FACULTAD DE INGENIERIA
UNIVERSIDAD NACIONAL DE JUJUY

Pruebas Unitarias - Maven

Programación Orientada a Objetos

San Salvador de Jujuy

UNJu – Facultad de Ingeniería
Ing. José Zapana

¿Qué es Maven?

- Es una herramienta de software para la gestión y construcción de proyectos Java
- Creada por Jason van Zyl, de Sonatype, en 2002
- https://maven.apache.org/

¿Cómo funciona?

- Utiliza un Project Object Model (POM) para
  - describir el proyecto de software a construir
  - sus dependencias de otros módulos y componentes externos, y
  - el orden de construcción de los elementos.
- Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado

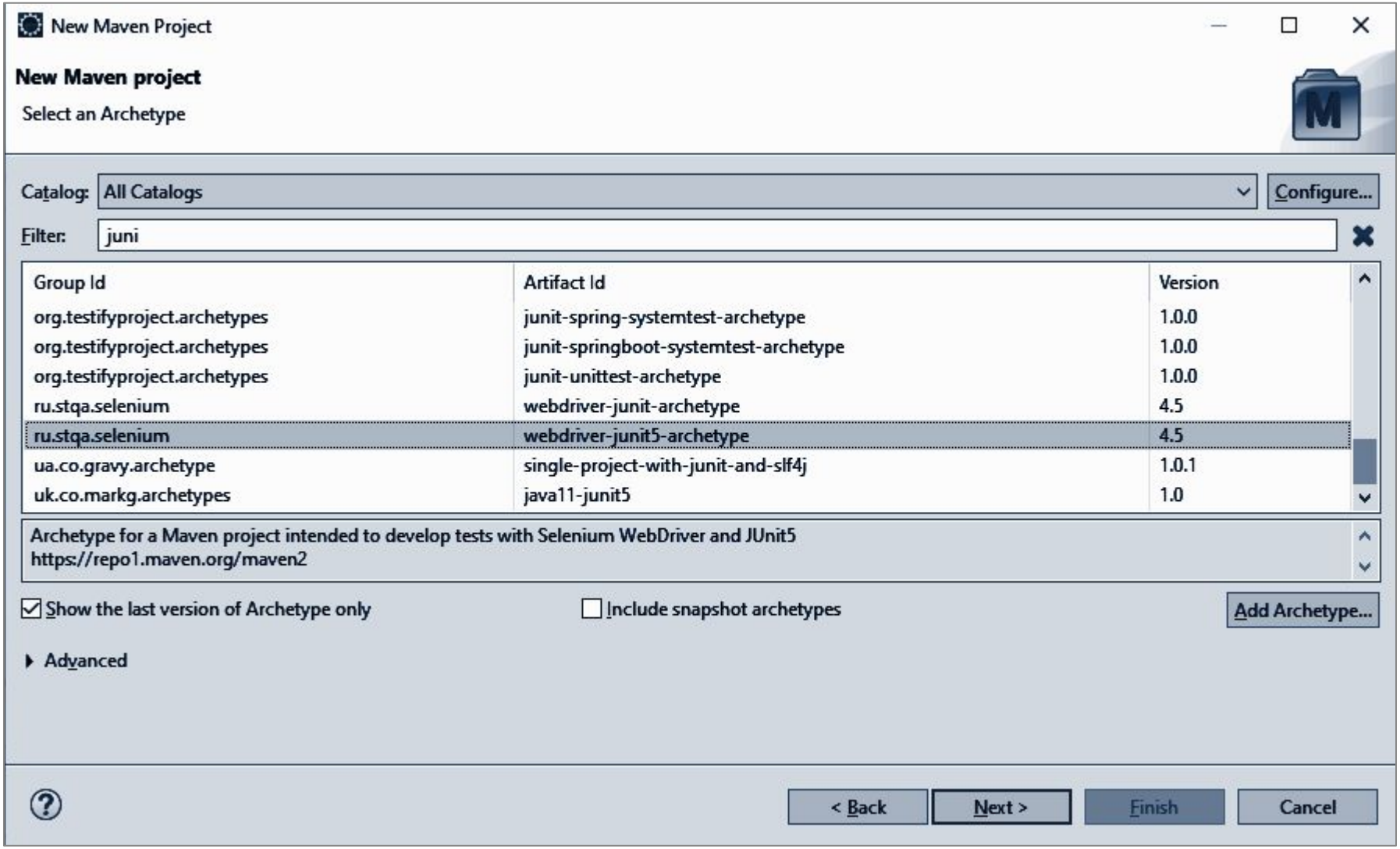
Ciclo de vida

- ***compile***: Genera los ficheros .class compilando los fuentes .java
- ***test*****:** Ejecuta los test automáticos de JUnit existentes, abortando el proceso si alguno de ellos falla.
- ***package*****:** Genera el fichero .jar con los .class compilados
- ***install*****:** Copia el fichero .jar a un directorio de nuestro ordenador donde maven deja todos los .jar. De esta forma esos .jar pueden utilizarse en otros proyectos maven en el mismo ordenador.
- ***deploy*****:** Copia el fichero .jar a un servidor remoto, poniéndolo disponible para cualquier proyecto maven con acceso a ese servidor remoto.

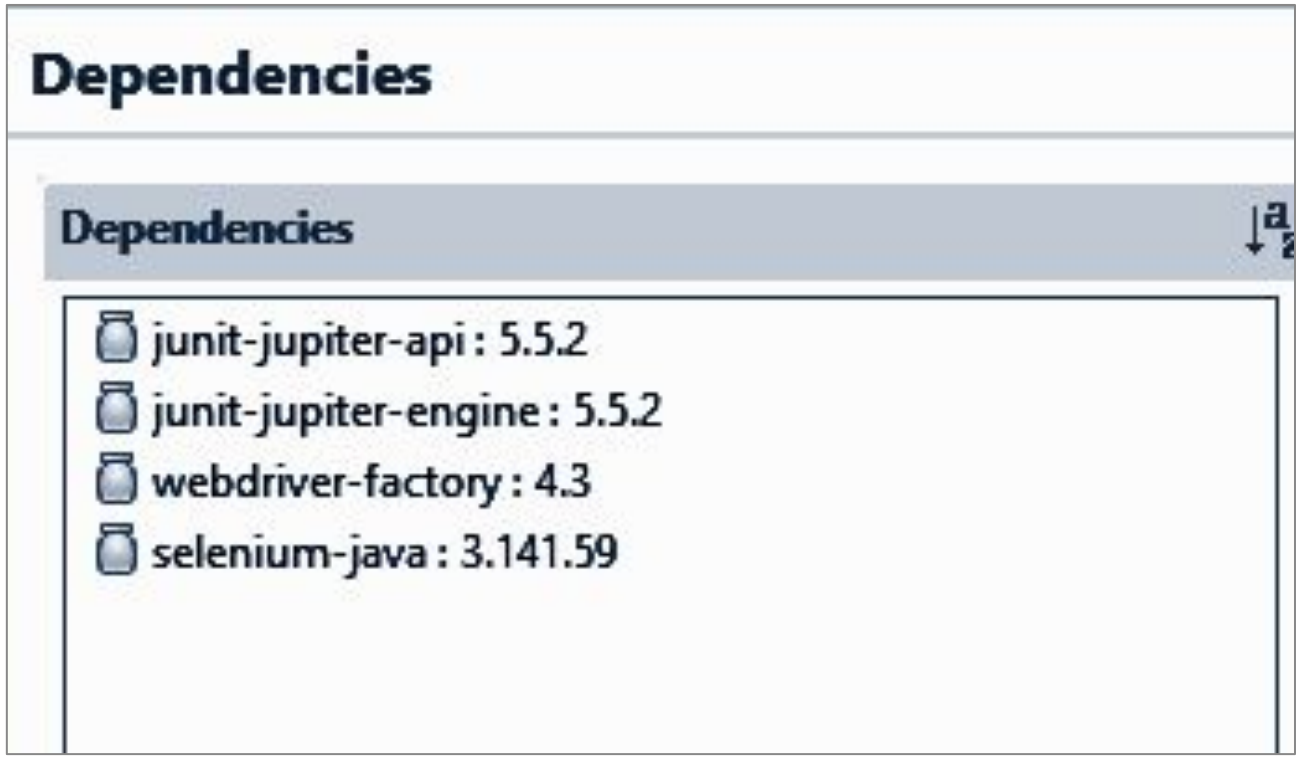
Crear un proyecto Maven

- Desde eclipse
  - Nuevo -> proyecto Maven
  - Es posible seleccionar archetypes predefinidos
  - Definir el "Group Id": identificador único de la organización
  - Definir el "Artifact Id": identificador único del artefacto principal de este proyecto
  - Agregar dependencias

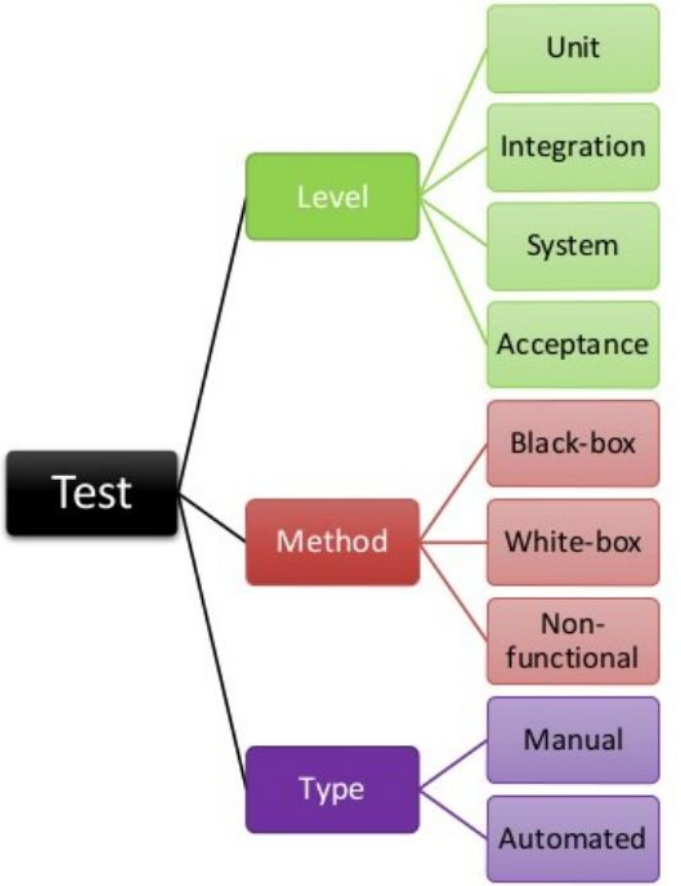
Arquetipo para JUnit5



POM - Dependencias creadas



Pruebas de software



- Pruebas unitarias (componentes aislados)
- Pruebas de integración (diferentes componentes)
- Pruebas de sistema (todos los componentes)
- Pruebas de aceptación (pruebas de usuario)
- Pruebas de caja negra (funcionales)
- Pruebas de caja blanca (estructurales)
- Pruebas no funcionales (rendimiento, seguridad,…)
- Pruebas manuales
- Pruebas automáticas (*frameworks* de prueba)

Técnicas de Testing

- Escribir casos de testing es deseable
- Escribir casos de testing es costoso
- Testear todos los métodos no es práctico
  - Por ejemplo métodos accessors
- Se busca la cobertura necesaria según el problema que se está analizando.

Pruebas Unitarias - Características

- Automatizable
  - No requiere intervención humana
- Completas
  - Deben cubrir la mayor cantidad de código
- Repetibles o Reutilizables
  - No deben funcionar para una sola vez
- Independientes
  - La ejecución de una prueba no debe afectar la ejecución de otra.
- Profesionales
  - Deben ser consideradas como un entregable más…

Pruebas Unitarias

- Fomentan el cambio
  - Permiten hacer las pruebas sobre cambios y asegurarse que los cambios no generan errores.
- Simplifican la integración
  - Permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
- Documenta el código
  - Las propias pruebas son documentación del código
- Separación de la interfaz y la implementación
  - El cambio de la tecnología de interfaz no debe afectar las pruebas.
- Los errores están más acotados y son más fáciles de localizar

Limitaciones

- Las pruebas unitarias NO descubrirán todos los errores de código.
  - No descubrirán errores de integración.
  - Problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto.
- Sólo son efectivas si se usan en conjunto con otras pruebas de software.

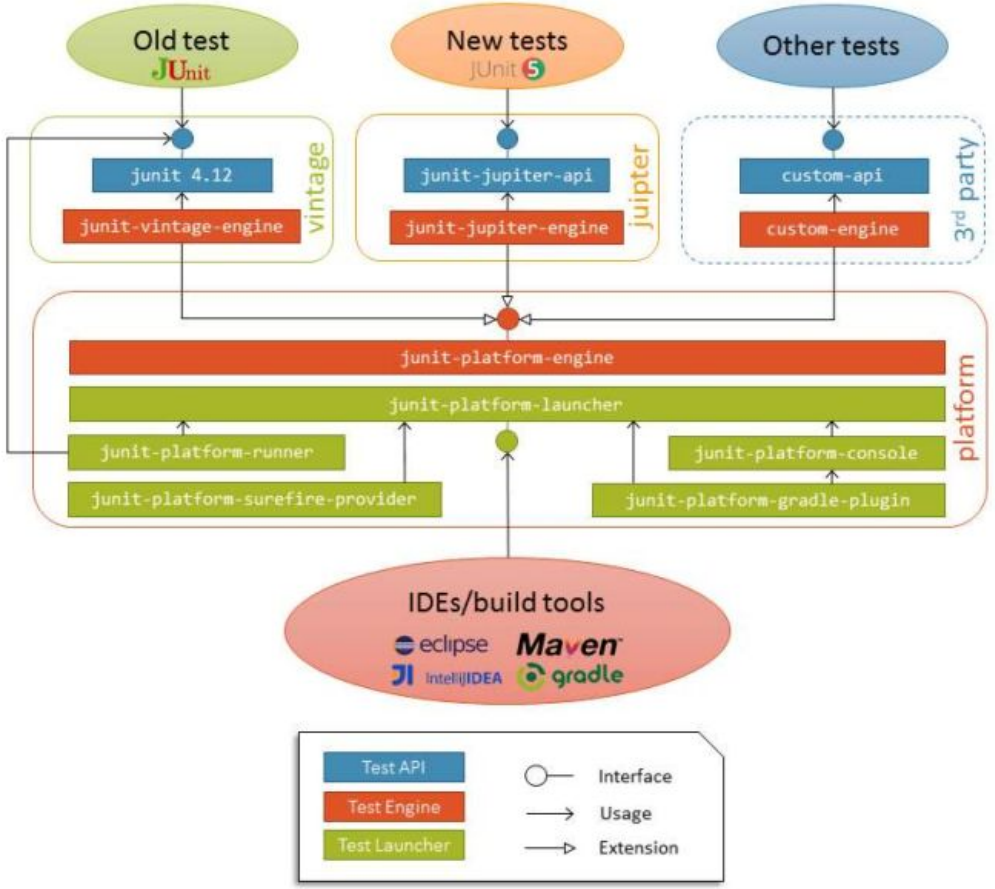
JUnit

- JUnit es un “framework” para automatizar las pruebas de programas Java
- Escrito por Erich Gamma y Kent Beck
- Open Source, disponible en http://www.junit.org

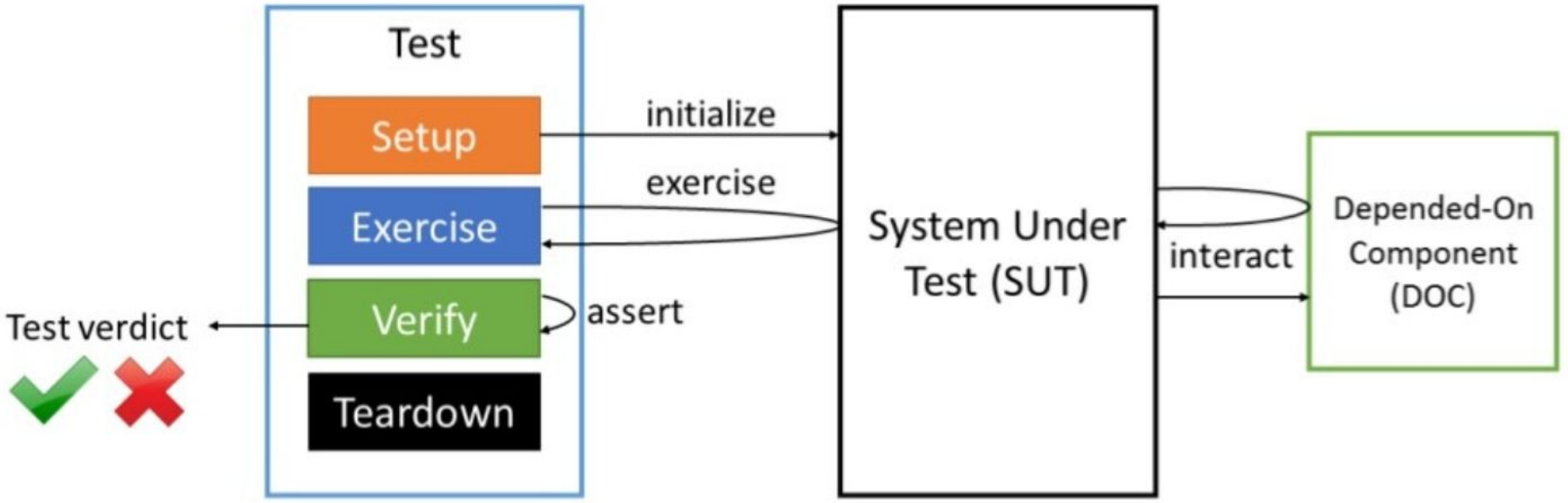
Arquitectura de JUnit 5

- Hay tres tipos de módulos:

1. **Test API**: Módulos usados por *testers* para implementar casos de prueba
2. **Test Engine SPI**: Módulos extendidos para un *framework* de pruebas Java para la ejecución de un modelo concreto de tests
3. **Test Launcher API**: Módulos usados por *clientes programáticos* para el descubrimiento de tests



Esquema de pruebas unitarias



Aserciones

- Son declaraciones o expresiones que se utilizan en pruebas unitarias para verificar que un resultado o una condición sean verdaderos.
- Son fundamentales para determinar si una prueba ha tenido éxito o ha fallado.
- Si la aserción es verdadera, la prueba se considera exitosa; si la aserción es falsa, la prueba falla y se genera una notificación para indicar que algo en el código probado no funciona como se espera.

Ejemplos

assertEquals(expected, actual): Compara si un valor actual es igual al valor expected. Por ejemplo, assertEquals(5, resultado) verifica si resultado es igual a 5.

assertTrue(condición): Verifica si una condición dada es verdadera. Por ejemplo, assertTrue(valor > 0) verifica si valor es mayor que cero.

assertFalse(condición): Verifica si una condición dada es falsa. Por ejemplo, assertFalse(lista.isEmpty()) verifica si la lista no está vacía.

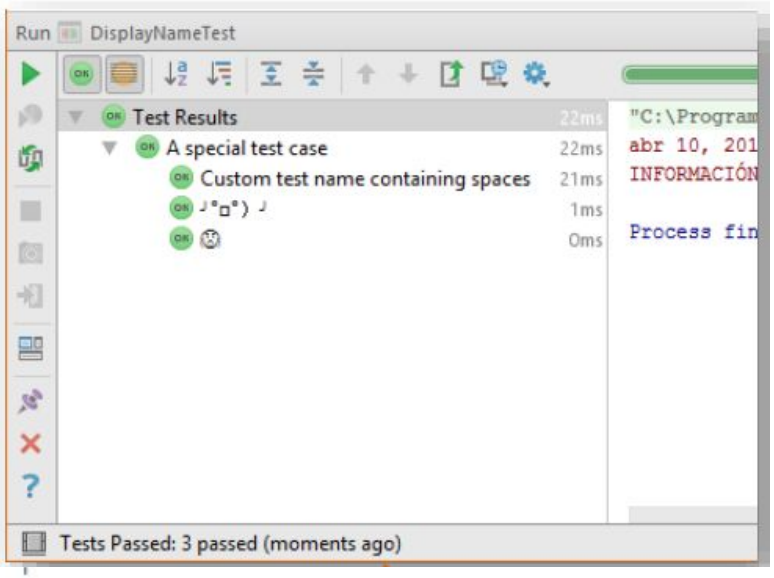
assertNull(objeto): Verifica si un objeto es nulo. Por ejemplo, assertNull(objeto) verifica si objeto es nulo.

assertNotNull(objeto): Verifica si un objeto no es nulo.

assertThrows(excepción, código): Verifica si se arroja una excepción específica cuando se ejecuta el código proporcionado. Por ejemplo, assertThrows(ArithmeticException.class, () -> dividir(10, 0)) verifica si la función dividir arroja una excepción de división por cero.

@DisplayName

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

@DisplayName("A special test case")
class DisplayNameTest {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("╯°□°）╯")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

    @Test
    @DisplayName("😱")
    void testWithDisplayNameContainingEmoji() {
    }

}
```



Anotaciones destacadas

- @BeforeAll
- @AfterAll
- @BeforeEach
- @AfterEach
- @Test

Nuevo Modelo de Programación con JUnit5

- Las clases y métodos de test en JUnit 5 se pueden **etiquetar** usando la anotación @Tag
- Estas etiquetas se pueden usar después para el descubrimiento y ejecución de los test (**filtrado**)

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

@Tag("functional")
class FunctionalTest {

    @Test
    void test1() {
        System.out.println("Functional Test 1");
    }

    @Test
    void test2() {
        System.out.println("Functional Test 2");
    }

}
```

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

@Tag("non-functional")
class NonFunctionalTest {

    @Test
    @Tag("performance")
    @Tag("load")
    void test1() {
        System.out.println("Non-Functional Test 1 (Performance/Load)");
    }

    @Test
    @Tag("performance")
    @Tag("stress")
    void test2() {
        System.out.println("Non-Functional Test 2 (Performance/Stress)");
    }

    @Test
    @Tag("security")
    void test3() {
        System.out.println("Non-Functional Test 3 (Security)");
    }

    @Test
    @Tag("usability")
    void test4() {
        System.out.println("Non-Functional Test 4 (Usability)");
    }

}
```

Filtrar por Tags en Maven

```xml
<!-- Maven Surefire plugin to run tests -->
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${maven-surefire-plugin.version}</version>
            <configuration>
                <properties>
                    <includeTags>functional</includeTags>
                    <excludeTags>non-functional</excludeTags>
                </properties>
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-surefire-provider</artifactId>
                    <version>${junit.platform.version}</version>
                </dependency>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>${junit.jupiter.version}</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

@Disabled

Deshabilitar test a nivel de clase o método

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

class DisabledTest {

    @Disabled
    @Test
    void skippedTest() {
    }

}
```

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

@Disabled("All test in this class will be skipped")
class AllDisabledTest {

    @Test
    void skippedTest1() {
    }

    @Test
    void skippedTest2() {
    }

}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.AllDisabledTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 1, Time elapsed:
0.059 sec - in io.github.bonigarcia.AllDisabledTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 1
```

@RepeatTest

Permite repetir un test un número determinado de veces

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

class SimpleRepeatedTest {

    @RepeatedTest(5)
    void test() {
        System.out.println("Repeated test");
    }

}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.SimpleRepeatedTest
Repeated test
Repeated test
Repeated test
Repeated test
Repeated test
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.11 sec - in io.github.bonigarcia.SimpleRepeatedTest

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

@ParameterizedTest

- Los pasos para implementar un test parametrizado son:

1. Usar la anotación `@ParameterizedTest` para declarar un test como parametrizado
2. Elegir un proveedor de argumentos (*argument provider*)

Arguments provider	Descripción
`@ValueSource`	Usado para especificar un array de valores `String`, `int`, `long`, o `double`
`@EnumSource`	Usado para especificar valores enumerados (`java.lang.Enum`)
`@MethodSource`	Usado para especificar un método estático de la clase que proporciona un `Stream` de valores
`@CsvSource`	Usado para especificar valores separados por coma, esto es, en formato CSV (*comma-separated values*)
`@CsvFileSource`	Usado para especificar valores en formato CSV en un fichero localizado en el classpath
`@ArgumentsSource`	Usado para especificar una clase que implementa el interfaz `org.junit.jupiter.params.provider.ArgumentsProvider`

Ejemplo con @ValueSource

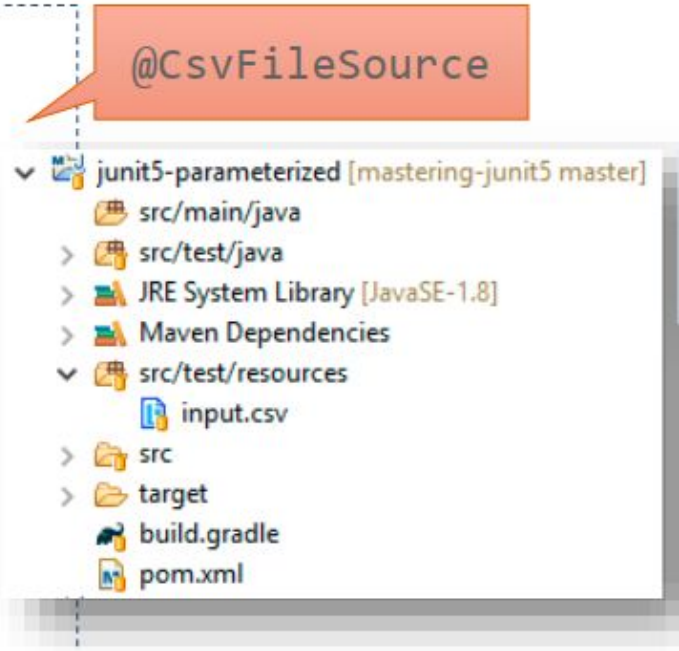
```java
import static org.junit.jupiter.api.Assertions.assertNotNull;

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

class ValueSourcePrimitiveTypesParameterizedTest {

    @ParameterizedTest
    @ValueSource(ints = { 0, 1 })
    void testWithInts(int argument) {
        System.out
                .println("Parameterized test with (int) argument: " + argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(longs = { 2L, 3L })
    void testWithLongs(long argument) {
        System.out.println(
                "Parameterized test with (long) argument: " + argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(doubles = { 4d, 5d })
    void testWithDoubles(double argument) {
        System.out.println(
                "Parameterized test with (double) argument: " + argument);
        assertNotNull(argument);
    }

}
```

@ValueSource

```
T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.ValueSourcePrimitiveTypes
Parameterized test with (int) argument: 0
Parameterized test with (int) argument: 1
Parameterized test with (long) argument: 2
Parameterized test with (long) argument: 3
Parameterized test with (double) argument: 4.0
Parameterized test with (double) argument: 5.0
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time
io.github.bonigarcia.ValueSourcePrimitiveTypesParamete

Results :

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
```

Ejemplo con @CsvFileSource

```java
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

class CsvFileSourceParameterizedTest {

    @ParameterizedTest
    @CsvFileSource(resources = "/input.csv")
    void testWithCsvFileSource(String first, int second) {
        System.out.println("Yet another parameterized test with (String) "
                + first + " and (int) " + second);

        assertNotNull(first);
        assertNotEquals(0, second);
    }

}
```



Referencias

https://maven.apache.org/archetype/project-info.html

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