JUnit 5:

All core annotations are located in the org.junit.jupiter.api package in the junit-jupiter-api module.



Let's discuss each annotation with a sample example.

## **@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. Example:

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

import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

@Test

void myFirstTest() {

assertEquals(2, 1 + 1);

}

}

## **@ParameterizedTest**

Parameterized tests make it possible to run a test multiple times with different arguments.They are declared just like regular @Test methods but use the @ParameterizedTest annotation instead. In addition, you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method. Example: The following example demonstrates a parameterized test that uses the @ValueSource annotation to specify a String array as the source of arguments.

@ParameterizedTest

@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })

void palindromes(String candidate) {

assertTrue(isPalindrome(candidate));

}

## **@RepeatedTest**

JUnit Jupiter provides the ability to repeat a test a specified number of times simply by annotating a method with @RepeatedTest and specifying the total number of repetitions desired. Each invocation of a repeated test behaves like the execution of a regular *@Test* method with full support for the same lifecycle callbacks and extensions.

Example: The following example demonstrates how to declare a test named *repeatedTest()* that will be automatically repeated 10 times.

@RepeatedTest(10)

void repeatedTest() {

// ...

}

@RepeatedTest(5)

void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {

assertEquals(5, repetitionInfo.getTotalRepetitions());

}

@RepeatedTest(value = 1, name = "{displayName} {currentRepetition}/{totalRepetitions}")

@DisplayName("Repeat!")

void customDisplayName(TestInfo testInfo) {

assertEquals(testInfo.getDisplayName(), "Repeat! 1/1");

}

[**More detail ...**](https://java-developers-guide.blogspot.com/2018/07/junit-5-repeated-tests-with-examples.html)

## **@DisplayName**

Test classes and test methods can declare custom display names — with spaces, special characters, and even emojis — that will be displayed by test runners and test reporting.

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.Test;

@DisplayName("A special test case")

class DisplayNameDemo {

@Test

@DisplayName("Custom test name containing spaces")

void testWithDisplayNameContainingSpaces() {

}

@Test

@DisplayName("╯°□°）╯")

void testWithDisplayNameContainingSpecialCharacters() {

}

@Test

@DisplayName("😱")

void testWithDisplayNameContainingEmoji() {

}

}

## **@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. Example:

class StandardTests {

@BeforeEach

void init() {

}

@Test

void succeedingTest() {

}

@AfterEach

void tearDown() {

}

}

## **@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. Example:

class StandardTests {

@BeforeEach

void init() {

}

@Test

void succeedingTest() {

}

@AfterEach

void tearDown() {

}

}

## **@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.

class StandardTests {

@BeforeAll

static void initAll() {

}

@Test

void succeedingTest() {

}

@AfterAll

static void tearDownAll() {

}

}

## **@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. Example:

class StandardTests {

@BeforeAll

static void initAll() {

}

@Test

void succeedingTest() {

}

@AfterAll

static void tearDownAll() {

}

}

## **@Nested**

Denotes that the annotated class is a nested, non-static test class. @BeforeAll and @AfterAll methods cannot be used directly in a @Nested test class unless the "per-class" test instance lifecycle is used. Such annotations are not inherited.

Nested tests give the test writer more capabilities to express the relationship among several group of tests. Here’s an elaborate example.

Nested test suite for testing a stack

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

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

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

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

import java.util.EmptyStackException;

import java.util.Stack;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.Nested;

import org.junit.jupiter.api.Test;

@DisplayName("A stack")

class TestingAStackDemo {

Stack<Object> stack;

@Test

@DisplayName("is instantiated with new Stack()")

void isInstantiatedWithNew() {

new Stack<>();

}

@Nested

@DisplayName("when new")

class WhenNew {

@BeforeEach

void createNewStack() {

stack = new Stack<>();

}

@Test

@DisplayName("is empty")

void isEmpty() {

assertTrue(stack.isEmpty());

}

@Test

@DisplayName("throws EmptyStackException when popped")

void throwsExceptionWhenPopped() {

assertThrows(EmptyStackException.class, () -> stack.pop());

}

@Test

@DisplayName("throws EmptyStackException when peeked")

void throwsExceptionWhenPeeked() {

assertThrows(EmptyStackException.class, () -> stack.peek());

}

@Nested

@DisplayName("after pushing an element")

class AfterPushing {

String anElement = "an element";

@BeforeEach

void pushAnElement() {

stack.push(anElement);

}

@Test

@DisplayName("it is no longer empty")

void isNotEmpty() {

assertFalse(stack.isEmpty());

}

@Test

@DisplayName("returns the element when popped and is empty")

void returnElementWhenPopped() {

assertEquals(anElement, stack.pop());

assertTrue(stack.isEmpty());

}

@Test

@DisplayName("returns the element when peeked but remains not empty")

void returnElementWhenPeeked() {

assertEquals(anElement, stack.peek());

assertFalse(stack.isEmpty());

}

}

}

}

## **@Tag**

Used to declare tags for filtering tests, 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. Example:

import org.junit.jupiter.api.Tag;

import org.junit.jupiter.api.Test;

@Tag("fast")

@Tag("model")

class TaggingDemo {

@Test

@Tag("taxes")

void testingTaxCalculation() {

}

}

## **@Disabled**

Used to disable a test class or test method; analogous to JUnit 4’s @Ignore. Such annotations are not inherited. Example: @Disabled used to disable test class.

import org.junit.jupiter.api.Disabled;

import org.junit.jupiter.api.Test;

@Disabled

class DisabledClassDemo {

@Test

void testWillBeSkipped() {

}

}

@Disabled annotation used to disable test method.

import org.junit.jupiter.api.Disabled;

import org.junit.jupiter.api.Test;

class DisabledTestsDemo {

@Disabled

@Test

void testWillBeSkipped() {

}

@Test

void testWillBeExecuted() {

}

}

## **@ExtendWith**

Used to register custom extensions. Such annotations are inherited. Example:For example, to register a custom RandomParametersExtension for a particular test method, you would annotate the test method as follows.

@ExtendWith(RandomParametersExtension.class)

@Test

void test(@Random int i) {

// ...

}