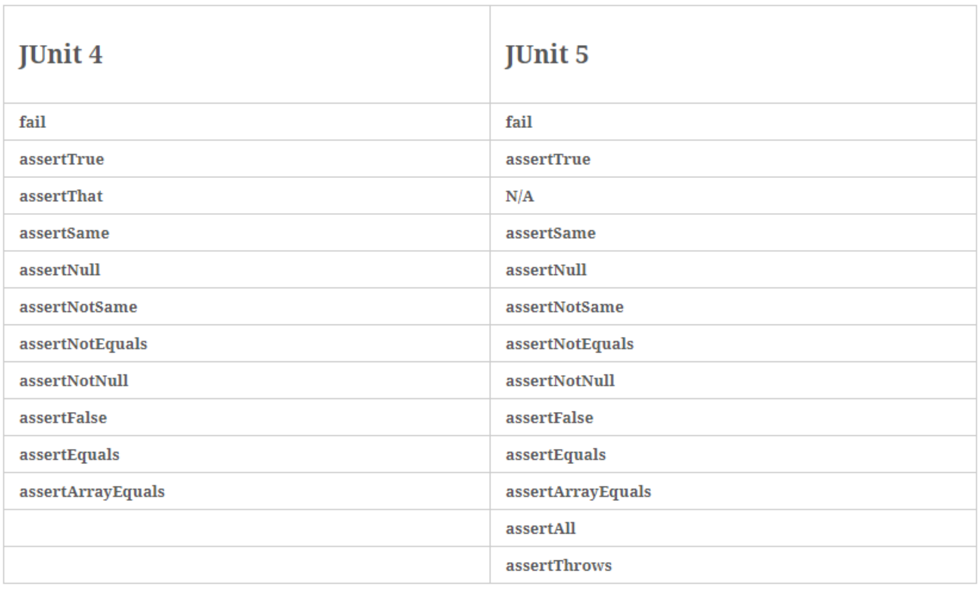
```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation "com.google.truth:truth:1.0"
    testImplementation "com.google.truth.extensions:truth-java8-extension:1.0"
}
```

Annotations



Features	JUnit 5	JUnit 4
Declares a test method	@Test	@Test
Denotes that the annotated method will be executed before all test methods in the current class	@BeforeAll	@BeforeClass
Denotes that the annotated method will be executed after all test methods in the current class	@AfterAll	@AfterClass
Denotes that the annotated method will be executed before each test method	@BeforeEach	@Before
Denotes that the annotated method will be executed after each test method	@AfterEach	@After
Disable a test method or a test class	@Disable	@Ignore
Denotes a method is a test factory for dynamic tests in JUnit 5	@TestFactory	N/A
Denotes that the annotated class is a nested, non-static test class	@Nested	N/A
Declare tags for filtering tests	@Tag	@Category
Register custom extensions in JUnit 5	@ExtendWith	N/A
Repeated Tests in JUnit 5	@RepeatedTest	N/A

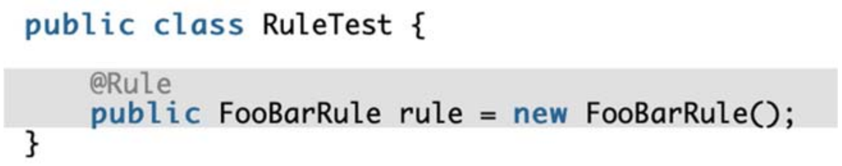
Assertions



JUnit 4	JUnit 5
fail	fail
assertTrue	assertTrue
assertThat	N/A
assertSame	assertSame
assertNull	assertNull
assertNotSame	assertNotSame
assertNotEquals	assertNotEquals
assertNotNull	assertNotNull
assertFalse	assertFalse
assertEquals	assertEquals
assertArrayEquals	assertArrayEquals
	assertAll
	assertThrows

JUnit4 Rule

- A component that intercepts test method calls.
- Allows us to do something
  - *before* a test method is run *and*
  - *after* a test method has been run.
- All JUnit 4 rule classes must implement the `org.junit.rules.TestRule`



```
public class RuleTest {

    @Rule
    public FooBarRule rule = new FooBarRule();
}
```

JUnit 4 requires that rule fields are public, aren't static.