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

```java
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 estándard 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:

```java
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("ﾉ°□°) ﾉ")
    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.

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…)**

```java
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 asercioness 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...:

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

import java.util.Arrays;
```

```java
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

Los *"assumptions"* (suposiciones) en JUnit se utilizan para realizar **pruebas condicionales**. Permiten que una prueba se ejecute solo si se cumplen ciertas condiciones. Si la condición no se cumple, la prueba se marca como *"skipped"* (omitida) en lugar de *"failed"* (fallida). Esto es útil para situaciones donde la prueba no es relevante o no puede ejecutarse en ciertos entornos o configuraciones.

JUnit Jupiter viene con un subconjunto de los métodos de suposición que JUnit 4 proporciona y añade algunos que se prestan bien para ser utilizados con expresiones lambda y referencias de métodos de Java 8. Todas las suposiciones de JUnit Jupiter son métodos estáticos en la clase `org.junit.jupiter.api.Assumptions`.

Algunas de las más comunes serían:

- `Assumptions.assumeTrue(condition)`: la prueba se ejecuta solo si la condición es verdadera.
- `Assumptions.assumeFalse(condition)`: la prueba se ejecuta solo si la condición es falsa.
- `Assumptions.assumeTrue(condition, "Message if condition is false")`: la prueba se ejecuta solo si la condición es verdadera, con un mensaje si no se cumple.
- `assumingThat(boolean condition, Executable executable)`: ejecuta un bloque de código solo si la condición es verdadera.

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class AssumptionsDemo {

    private final Calculator calculator = new Calculator();

    @Test
    void testOnlyOnCiServer() {
        assumeTrue("CI".equals(System.getenv("ENV")));
        // remainder of test
    }

    @Test
    void testOnlyOnDeveloperWorkstation() {
        assumeTrue("DEV".equals(System.getenv("ENV")),
            () -> "Aborting test: not on developer workstation");
        // remainder of test
```

```
    }

    @Test
    void testInAllEnvironments() {
        assumingThat("CI".equals(System.getenv("ENV")),
            () -> {
                // perform these assertions only on the CI server
                assertEquals(2, calculator.divide(4, 2));
            });

        // perform these assertions in all environments
        assertEquals(42, calculator.multiply(6, 7));
    }

}
```

Disabling Tests

Se pueden deshabilitar clases de prueba completas o métodos de prueba individuales mediante la anotación `@Disabled`.

Esta anotación puede declararse sin proporcionar un motivo; sin embargo, el equipo de JUnit recomienda que los desarrolladores proporcionen una breve explicación de por qué se ha deshabilitado una clase o método de prueba.

Clase de prueba deshabilitada:

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

@Disabled("Disabled until bug #99 has been fixed")
class DisabledClassDemo {

    @Test
    void testWillBeSkipped() {
    }

}
```

Método de prueba deshabilitado en una clase de prueba:

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

class DisabledTestsDemo {

    @Disabled("Disabled until bug #42 has been resolved")
    @Test
    void testWillBeSkipped() {
    }

    @Test
    void testWillBeExecuted() {
    }

}
```

Esta anotación `@Disabled` no se hereda, por lo que subclases que hereden de claes deshabilitadas o con métodos deshabilitados deberán redeclarar de nuevo esta anotación.

Conditional Test Execution

La API *"ExecutionCondition"* en JUnit Jupiter permite a los desarrolladores decidir si un conjunto de pruebas o una prueba individual debe ejecutarse o no, basándose en ciertas condiciones. Estas condiciones pueden ser definidas de forma programática.

Un ejemplo básico de una condición es *"DisabledCondition"*, que se utiliza con la anotación `@Disabled`. Esta anotación se usa para desactivar una prueba, evitando que se ejecute.

Además de esta anotación, JUnit Jupiter incluye varias otras anotaciones en el paquete 'org.junit.jupiter.api.condition' que permiten activar o desactivar pruebas de manera declarativa, es decir, simplemente aplicando una anotación a las pruebas o contenedores de pruebas.

Es recomendable indicar los motivos por los que una prueba está desactivada. Para ello, muchas de estas anotaciones tienen un atributo `disabledReason` donde se puede especificar el motivo.

Cualquiera de las anotaciones condicionales listadas en las secciones siguientes también puede usarse como meta-anotación para crear una anotación compuesta personalizada, como ya se explicó en meta-anotaciones:

```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
    // ...
}

// Meta-anotación entre '@Test' y '@EnabledOnOs(MAC)'
@TestOnMac
void testOnMac() {
    // ...
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Test
@EnabledOnOs(MAC)
@interface TestOnMac {}
```

A menos que se indique lo contrario, cada una de las anotaciones condicionales enumeradas en las siguientes secciones **solo puede declararse una vez en una interfaz de prueba, clase de prueba o método de prueba**. Si una anotación condicional está presente directamente, indirectamente o como meta-anotación múltiples veces en un mismo elemento, solo se usará la primera anotación descubierta por JUnit.

Operating System and Architecture Conditions

TODO

Java Runtime Environment Conditions

TODO

Native Image Conditions

TODO

System Property Conditions

TODO

Environment Variable Conditions

TODO

Custom Conditions

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

Referencias

- https://junit.org/junit5/
- 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