JUnit 5

… EN DESARROLLO …

Introducción

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 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:

```
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() {
        // ...
    }

}
```

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…:

```
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

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.

```
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

Un contenedor o prueba se puede habilitar o deshabilitar en un sistema operativo, arquitectura o combinación de ambos en particular a través de las anotaciones `@EnabledOnOs` y `@DisabledOnOs`.

Ejecución condicional basada en el **sistema operativo**:

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

```
@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}
```

Ejecución condicional basada en **arquitectura**:

```
@Test
@EnabledOnOs(architectures = "aarch64")
void onAarch64() {
    // ...
}

@Test
@DisabledOnOs(architectures = "x86_64")
void notOnX86_64() {
    // ...
}

@Test
@EnabledOnOs(value = MAC, architectures = "aarch64")
void onNewMacs() {
    // ...
}

@Test
@DisabledOnOs(value = MAC, architectures = "aarch64")
void notOnNewMacs() {
    // ...
}
```

Java Runtime Environment Conditions

Un contenedor o prueba puede habilitarse o deshabilitarse en versiones particulares del Entorno de Ejecución de Java (JRE) mediante las anotaciones `@EnabledOnJre` y `@DisabledOnJre` o en un rango particular de versiones del JRE mediante las anotaciones `@EnabledForJreRange` y `@DisabledForJreRange` .

El rango predeterminado establece "JRE.JAVA_8" como el límite inferior (mínimo) y "JRE.OTHER" como el límite superior (máximo), lo cual permite el uso de rangos semiabiertos.

```
@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
    // ...
}

@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
    // ...
}

@Test
@EnabledForJreRange(min = JAVA_9, max = JAVA_11)
void fromJava9to11() {
    // ...
```

```
}

@Test
@EnabledForJreRange(min = JAVA_9)
void fromJava9toCurrentJavaFeatureNumber() {
    // ...
}

@Test
@EnabledForJreRange(max = JAVA_11)
void fromJava8To11() {
    // ...
}

@Test
@DisabledOnJre(JAVA_9)
void notOnJava9() {
    // ...
}

@Test
@DisabledForJreRange(min = JAVA_9, max = JAVA_11)
void notFromJava9to11() {
    // ...
}

@Test
@DisabledForJreRange(min = JAVA_9)
void notFromJava9toCurrentJavaFeatureNumber() {
    // ...
}

@Test
@DisabledForJreRange(max = JAVA_11)
void notFromJava8to11() {
    // ...
}
```

Native Image Conditions

Un contenedor o prueba puede habilitarse o deshabilitarse dentro de una imagen nativa de GraalVM mediante las anotaciones `@EnabledInNativeImage` y `@DisabledInNativeImage`. Estas anotaciones se utilizan típicamente cuando se ejecutan pruebas dentro de una imagen nativa usando los complementos de Gradle y Maven del proyecto "GraalVM Native Build Tools".

```
@Test
@EnabledInNativeImage
void onlyWithinNativeImage() {
    // ...
}

@Test
@DisabledInNativeImage
void neverWithinNativeImage() {
    // ...
}
```

System Property Conditions

Un contenedor o prueba puede habilitarse o deshabilitarse en función del valor de una **propiedad con nombre del sistema JVM** mediante las anotaciones `@EnabledIfSystemProperty` y `@DisabledIfSystemProperty`. El valor proporcionado a través del atributo *"matches"* se interpretará como una expresión regular.

A partir de JUnit Jupiter 5.6, estas anotaciones son **anotaciones repetibles**. En consecuencia, estas anotaciones pueden declararse múltiples veces en una interfaz de prueba, clase de prueba o método de prueba.

```
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
    // ...
}

@Test
@DisabledIfSystemProperty(named = "ci-server", matches = "true")
void notOnCiServer() {
    // ...
}
```

Environment Variable Conditions

Un contenedor o prueba puede habilitarse o deshabilitarse en función del valor de una **variable de entorno nombrada del sistema operativo subyacente** mediante las anotaciones `@EnabledIfEnvironmentVariable` y `@DisabledIfEnvironmentVariable`. El valor proporcionado a través del atributo *"matches"* se interpretará como una expresión regular.

A partir de JUnit Jupiter 5.6, estas anotaciones son **anotaciones repetibles**. En consecuencia, estas anotaciones pueden declararse múltiples veces en una interfaz de prueba, clase de prueba o método de prueba.

```
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
void onlyOnStagingServer() {
    // ...
}

@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
void notOnDeveloperWorkstation() {
    // ...
}
```

Custom Conditions

Como alternativa a la implementación de una *"ExecutionCondition"*, un contenedor o prueba puede habilitarse o deshabilitarse en función de un método de condición configurado mediante las anotaciones `@EnabledIf` y `@DisabledIf`.

Un método de condición debe tener un tipo de retorno booleano y puede aceptar sin argumentos o un solo argumento de tipo *"ExtensionContext"*.

```
@Test
@EnabledIf("customCondition")
void enabled() {
    // ...
}

@Test
@DisabledIf("customCondition")
void disabled() {
    // ...
}

boolean customCondition() {
    return true;
}
```

Alternativamente, el método de condición puede ubicarse **fuera de la clase de prueba**. En este caso, se debe hacer referencia a él por su nombre completo:

```
package example;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.condition.EnabledIf;

class ExternalCustomConditionDemo {

    @Test
    @EnabledIf("example.ExternalCondition#customCondition")
    void enabled() {
        // ...
    }

}

class ExternalCondition {

    static boolean customCondition() {
        return true;
    }

}
```

Hay varios casos en los que un método de condición debería ser estático:

- cuando `@EnabledIf` o `@DisabledIf` se usan a nivel de clase
- cuando se usa `@EnabledIf` o `@DisabledIf` en un método `@ParameterizedTest` o `@TestTemplate`
- cuando el método de condición está ubicado en una clase externa

A menudo ocurre que puede utilizar un **método estático existente** en una clase de utilidad como condición personalizada.

Por ejemplo, *"java.awt.GraphicsEnvironment"* proporciona el método `public static boolean isHeadless()` que puede usarse para determinar si el entorno actual no admite una pantalla gráfica. Por lo tanto, si tienes una prueba que depende del soporte gráfico, puedes deshabilitarla cuando dicho soporte no esté disponible de la siguiente manera:

```
@DisabledIf(value = "java.awt.GraphicsEnvironment#isHeadless",
    disabledReason = "headless environment")
```

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

Por defecto, las clases y métodos de prueba se ordenarán utilizando un algoritmo que es determinista pero intencionalmente no obvio. Esto asegura que ejecuciones subsecuentes de un conjunto de pruebas ejecuten las clases y métodos de prueba en el mismo orden, permitiendo así construcciones repetibles.

Aunque las verdaderas pruebas unitarias normalmente no deberían depender del orden en el que se ejecutan, hay ocasiones en las que es necesario imponer un orden de ejecución de un método de prueba específico, como por ejemplo en pruebas de integración o pruebas funcionales.

Method Order

Para controlar el orden en el que se ejecutan los métodos de prueba, anota tu clase de prueba o interfaz de prueba con `@TestMethodOrder` y especifica la implementación de "MethodOrderer" deseada.

Se puede implementar un *"MethodOrderer"* personalizado o usar una de las implementaciones integradas.

El siguiente ejemplo demuestra cómo garantizar que los métodos de prueba se ejecuten en el orden especificado mediante la anotación `@Order`:

```
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;

@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {

    @Test
    @Order(1)
    void nullValues() {
        // perform assertions against null values
    }

    @Test
    @Order(2)
    void emptyValues() {
        // perform assertions against empty values
    }

    @Test
    @Order(3)
    void validValues() {
        // perform assertions against valid values
    }

}
```

Class Order

Aunque las clases de prueba típicamente no deben depender del orden en que se ejecutan, hay ocasiones en las que es deseable aplicar un orden específico de ejecución a las clases de prueba. Puedes desear ejecutar las clases de prueba en un orden aleatorio para asegurarte de que no existan dependencias accidentales entre ellas, por ejemplo.

Se puede implementar un "ClassOrderer" personalizado o utilizar una de las implementaciones integradas.

El siguiente ejemplo demuestra cómo garantizar que las clases de prueba `@Nested` se ejecuten en el orden especificado mediante la anotación `@Order`.

```
import org.junit.jupiter.api.ClassOrderer;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestClassOrder;

@TestClassOrder(ClassOrderer.OrderAnnotation.class)
class OrderedNestedTestClassesDemo {

    @Nested
    @Order(1)
    class PrimaryTests {

        @Test
        void test1() {
        }
    }

    @Nested
    @Order(2)
    class SecondaryTests {

        @Test
        void test2() {
        }
    }
}
```

Test Instance Lifecycle

Para permitir que los métodos de prueba individuales se ejecuten de **manera aislada** y evitar **efectos secundarios inesperados** debido al estado mutable de la instancia de prueba, JUnit **crea una nueva instancia** de cada clase de prueba antes de ejecutar cada método de prueba.

Este ciclo de vida de instancia de prueba "por método" es el **comportamiento predeterminado** en JUnit Jupiter y es análogo a todas las versiones anteriores de JUnit.

Si se requiere que JUnit Jupiter ejecute todos los métodos de prueba en **la misma instancia de prueba**, se debe anotar la clase de prueba con `@TestInstance(Lifecycle.PER_CLASS)`:

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

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class MyClassTest {

    private int counter = 0;

    @BeforeAll
    void init() {
        // Este método se ejecuta una vez antes de todos los métodos de prueba
        System.out.println("Inicialización de la clase de prueba...");
    }

    @BeforeEach
    void setUp() {
        // Este método se ejecuta antes de cada método de prueba
        counter++;
        System.out.println("Preparando para ejecutar el método de prueba " + counter);
```

```
    }

    @Test
    void testMethod1() {
        // Método de prueba 1
        System.out.println("Ejecutando método de prueba 1...");
        Assertions.assertEquals(1, counter);
    }

    @Test
    void testMethod2() {
        // Método de prueba 2
        System.out.println("Ejecutando método de prueba 2...");
        Assertions.assertEquals(2, counter);
    }

    @AfterEach
    void tearDown() {
        // Este método se ejecuta después de cada método de prueba
        System.out.println("Finalizando la ejecución del método de prueba " + counter);
    }

    @AfterAll
    void cleanUp() {
        // Este método se ejecuta una vez después de todos los métodos de prueba
        System.out.println("Limpieza después de todos los métodos de prueba");
    }
}
```

Al usar este modo, se creará una nueva instancia de prueba una vez por clase de prueba. Por lo tanto, si los métodos de prueba dependen del estado almacenado en variables de instancia, es posible que se necesite restablecer ese estado en los métodos `@BeforeEach` o `@AfterEach`.

El modo "por clase" tiene algunos beneficios adicionales sobre el modo predeterminado "por método". Específicamente, con el modo "por clase", es posible declarar los métodos `@BeforeAll` y `@AfterAll` en métodos no estáticos, así como en métodos por defecto de interfaces. Por lo tanto, el modo "por clase" también hace posible utilizar métodos `@BeforeAll` y `@AfterAll` en clases de prueba `@Nested`.

Changing the Default Test Instance Lifecycle

Si una clase de prueba o una interfaz de prueba no está anotada con `@TestInstance`, JUnit Jupiter utilizará un modo de ciclo de vida predeterminado. El modo predeterminado estándar es ***"PER_METHOD"***; sin embargo, es posible cambiar el valor predeterminado para la ejecución de un plan de prueba completo.

Para establecer el modo de ciclo de vida de la instancia de prueba predeterminado en ***"Lifecycle.PER_CLASS"*** a través del archivo de configuración de la plataforma JUnit, cree un archivo llamado **"junit-platform.properties"** en la raíz de la ruta de clase (por ejemplo, *"src/test/resources"*) con el siguiente contenido:

```
junit.jupiter.testinstance.lifecycle.default = per_class
```

Existen otras formas de para configurar el modo, pero esta es la recomendable ya que este archivo de configuración puede ser incluido en un sistema de control de versiones junto con el proyecto, lo que permite su uso dentro de los entornos de desarrollo integrado (IDE) y en el software de construcción.

Nested Tests

Los tests anidados (`@Nested`) brindan al desarrollador más capacidades para expresar la relación entre varios grupos de pruebas. Dichos tests anidados hacen uso de las clases anidadas de Java y facilitan el pensamiento jerárquico sobre la estructura de las pruebas.

```java
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());
            }
        }
    }
}
```

En este ejemplo, las condiciones previas de las pruebas externas se utilizan en las pruebas internas mediante la definición de métodos de ciclo de vida jerárquicos para el código de configuración. Por ejemplo, `createNewStack()` es un método de ciclo de vida `@BeforeEach` que se utiliza en la clase de prueba en la que está definido y en todos los niveles del árbol de anidación por debajo de la clase en la que está definido.

Dependency Injection for Constructors and Methods

En todas las versiones anteriores de JUnit, no se permitía que los constructores o métodos de prueba tuvieran parámetros (al menos no con las implementaciones estándar de Runner). Como uno de los principales cambios en JUnit Jupiter, ahora se permite que tanto los constructores de prueba como los métodos tengan **parámetros**. Esto permite una mayor flexibilidad y habilita la inyección de dependencias (DI) para constructores y métodos.

ParameterResolver define la API para extensiones de prueba que desean resolver dinámicamente los parámetros en tiempo de ejecución. Si un constructor de clase de prueba, un método de prueba o un método de ciclo de vida acepta un parámetro, el parámetro debe ser resuelto en tiempo de ejecución por un *"ParameterResolver"* registrado.

Actualmente, hay tres resolutores integrados que se registran automáticamente:

- TestInfoParameterResolver

Si un parámetro de constructor o método es de tipo `TestInfo`, el `TestInfoParameterResolver` proporcionará una instancia de `TestInfo` correspondiente al contenedor o prueba actual como valor para el parámetro. Se puede utilizar para obtener información sobre el contenedor o la prueba actual, como el nombre visible, la clase de prueba, el método de prueba y las etiquetas asociadas. El nombre visible puede ser un nombre técnico, como el nombre de la clase de prueba o del método de prueba, o un nombre personalizado configurado mediante `@DisplayName`.

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

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInfo;

@DisplayName("TestInfo Demo")
class TestInfoDemo {

    TestInfoDemo(TestInfo testInfo) {
        assertEquals("TestInfo Demo", testInfo.getDisplayName());
    }

    @BeforeEach
    void init(TestInfo testInfo) {
        String displayName = testInfo.getDisplayName();
        assertTrue(displayName.equals("TEST 1") || displayName.equals("test2()"));
    }
```

```
    @Test
    @DisplayName("TEST 1")
    @Tag("my-tag")
    void test1(TestInfo testInfo) {
        assertEquals("TEST 1", testInfo.getDisplayName());
        assertTrue(testInfo.getTags().contains("my-tag"));
    }

    @Test
    void test2() {
    }

}
```

- RepetitionExtension

Si un parámetro de método en un método `@RepeatedTest`, `@BeforeEach` o `@AfterEach` es de tipo `RepetitionInfo`, la extensión `RepetitionExtension` proporcionará una instancia de `RepetitionInfo`. Se puede utilizar para obtener información sobre la repetición actual, el número total de repeticiones, el número de repeticiones que han fallado y el umbral de fallos para el `@RepeatedTest` correspondiente. Sin embargo, es importante tener en cuenta que `RepetitionExtension` no se registra fuera del contexto de un `@RepeatedTest`.

- TestReporterParameterResolver

Si un parámetro de constructor o método es de tipo `TestReporter`, el `TestReporterParameterResolver` proporcionará una instancia de `TestReporter`. Se puede utilizar para publicar datos adicionales sobre la ejecución actual de la prueba. Estos datos pueden ser consumidos a través del método `reportingEntryPublished()` en un TestExecutionListener, lo que permite verlos en entornos de desarrollo integrado (IDEs) o incluirlos en informes.

```
class TestReporterDemo {

    @Test
    void reportSingleValue(TestReporter testReporter) {
        testReporter.publishEntry("a status message");
    }

    @Test
    void reportKeyValuePair(TestReporter testReporter) {
        testReporter.publishEntry("a key", "a value");
    }

    @Test
    void reportMultipleKeyValuePairs(TestReporter testReporter) {
        Map<String, String> values = new HashMap<>();
        values.put("user name", "dk38");
        values.put("award year", "1974");

        testReporter.publishEntry(values);
    }

}
```

Test Interfaces and Default Methods

JUnit Jupiter permite que las siguientes anotaciones se declaren en métodos por defecto de interfaces:

- @Test
- @RepeatedTest
- @ParameterizedTest
- @TestFactory

- @TestTemplate
- @BeforeEach
- @AfterEach

Además, las anotaciones `@BeforeAll` y `@AfterAll` pueden ser declaradas en métodos estáticos de una interfaz de prueba o en métodos por defecto de una interfaz si la interfaz de prueba o la clase de prueba están anotadas con `@TestInstance(Lifecycle.PER_CLASS)`.

```
@TestInstance(Lifecycle.PER_CLASS)
interface TestLifecycleLogger {

    static final Logger logger = Logger.getLogger(TestLifecycleLogger.class.getName());

    @BeforeAll
    default void beforeAllTests() {
        logger.info("Before all tests");
    }

    @AfterAll
    default void afterAllTests() {
        logger.info("After all tests");
    }

    @BeforeEach
    default void beforeEachTest(TestInfo testInfo) {
        logger.info(() -> String.format("About to execute [%s]",
            testInfo.getDisplayName()));
    }

    @AfterEach
    default void afterEachTest(TestInfo testInfo) {
        logger.info(() -> String.format("Finished executing [%s]",
            testInfo.getDisplayName()));
    }

}
```

En su clase de prueba, puede implementar estas interfaces de prueba para aplicarlas:

```
class TestInterfaceDemo implements TestLifecycleLogger,
        TimeExecutionLogger, TestInterfaceDynamicTestsDemo {

    @Test
    void isEqualValue() {
        assertEquals(1, "a".length(), "is always equal");
    }

}
```

Su finalidad es definir métodos de prueba genéricos en la interfaz que luego pueden ser implementados por múltiples clases de prueba. Esto promueve la reutilización del código de prueba común entre diferentes implementaciones de pruebas.

Además, con las anotaciones como `@BeforeEach`, `@AfterEach`, `@BeforeAll` y `@AfterAll` en métodos de interfaz, puedes definir configuraciones de inicialización y limpieza que se aplican a todas las implementaciones de pruebas que implementen esa interfaz.

Repeated Test

JUnit Jupiter proporciona la capacidad de **repetir una prueba un número especificado de veces** anotando un método con `@RepeatedTest` y especificando el número total de repeticiones deseadas. Cada invocación de una prueba repetida se comporta

como la ejecución de un método `@Test` regular, con soporte completo para las mismas devoluciones de llamada de ciclo de vida y extensiones.

```
// Se repite 10 veces
@RepeatedTest(10)
void repeatedTest() {
    // ...
}
```

Desde JUnit Jupiter 5.10, `@RepeatedTest` se puede configurar con un **umbral de fallos** que indica el número de fallos después del cual las repeticiones restantes se omitirán automáticamente. Establezca el atributo `failureThreshold` en un número positivo menor que el número total de repeticiones para omitir las invocaciones de las repeticiones restantes después de haber encontrado el número especificado de fallos.

Si por ejemplo se establece en `failureThreshold = 1` la repetición de test se detendrá cuando se produzca 1 test fallido. Por defecto, el atributo `failureThreshold` se establece en `Integer.MAX_VALUE`, lo que significa que no se aplicará ningún umbral de fallos.

De hecho, para evitar efectos inesperados, se recomienda establecer `@Execution(SAME_THREAD)` para que los tests repetidos se ejecuten en el mismo hilo si se ha configurado la ejecución en paralelo.

Además de especificar el número de repeticiones y el umbral de fallos, se puede configurar un nombre personalizado para cada repetición a través del atributo *"name"* de la anotación `@RepeatedTest`. Además, el nombre mostrado puede ser un patrón compuesto por una combinación de texto estático y marcadores de posición dinámicos. Actualmente, se admiten los siguientes marcadores de posición:

- `{displayName}`: nombre mostrado del método `@RepeatedTest`
- `{currentRepetition}`: el conteo de la repetición actual
- `{totalRepetitions}`: el número total de repeticiones

El nombre mostrado predeterminado para una repetición dada se genera basado en el siguiente patrón:

- "repetition {currentRepetition} of {totalRepetitions}"

Para recuperar información sobre la repetición actual, el número total de repeticiones, el número de repeticiones que han fallado y el umbral de fallos, un desarrollador puede optar por tener una instancia de `RepetitionInfo` inyectada en un método `@RepeatedTest`, `@BeforeEach` o `@AfterEach`.

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

import java.util.logging.Logger;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.RepetitionInfo;
import org.junit.jupiter.api.TestInfo;

class RepeatedTestsDemo {

    private Logger logger = // ...

    @BeforeEach
    void beforeEach(TestInfo testInfo, RepetitionInfo repetitionInfo) {
        int currentRepetition = repetitionInfo.getCurrentRepetition();
        int totalRepetitions = repetitionInfo.getTotalRepetitions();
```

```
        String methodName = testInfo.getTestMethod().get().getName();
        logger.info(String.format("About to execute repetition %d of %d for %s", //
            currentRepetition, totalRepetitions, methodName));
    }

    @RepeatedTest(10)
    void repeatedTest() {
        // ...
    }

    @RepeatedTest(5)
    void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
        assertEquals(5, repetitionInfo.getTotalRepetitions());
    }

    @RepeatedTest(value = 8, failureThreshold = 2)
    void repeatedTestWithFailureThreshold(RepetitionInfo repetitionInfo) {
        // Simulate unexpected failure every second repetition
        if (repetitionInfo.getCurrentRepetition() % 2 == 0) {
            fail("Boom!");
        }
    }

    @RepeatedTest(value = 1, name = "{displayName} {currentRepetition}/{totalRepetitions}")
    @DisplayName("Repeat!")
    void customDisplayName(TestInfo testInfo) {
        assertEquals("Repeat! 1/1", testInfo.getDisplayName());
    }

    @RepeatedTest(value = 1, name = RepeatedTest.LONG_DISPLAY_NAME)
    @DisplayName("Details...")
    void customDisplayNameWithLongPattern(TestInfo testInfo) {
        assertEquals("Details... :: repetition 1 of 1", testInfo.getDisplayName());
    }

    @RepeatedTest(value = 5, name = "Wiederholung {currentRepetition} von {totalRepetitions}")
    void repeatedTestInGerman() {
        // ...
    }

}
```

Parameterized Test

Las pruebas parametrizadas permiten ejecutar una prueba **varias veces con diferentes argumentos**. Se declaran como los métodos `@Test` normales, pero en su lugar utilizan la anotación `@ParameterizedTest`.

Además, se debe declarar al menos una fuente que proporcionará los argumentos para cada invocación y luego se consumirán los argumentos en el método de prueba.

Ejemplo de test parametrizado con la anotación `@ValueSouce` que provee un array de strings como fuente:

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

Required Setup

Para utilizar las pruebas parametrizadas, se debe agregar la dependencia en el artefacto ***"junit-jupiter-params"***.

En Maven:

```xml
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.9.3</version> <!-- Usar la versión más actual -->
    <scope>test</scope>
</dependency>
```

En Gradle:

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-params:5.9.3' // Usar la versión más actual
}
```

Consuming Arguments

Este fragmento de la documentación de JUnit explica cómo deben declararse los métodos de prueba parametrizados y cómo se manejan los diferentes tipos de argumentos que pueden ser proporcionados a estos métodos.

1. **Indexed Arguments (Argumentos Indexados)**:
    - Los métodos de prueba parametrizados normalmente consumen argumentos directamente desde una fuente configurada, como `@CsvSource`.
    - Hay una correspondencia directa uno a uno entre el índice de la fuente de argumentos y el índice del parámetro del método.
    - Por ejemplo, si estás utilizando `@CsvSource`, cada valor separado por comas en la fila de datos CSV se asignará directamente a un parámetro en el método de prueba.

2. **Argument Aggregation (Agregación de Argumentos)**:
    - Además de consumir argumentos uno a uno, un método de prueba parametrizado puede optar por agrupar varios argumentos de la fuente en un solo objeto que se pasa al método.
    - Esto es útil cuando los datos en la fuente están estructurados y se desean manejar como una única entidad en el método de prueba.

3. **ParameterResolver (Resolutores de Parámetros)**:
    - Los métodos de prueba parametrizados pueden también recibir argumentos adicionales proporcionados por un ParameterResolver.
    - Por ejemplo, esto podría incluir objetos como `TestInfo`, `TestReporter`, etc., que son proporcionados por JUnit para obtener información sobre la prueba en curso o para reportar resultados.

En cuanto a la declaración de parámetros en un método de prueba parametrizado:

- Deben declararse primero los argumentos indexados (si los hay), seguidos de los agregadores (si los hay), y finalmente los argumentos proporcionados por un `ParameterResolver`.
- **Argumento indexado**: Es aquel que corresponde directamente al índice de un ArgumentsProvider y se asigna al parámetro del método en el mismo índice.
- **Aggregator (Agregador)**: Es un parámetro que maneja múltiples valores de la fuente de argumentos, ya sea utilizando `ArgumentsAccessor` o anotándolo con `@AggregateWith`.

JUnit proporciona flexibilidad en cómo se pueden manejar los argumentos en los métodos de prueba parametrizados, permitiendo diferentes estrategias según las necesidades específicas de la prueba.

Source of Arguments

JUnit Jupiter proporciona una serie de anotaciones de fuente de argumentos. Estas anotaciones se encuentran en el paquete *org.junit.jupiter.params.provider*.

@ValueSource

`@ValueSource` es una de las fuentes más simples posibles. Permite especificar una única matriz de valores literales y solo se puede utilizar para proporcionar un único argumento por invocación de prueba parametrizada.

Los tipos soportados por esta anotación son:

- **short**
- **byte**
- **int**
- **long**
- **float**
- **double**
- **char**
- **boolean**
- **java.lang.String**
- **java.lang.Class**

En el ejemplo la prueba se ejecuta **tres veces**, una por cada valor suministrado:

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

Null and Empty Sources

Para comprobar los casos extremos y verificar el comportamiento adecuado cuando se recibe una entrada incorrecta, puede resultar útil proporcionar valores nulos y vacíos a las pruebas parametrizadas:

- **@NullSource**: proporciona un único argumento nulo en el método anotado.
- **@EmptySource**: proporciona un único argumento vacío.
- **@NullAndEmptySource**: es una meta-anotación de las otras dos anotaciones.

Si se necesita proporcionar varios tipos diferentes de **cadenas en blanco** a una prueba parametrizada, se puede lograr con algo parecido al ejemplo `@ValueSource(strings = {" ", " ", "\t", "\n"})`.

También se puede combinar `@NullSource`, `@EmptySource` y `@ValueSource` para probar una gama más amplia de entradas nulas, vacías y en blanco. El siguiente ejemplo demuestra cómo lograr esto para cadenas:

```
@ParameterizedTest
@NullSource
@EmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}
```

@EnumSource

Esta anotación `@EnumSource` proporciona una manera conveniente de utilizar enumeraciones. La prueba se ejecutará por **cada una de las constantes de la enumeración**:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;
import static org.junit.jupiter.api.Assertions.assertNotNull;

enum Day implements IDayOfWeek {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumSourceTest {

    @ParameterizedTest
    @EnumSource
    void testWithEnumSource(Day day) {
        assertNotNull(day);
    }

}
```

El atributo `value` de la anotación `@EnumSource` es opcional. En el siguiente ejemplo se utiliza este atributo en la anotación para indicar la clase de la enumeración, ya que en el parámetro del método de prueba se utiliza la interfaz implementada por el enumerado:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;
import static org.junit.jupiter.api.Assertions.assertNotNull;

enum Day implements IDayOfWeek {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumSourceTest {

    @ParameterizedTest
    @EnumSource(Day.class)
    void testWithEnumSource(IDayOfWeek day) {
        assertNotNull(day);
    }

}
```

La anotación proporciona un atributo `names` opcional que le permite especificar qué constantes se utilizarán. Si se omite, se utilizarán todas las constantes.

```
// Sólo se ejecuta para "MONDAY", "TUESDAY"
@ParameterizedTest
@EnumSource(names = { "MONDAY", "TUESDAY" })
void testWithEnumSource(Day day) {
    assertNotNull(day);
}
```

La anotación `@EnumSource` también proporciona un atributo `mode` opcional que permite un control detallado sobre qué constantes se pasan al método de prueba, como por ejemplo, excluir algunas constantes:

```
// Sólo se ejecuta para "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY"
@ParameterizedTest
@EnumSource(mode = EnumSource.Mode.EXCLUDE, names = {"SATURDAY", "SUNDAY"})
void testWorkingDays(Day day) {
    assertNotNull(day);
}
```

@MethodSource

La anotación `@MethodSource` le permite hacer referencia a uno o más métodos de fábrica de la clase de prueba o clases externas.

Los métodos de fábrica dentro de la clase de prueba deben ser **estáticos** a menos que la clase de prueba esté anotada con `@TestInstance(Lifecycle.PER_CLASS);` mientras que los métodos de fábrica en **clases externas siempre deben ser estáticos**.

Cada método de fábrica debe generar una secuencia o *"Stream"* de argumentos, y cada conjunto de argumentos dentro de la secuencia se proporcionará como argumentos físicos para invocaciones individuales del método `@ParameterizedTest` anotado.

En términos generales, esto se traduce en un *"Stream"* de *"Arguments"* (es decir, `Stream<Arguments>`); sin embargo, el tipo de devolución concreto real puede adoptar muchas formas. En este contexto, una "secuencia" es cualquier cosa que JUnit pueda convertir de manera confiable en una secuencia, como `Stream`, `DoubleStream`, `LongStream`, `IntStream`, `Collection`, `Iterator`, `Iterable`, una matriz de objetos o una matriz de primitivas.

Por ejemplo, tres métodos que suministren argumentos a un método de prueba podrían ser:

```
static String[] workingDaysProvider() {
    return new String[]{"MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY"};
}

static Stream<String> workingDaysProvider2() {
    return Stream.of("MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY");
}

static Stream<Integer> workingDaysProvider3() {
    return Stream.of(1, 2, 3, 4, 5);
}

static IntStream workingDaysProvider4() {
    return IntStream.of(1, 2, 3, 4, 5);
}
```

El método de prueba que podría hacer uso del método `workingDaysProvider()`:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;
import java.util.stream.Stream;
import static org.junit.jupiter.api.Assertions.assertNotNull;

public class MethodSourceTest {

    @ParameterizedTest
    @MethodSource("workingDaysProvider")
    void testWithExplicitLocalMethodSource(String argument) {
        assertNotNull(argument);
    }

    static Stream<String> workingDaysProvider() {
        return Stream.of("MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY");
    }
```

```
}
```

Si no se proporciona explícitamente un nombre de método de fábrica a través de `@MethodSource`, JUnit Jupiter buscará un método de fábrica que tenga el mismo nombre que el método `@ParameterizedTest` actual por convención.

```
@ParameterizedTest
@MethodSource
void testWithDefaultLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> testWithDefaultLocalMethodSource() {
    return Stream.of("apple", "banana");
}
```

@CsvSource

La anotación `@CsvSource` le permite expresar listas de argumentos como valores separados por comas (es decir, literales de cadena CSV). Cada cadena proporcionada mediante el atributo de valor en `@CsvSource` representa un registro CSV y da como resultado una invocación de la prueba parametrizada. Opcionalmente, el primer registro se puede utilizar para proporcionar encabezados CSV.

```
@ParameterizedTest
@CsvSource({
    "apple,         1", // "apple", "1"
    "banana,        2", // "banana", "2"
    "'lemon, lime', 3", // "lemon, lime", "3"
    "strawberry,    ''", // "strawberry", ""
    "banana, "          // "banana", null
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertNotEquals(0, rank);
}
```

El delimitador predeterminado es una coma (,), pero puede usar otro carácter configurando el atributo `delimiter`. Alternativamente, el atributo `delimiterString` le permite utilizar un delimitador de cadena en lugar de un solo carácter. Sin embargo, ambos atributos delimitadores no se pueden establecer simultáneamente.

De forma predeterminada, `@CsvSource` utiliza una comilla simple (') como carácter de comilla, pero esto se puede cambiar mediante el atributo `quoteCharacter`.

Un valor entrecomillado vacío ('') da como resultado una cadena vacía a menos que se establezca el atributo `emptyValue`; mientras que un valor completamente vacío se interpreta como una referencia nula, como por ejemplo `"banana, "`, que falta el segundo argumento.

@CsvFileSource

La anotación `@CsvFileSource` le permite utilizar archivos de valores separados por comas (CSV) del classpath o del sistema de archivos local. Cada registro de un archivo CSV da como resultado una invocación de la prueba parametrizada. Opcionalmente, el primer registro se puede utilizar para proporcionar encabezados CSV.

Se pueden añadir **comentarios** en un fichero CSV. Cualquier línea que comience con el símbolo `#` se interpretará como un comentario y se ignorará.

```
# Fichero CSV
COUNTRY, REFERENCE
Sweden, 1
Poland, 2
"United States of America", 3
France, 700_000
```

A diferencia de la sintaxis predeterminada utilizada en `@CsvSource`, `@CsvFileSource` utiliza una comilla doble (") como carácter de comilla de forma predeterminada, pero esto se puede cambiar mediante el atributo `quoteCharacter`.

@ArgumentsSource

La anotación `@ArgumentsSource` se puede utilizar para especificar un `ArgumentsProvider` personalizado y reutilizable.

Hay que tener en cuenta que debe crearse una clase de nivel superior (o clase anidada estática) que implemente la interfaz `ArgumentsProvider`. Esta interfaz tiene un método llamado `provideArguments(ExtensionContext context)` que devuelve un `Stream` de `Arguments`:

```
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.ArgumentsProvider;

import java.util.stream.Stream;

public class MyArgumentsProvider implements ArgumentsProvider {
    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
        return Stream.of(
            Arguments.of("arg1", 1),
            Arguments.of("arg2", 2),
            Arguments.of("arg3", 3)
        );
    }
}
```

La clase de prueba que utiliza la anotación `@ArgumentsSource`:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ArgumentsSource;

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

public class MyParameterizedTest {
    @ParameterizedTest
    @ArgumentsSource(MyArgumentsProvider.class)
    void testWithArgumentsSource(String argument, int number) {
        // Realiza tus aserciones aquí
        System.out.println(argument + " - " + number);
        assertEquals(1, number);  // Ejemplo de aserción
    }
}
```

Esta anotación ofrece **flexibilidad** ya que se pueden crear conjuntos de argumentos complejos o dinámicos, como leer datos de un archivo, una base de datos o generar datos en tiempo de ejecución.

Además, promueve la **reutilización** y favorece el **desacoplamiento** ya que se puede reutilizar los mismos conjuntos de argumentos en diferentes pruebas al no estar ligados los argumentos y la prueba.

Argument Conversion

JUnit Jupiter admite la conversión ampliada de tipos primitivos o *"Widening Primitive Conversion"* para los argumentos proporcionados a `@ParameterizedTest`.

La _"Widening Primitive Conversion" es un concepto de Java que se refiere a la conversión automática de un tipo de datos primitivo a otro tipo más amplio (de menor a mayor capacidad) sin pérdida de información:

- `byte` ⇒ short, int, long, float, double
- `short` ⇒ int, long, float, double
- `char` ⇒ int, long, float, double
- `int` ⇒ long, float, double
- `long` ⇒ float, double
- `float` ⇒ double

En el contexto de JUnit, especialmente con pruebas parametrizadas, las conversiones ampliadas le permiten a JUnit manejar estas conversiones automáticamente al pasar argumentos a métodos de prueba, evitando la necesidad de conversión manual y reduciendo la posibilidad de errores.

Las conversiones pueden ser **conversiones implícitas**. JUnit Jupiter proporciona una serie de convertidores de tipo implícitos integrados.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

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

public class ImplicitConversionTest {

    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5})
    void testWithImplicitConversion(long number) {
        // JUnit maneja automáticamente la conversión de int a long
        System.out.println(number);
        assertTrue(number > 0);
    }
}
```

Además, JUnit Jupiter proporciona una serie de convertires implícitos de `String` a una variedad de tipos, tanto primitivos como clases relacionadas con fechas, UUID, etcétera… La lista completa se puede consultar en la documentación oficial.

Además de esta conversión implícita de `String` a los tipos de destino enumerados en la tabla, JUnit Jupiter también proporciona un mecanismo alternativo para la conversión automática de una cadena a un tipo de destino determinado si el tipo de destino declara exactamente un método de fábrica adecuado o un constructor de fábrica como definido a continuación:

- ***"factory method"***: un método estático no privado declarado en el tipo de destino que acepta un único argumento `String` y devuelve una instancia del tipo de destino. El nombre del método puede ser arbitrario y no necesita seguir ninguna convención particular.
- ***"factory constructor"***: un constructor no privado en el tipo de destino que acepta un único argumento `String`. Tenga en cuenta que el tipo de destino debe declararse como una clase de nivel superior o como una clase anidada estática.

```
public class Book {

    private final String title;
```

```
    private Book(String title) {
        this.title = title;
    }

    public static Book fromTitle(String title) {
        return new Book(title);
    }

    public String getTitle() {
        return this.title;
    }
}
```

De forma automática el argumento `Book` se creará invocando el método de fábrica `Book.fromTitle(String)` y pasando la cadena "42 Cats" como título del libro.

```
@ParameterizedTest
@ValueSource(strings = "42 Cats")
void testWithImplicitFallbackArgumentConversion(Book book) {
    assertEquals("42 Cats", book.getTitle());
}
```

JUnit Jupiter también ofrece la posibilidad de la **conversión explícita**. Para ello se puede especificar explícitamente un `ArgumentConverter` para usar con un determinado parámetro usando la anotación `@ConvertWith`.

Argument Aggregation

Es una característica avanzada de las pruebas parametrizadas en JUnit 5 que permite combinar múltiples parámetros de entrada en un solo objeto antes de pasar los valores al método de prueba.

Esta técnica es útil cuando se tienen múltiples argumentos que representan una entidad lógica y se prefiere encapsular estos argumentos en un solo objeto.

En tales casos, se puede utilizar un `ArgumentsAccessor` en lugar de varios parámetros, teniendo en cuenta que se aplica la conversión implícita de `String` a otros tipos vista anteriormente.

```
@ParameterizedTest
@CsvSource({
    "Jane, Doe, F, 1990-05-20",
    "John, Doe, M, 1990-10-22"
})
void testWithArgumentsAccessor(ArgumentsAccessor arguments) {
    Person person = new Person(arguments.getString(0),
                               arguments.getString(1),
                               arguments.get(2, Gender.class),
                               arguments.get(3, LocalDate.class));

    if (person.getFirstName().equals("Jane")) {
        assertEquals(Gender.F, person.getGender());
    }
    else {
        assertEquals(Gender.M, person.getGender());
    }
    assertEquals("Doe", person.getLastName());
    assertEquals(1990, person.getDateOfBirth().getYear());
}
```

Además, es posible crear agregaciones personalizadas implementando la interfaz `ArgumentsAggregator`.

Customizing Display Names

De forma predeterminada, el nombre mostrado en una invocación de prueba parametrizada contiene el índice de invocación y la representación `String` de todos los argumentos para esa invocación específica.

Sin embargo, se puede personalizar los nombres para mostrar de invocaciones mediante el atributo `name` de la anotación `@ParameterizedTest`:

```
@DisplayName("Display name of container")
@ParameterizedTest(name = "{index} ==> the rank of ''{0}'' is {1}")
@CsvSource({ "apple, 1", "banana, 2", "'lemon, lime', 3" })
void testWithCustomDisplayNames(String fruit, int rank) {
    // ...
}
```

La ejecución de esta prueba parametrizada mostraría por consola:

```
Display name of container ✓
├─ 1 ==> the rank of 'apple' is 1 ✓
├─ 2 ==> the rank of 'banana' is 2 ✓
└─ 3 ==> the rank of 'lemon, lime' is 3 ✓
```

El parámetro `name` es un patrón `MessageFormat`. Por lo tanto, una comilla simple (') debe representarse como una comilla simple doble ('') para poder mostrarse.

La lista de *"placeholders"* que se pueden utilizar:

- `{displayName}`: el nombre para mostrar del método
- `{index}`: el índice de invocación actual (basado en 1)
- `{arguments}`: la lista completa de argumentos separados por comas
- `{argumentsWithNames}`: la lista completa de argumentos separados por comas con nombres de parámetros
- `{0}, {1}, ...`: un argumento individual

Al utilizar `@MethodSource` o `@ArgumentsSource`, puede proporcionar nombres personalizados para los argumentos utilizando la API `Named`:

```
@DisplayName("A parameterized test with named arguments")
@ParameterizedTest(name = "{index}: {0}")
@MethodSource("namedArguments")
void testWithNamedArguments(File file) {
}

static Stream<Arguments> namedArguments() {
    return Stream.of(
        arguments(named("An important file", new File("path1"))),
        arguments(named("Another file", new File("path2")))
    );
}
```

Cuya salida es:

```
A parameterized test with named arguments ✓
├─ 1: An important file ✓
└─ 2: Another file ✓
```

Lifecycle and Interoperability

Cada invocación de una prueba parametrizada tiene **el mismo ciclo de vida** que un método `@Test` normal. Por ejemplo, los métodos `@BeforeEach` se ejecutarán antes de cada invocación.

Test Templates

Un método anotado con `@TestTemplate` no es un caso de prueba regular, sino más bien una plantilla para casos de prueba. Como tal, está diseñado para ser invocado múltiples veces, dependiendo del número de contextos de invocación devueltos por los proveedores registrados. Por lo tanto, debe usarse en conjunto con una extensión registrada de `TestTemplateInvocationContextProvider`.

Cada invocación de un método de plantilla de prueba se comporta como la ejecución de un método `@Test` regular, con soporte completo para las mismas devoluciones de llamada del ciclo de vida y extensiones.

Las *"Repeated Tests"* y las *"Parameterized Tests"* son especializaciones integradas de las *"Test Templates"*.

Dynamic Tests

Los ***Dynamic Tests*** en JUnit 5 son una característica poderosa que permite definir pruebas de manera programática en tiempo de ejecución, en lugar de de manera estática como con los métodos anotados con `@Test`. Esto es particularmente útil cuando la cantidad de pruebas o los datos de prueba no son conocidos de antemano.

Los tests dinámicos se anotan con `@TestFactory`. A diferencia de los métodos `@Test`, un método `@TestFactory` no es en sí mismo un caso de prueba sino una fábrica de casos de prueba.

Técnicamente hablando, un método `@TestFactory` debe devolver un único `DynamicNode` o un `Stream`, `Collection`, `Iterable`, `Iterator` o array de instancias de `DynamicNode`.

Las subclases instanciables de `DynamicNode` son `DynamicContainer` y `DynamicTest`. Las instancias de `DynamicContainer` se componen de un nombre para mostrar y una lista de nodos secundarios dinámicos, lo que permite la creación de jerarquías anidadas arbitrariamente de nodos dinámicos.

Para crear un *"Dynamic Test"*, usas el método `DynamicTest.dynamicTest(String displayName, Executable executable)`, donde *"displayName"* es una descripción legible de la prueba y *"executable"* es el código que se ejecuta como la prueba.

```java
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;
import org.junit.jupiter.api.function.Executable;

import java.util.Arrays;
import java.util.Collection;

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

public class DynamicTestExample {

    @TestFactory
    Collection<DynamicTest> dynamicTests() {
        return Arrays.asList(
            DynamicTest.dynamicTest("Test 1", () -> assertTrue(isEven(2))),
            DynamicTest.dynamicTest("Test 2", () -> assertTrue(isEven(4))),
            DynamicTest.dynamicTest("Test 3", () -> assertTrue(isEven(6)))
```

```
        );
    }

    private boolean isEven(int number) {
        return number % 2 == 0;
    }
}
```

Timeouts

La anotación `@Timeout` permite declarar que una prueba, una fábrica de pruebas, una plantilla de prueba o un método de ciclo de vida deben fallar si su tiempo de ejecución excede una duración determinada. La unidad de tiempo para la duración predeterminada es segundos, pero es configurable.

```
class TimeoutDemo {

    @BeforeEach
    @Timeout(5)
    void setUp() {
        // fails if execution time exceeds 5 seconds
    }

    @Test
    @Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
    void failsIfExecutionTimeExceeds500Milliseconds() {
        // fails if execution time exceeds 500 milliseconds
    }

    @Test
    @Timeout(value = 500, unit = TimeUnit.MILLISECONDS, threadMode = ThreadMode.SEPARATE_THREAD)
    void failsIfExecutionTimeExceeds500MillisecondsInSeparateThread() {
        // fails if execution time exceeds 500 milliseconds, the test code is executed in a separate thread
    }

}
```

Para aplicar el mismo tiempo de espera a todos los métodos de prueba dentro de una clase de prueba y a todas sus clases `@Nested`, puede declarar la anotación `@Timeout` **a nivel de clase**. Luego se aplicará a todos los métodos de prueba, fábrica de pruebas y plantilla de prueba dentro de esa clase y sus clases anidadas, a menos que lo anule una anotación `@Timeout` en un método específico o clase anidada.

La anotación `@Timeout` en JUnit 5 se puede configurar mediante **parámetros de configuración**. Esto es útil para establecer un tiempo de espera global o específico para las pruebas sin tener que especificarlo en cada prueba individualmente.

Se puede establecer estas propiedades en un archivo de configuración (*"junit-platform.properties"*) o pasarlas como parámetros del sistema.

Algunos de los parámetros son:

- *"junit.jupiter.execution.timeout.default"*
- *"junit.jupiter.execution.timeout.test.method.default"*
- *"junit.jupiter.execution.timeout.testable.method.default"*

Parallel Execution

TODO

Built-in Extensions

TODO

Referencias

- https://junit.org/junit5
- https://github.com/junit-team/junit5-samples
- https://junit.org/junit5/docs/current/api/
- https://www.baeldung.com/junit
- https://www.tutorialspoint.com/junit/index.htm
- https://www.vogella.com/tutorials/JUnit/article.html

Licencia