

JUnit 5

JUnit es un framework Java para implementar test en Java. A diferencia de versiones anteriores, JUnit 5 se compone de tres sub-proyectos:

- **JUnit Platform.** Sirve como base sobre la cual se pueden ejecutar diferentes frameworks de pruebas. Es la plataforma que inicia el motor de pruebas y ejecuta las pruebas. A su vez se compone de:
 - **Launcher API:** permite a los clientes (IDEs, herramientas de construcción, etcétera...) descubrir y ejecutar pruebas.
 - **Test Engine API:** permite a los desarrolladores crear sus propios motores de pruebas que se pueden integrar con la plataforma. Por ejemplo se puede utilizar para ejecutar pruebas escritas en Spock, Cucumber, FitNesse, etcétera..., siempre y cuando haya un motor de pruebas compatible.
 - **Console Launcher:** una herramienta de línea de comandos para ejecutar pruebas.
- **JUnit Jupiter:** es la combinación del nuevo modelo de programación y la implementación del motor de pruebas para JUnit 5.
- **JUnit Vintage:** proporciona compatibilidad con versiones anteriores, permitiendo que las pruebas escritas con JUnit 3 y JUnit 4 se ejecuten en JUnit 5. Requiere que JUnit 4.12 esté presente en el class path o el module path.

JUnit 5 requiere **Java 8 (o superior)**.

[Más información](#)

Writing Tests

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

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

class ExampleTest {

    @BeforeAll
    static void setupAll() {
        System.out.println("Ejecutado una vez antes de todas las pruebas.");
    }

    @BeforeEach
    void setup() {
        System.out.println("Ejecutado antes de cada prueba.");
    }

    @Test
    void testAddition() {
        assertEquals(2, 1 + 1, "1 + 1 debe ser igual a 2");
    }

    @Test
    void testSubtraction() {
        assertEquals(0, 1 - 1, "1 - 1 debe ser igual a 0");
    }

    @AfterEach
    void tearDown() {
        System.out.println("Ejecutado después de cada prueba.");
    }

    @AfterAll
```

```

static void tearDownAll() {
    System.out.println("Ejecutado una vez después de todas las pruebas.");
}
}

```

Annotations

JUnit se basa en [anotaciones](#). Casi todas las anotaciones están en el paquete `org.junit.jupiter.api` en el módulo *'junit-jupiter-api'*:

- `@Test` : indica que el método que la contiene es un test.
- `@ParameterizedTest` : indica que ese método es un [test parametrizado](#).
- `@RepeatedTest` : indica que ese método es una plantilla para un [test repetible](#).
- `@Before` (JUnit4) / `@BeforeEach` (JUnit5): indica que ese método anotado debe ser ejecutado **antes** que cada método anotado como `@Test` , `RepeatedTest` , `@ParameterizedTest` o `@TestFactory` .
- `@After` (JUnit4) / `@AfterEach` (JUnit5): indica que ese método anotado debe ser ejecutado **después** que cada método anotado como `@Test` , `RepeatedTest` , `@ParameterizedTest` o `@TestFactory` .
- `@BeforeClass` (JUnit4) / `@BeforeAll` (JUnit5): indica que ese método anotado debe ser ejecutado **antes** que cualquier otro método anotado como `@Test` , `RepeatedTest` , `@ParameterizedTest` o `@TestFactory` .
- `@AfterClass` (JUnit4) / `@AfterAll` (JUnit5): indica que ese método anotado debe ser ejecutado **después** de todos los métodos anotado como `@Test` , `RepeatedTest` , `@ParameterizedTest` o `@TestFactory` .
- `@Ignore` (JUnit 4) / `@Disabled` (JUnit5): se utiliza para [deshabilitar el test](#).
- `@Tag` : se utiliza para declarar [etiquetas](#) que permitan filtrar por tests.
- `@Timeout` : se utiliza para fallar un test, una factoría de test, una plantilla de test o ciclo de vida si su ejecución excede una duración determinada.
- `@ExtendWith` : se utiliza para [registrar](#) extensiones de forma declarativa.
- `@DisplayName("cadena")` (JUnit5): Declara un nombre de visualización personalizado para la clase de prueba o el método de prueba. En lugar de usar esta anotación es recomendable utilizar nombres para los métodos lo suficientemente descriptivos como para que no sea necesario usar esta anotación.

Meta-Annotations and Composed Annotations

Las anotaciones de JUnit Jupiter se pueden utilizar como meta-anotaciones. Eso significa que se pueden definir anotaciones compuestas personalizadas que heredarán automáticamente la semántica de las meta-anotaciones.

Por ejemplo, en vez de copiar y pegar la anotación `@Tag("fast")` (que permite etiquetar y agrupar pruebas), es posible crear una anotación personalizada como por ejemplo `@Fast` :

```

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })

```

```
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
public @interface Fast { }
```

Esta anotación sería equivalente y podría utilizarse cualquiera de las dos anotaciones:

```
@Fast
@Test
void myFastTest() {
    // ...
}

@Tag("fast")
@Test
void otherFastTest() {
    // ...
}
```

Incluso se podría ir un paso más allá introduciendo una anotación `@FastTest` personalizada que se puede usar como reemplazo directo de `@Tag("fast")` y `@Test` :

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
@Test
public @interface FastTest { }
```

Definitions

- **Container:** un nodo en el árbol de prueba que contiene otros contenedores o pruebas como sus hijos (por ejemplo, una clase de prueba).
- **Test:** un nodo en el árbol de prueba que verifica el comportamiento esperado cuando se ejecuta (por ejemplo, un método `@Test`).
- **Lifecycle Method:** cualquier método que esté directamente anotado o meta-anotado con `@BeforeAll` , `@AfterAll` , `@BeforeEach` o `@AfterEach` .
- **Test Class:** cualquier clase de nivel superior, clase miembro estática o clase `@Nested` que contenga al menos un método de prueba, es decir, un contenedor. Las clases de prueba no deben ser abstractas y deben tener un único constructor.
- **Test Method:** cualquier método de instancia que esté directamente anotado o meta- anotado con `@Test` , `@RepeatedTest` , `@ParameterizedTest` , `@TestFactory` o `@TestTemplate` . Con la excepción de `@Test` , estos crean un contenedor en el árbol de pruebas que agrupa pruebas o, potencialmente (para `@TestFactory`), otros contenedores.

Test Classes and Methods

Los métodos de prueba y los métodos de ciclo de vida pueden ser declarados localmente dentro de la clase de prueba actual, heredados de superclases o heredados de [interfaces](#). Además, los métodos de prueba y los métodos de ciclo de vida **no deben**

ser abstractos y no deben retornar un valor (excepto los métodos `@TestFactory`, que deben retornar un valor).

No es necesario que las clases de prueba, los métodos de prueba y los métodos de ciclo de vida sean `public`, pero **no deben ser privados**.

Generalmente se recomienda omitir el modificador `public` para clases de prueba, métodos de prueba y métodos de ciclo de vida a menos que exista una razón técnica para hacerlo.

Una clase estándar de prueba con todos los métodos del ciclo de vida:

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

Display Names

Las clases de prueba y los métodos de prueba pueden declarar nombres personalizados mediante `@DisplayName`, con espacios, caracteres especiales e incluso emojis, que se mostrarán en los informes de prueba y por los ejecutores de prueba y los IDE:

```
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("J o _ o J")
    void testWithDisplayNameContainingSpecialCharacters() {
        // ...
    }

    @Test
    @DisplayName("👹")
    void testWithDisplayNameContainingEmoji() {
        // ...
    }
}
```

Display Name Generators

JUnit Jupiter admite generadores de nombres personalizados que se pueden configurar mediante la anotación

`@DisplayNameGeneration`. Los valores proporcionados a través de anotaciones `@DisplayName` siempre tienen prioridad sobre los nombres generados por `@DisplayNameGenerator`.

Esta anotación se utiliza para configurar una estrategia global de generación de nombres de visualización (*'display names'*) para todas las pruebas en una clase de prueba. Esto es útil cuando se requiere aplicar una convención de nombres de manera consistente sin tener que especificar `@DisplayName` individualmente en cada método de prueba.

Assertions

JUnit Jupiter incluye muchos de los métodos de aserción que tiene JUnit 4 y añade algunos que se prestan bien para ser utilizados con lambdas de Java 8. Todas las aserciones de JUnit Jupiter son métodos estáticos en la clase [org.junit.jupiter.api.Assertions](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api.Assertions).

Algunos de las aserciones más comunes son los siguientes:

- `assertEquals(expected, actual, message)`
- `assertNotEquals(unexpected, actual, message)`
- `assertTrue(condition, message)`
- `assertFalse(condition, message)`
- `assertNull(object, message)`
- `assertNotNull(object, message)`
- `assertThrows(expectedType, executable)`
- `assertAll(executables...)`

```

import org.junit.jupiter.api.*;

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

class CommonAssertionsTest {

    @Test
    @DisplayName("Prueba de assertEquals y assertNotEquals")
    void testEquals() {
        assertEquals(2, 1 + 1, "1 + 1 debe ser igual a 2");
        assertNotEquals(3, 1 + 1, "1 + 1 no debe ser igual a 3");
    }

    @Test
    @DisplayName("Prueba de assertTrue y assertFalse")
    void testTrueFalse() {
        assertTrue(5 > 1, "5 debe ser mayor que 1");
        assertFalse(5 < 1, "5 no debe ser menor que 1");
    }

    @Test
    @DisplayName("Prueba de assertNull y assertNotNull")
    void testNull() {
        String nullString = null;
        String nonNullString = "JUnit";
        assertNull(nullString, "El objeto debe ser nulo");
        assertNotNull(nonNullString, "El objeto no debe ser nulo");
    }

    @Test
    @DisplayName("Prueba de assertThrows")
    void testThrows() {
        assertThrows(ArithmeticException.class, () -> {
            int result = 1 / 0;
        }, "Debe lanzar ArithmeticException");
    }

    @Test
    @DisplayName("Prueba de assertAll")
    void testAll() {
        assertAll("Pruebas de suma",
            () -> assertEquals(4, 2 + 2, "2 + 2 debe ser igual a 4"),
            () -> assertTrue(4 > 0, "4 debe ser mayor que 0"),
            () -> assertFalse(4 < 0, "4 no debe ser menor que 0")
        );
    }
}

```

Kotlin Assertion Support

JUnit Jupiter también viene con algunos métodos de aserción que se prestan bien para ser utilizados en Kotlin. Todas las aserciones de JUnit Jupiter en Kotlin son funciones de nivel superior en el paquete *'org.junit.jupiter.api'*.

Third-party Assertion Libraries

Aunque las facilidades de aserción proporcionadas por JUnit Jupiter son suficientes para muchos escenarios de prueba, hay ocasiones en las que se desea o se requiere más potencia y funcionalidades adicionales, como los *'matchers'*.

En tales casos, el equipo de JUnit recomienda el uso de bibliotecas de aserción de terceros como [AssertJ](#), [Hamcrest](#), [Truth](#), etc....:

```

import org.junit.jupiter.api.Test;

import java.util.Arrays;

```

```
import java.util.List;

import static org.assertj.core.api.Assertions.*;

public class ExampleTest {

    @Test
    void testWithAssertJ() {
        // Given
        List<Integer> numbers = Arrays.asList(1, 2, 3);

        // Assertions with AssertJ
        assertThat(numbers).hasSize(3)
            .contains(2)
            .doesNotContain(4);
    }
}
```

Assumptions

TODO

Disabling Tests

TODO

Conditional Test Execution

TODO

Tagging and Filtering

Las clases y métodos de prueba se pueden **etiquetar** via la anotación `@Tag`. Esto permite agrupar tests bajo ciertas etiquetas, facilitando la selección y ejecución de subconjuntos específicos de tests según sea necesario.

En JUnit 4, estas etiquetas se mapean a `@Category`.

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("fast")
@Tag("model")
class TaggingDemo {

    @Test
    @Tag("taxes")
    void testingTaxCalculation() {
    }

}
```

La sintaxis de las etiquetas debe seguir ciertas [reglas](#).

Test Execution Order

TODO

Test Instance Lifecycle

TODO

Nested Tests

TODO

Dependency Injection for Constructors and Methods

TODO

Test Interfaces and Default Methods

TODO

Enlaces de interés

- [JUnit 5](#)
- <https://github.com/junit-team/junit5-samples>
- <https://www.baeldung.com/junit>
- <https://www.tutorialspoint.com/junit/index.htm>
- <https://www.vogella.com/tutorials/JUnit/article.html>

Licencia



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](#).