**JUnit & Mockito**

# Descripción:

JUnit es un framework de pruebas unitarias de Java. Una prueba unitaria es el testeo mediante software (automatizado) de una pequeña parte de una aplicación, por ejemplo, una clase o un método.

Idealmente un caso de prueba es independiente de otros, para lo cual se puede usar código para simular a otro objeto o método.

Mockito es un framework que simplifica la simulación de objetos, permitiendo hacer casos de prueba independientes.

# Dependencias:

JUnit 5:

<dependency>

    <groupId>org.junit.jupiter</groupId>

    <artifactId>junit-jupiter-api</artifactId>

    <version>5.1.0</version>

    <scope>test</scope>

</dependency>

Si estamos usando un IDE que tiene integrada una versión más vieja de JUnit, hay que agregar las siguientes dependencias para poder correr los tests:

<dependency>

    <groupId>org.junit.jupiter</groupId>

    <artifactId>junit-jupiter-engine</artifactId>

    <version>5.1.0</version>

    <scope>test</scope>

</dependency>

<dependency>

    <groupId>org.junit.vintage</groupId>

    <artifactId>junit-vintage-engine</artifactId>

    <version>5.1.0</version>

    <scope>test</scope>

    </dependency>

<dependency>

    <groupId>org.junit.platform</groupId>

    <artifactId>junit-platform-launcher</artifactId>

    <version>1.1.0</version>

    <scope>test</scope>

</dependency>

<dependency>

    <groupId>org.junit.platform</groupId>

    <artifactId>junit-platform-runner</artifactId>

    <version>1.1.0</version>

    <scope>test</scope>

</dependency>

Mockito:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>2.21.0</version>

<scope>test</scope>

</dependency>

# Anotaciones y ejemplos:

## JUnit:

* **@Test:** le indica al framework que el método puede ser ejecutado como prueba unitaria (test case).

@Test

Public void suma\_test(){

AssertEquals(2 , metodosuma(1,1)); // el resultado esperado está siendo comparado con

} // aquel que devuelva el método metodosuma(1,1).

* **@RepeatedTest:** Se usa para hacer test que se repitan X cantidad de veces. Cada iteración se comportará como un @Test independiente, esto significa que las anotaciones que se ejecuten antes o después de cada test se ejecutaran en cada iteración.

@RepeatedTest(10) //la unidad de testeo se repetirá 10 veces.

Public void suma\_test(){

AssertEquals(2 , metodosuma(1,1));

}

* **@BeforeEach:** Lo que este debajo de esta anotación se ejecuta una vez antes de empezar cada test (en JUnit 4 se usaba @Before), se puede usar para reiniciar un objeto necesario para varios tests.

Public class Test {

List listaVacia;

@BeforeEach

Public void initialize(){

listaVacia = new ArrayList();

}

}

* **@BeforeAll:** Esta anotación indica que el método será ejecutado por única vez antes de comenzar cada testeo, útil para realizar inicializaciones. Este método debe ser estático.

Public class Test {

List listaVacia;

@BeforeAll

Public static void initialize(){

listaVacia = new ArrayList();

listaVacia.add(1);

}

}

* **@AfterEach:** Igual que @BeforeEach pero luego de cada test. Un ejemplo de uso es hacer rollback a un cambio en la base de datos.

@AfterEach

void tearDown(TestInfo testInfo) {

System.out.println("Finished..." + testInfo.getDisplayName());

} //getDisplayName() //muestra el nombre asociado al método, para asignárselo, se

//utiliza la anotación @DisplayName(“..”)

* **@AfterAll:** Igual que @BeforeAll pero luego de ejecutar todos los tests.

@AfterAll

void fin(){

System.out.println(“Tests finished”);

}

* **@DisplayName:** Sirve para ponerle un nombre a un caso de prueba.

@Test

@DisplayName(“Test Prueba 1”)

Test method(){}

* **@Disable:** para deshabilitar una prueba, en Junit 4 se utiliza @ignore.

@Ignore

@Test

Test notDoneYet(){}

||

## Mockito:

* **@Mock:** Se utiliza para escribir menos código a la hora de “mockear” objetos, además simplifica la lectura del mismo.

**Sin @Mock:**

List mockedList = mock(List.class);

**Con @Mock:**

@Mock List mockedList;

* **@InyectMocks:** indica un campo al cual se le deberían inyectar mocks.
* **@Captor:** indica un elemento del tipo ArgumentCaptor, utilizado para capturar argumentos y poder realizar pruebas específicas y/o modificaciones con ellos. Un mismo elemento de este tipo puede capturar una o más veces.
* **@Spy:** es un mock parcial, crea una copia de una instancia de un objeto y llama a los métodos reales, por ej.

@Mock List list;

List.add(1); // esto no añade “1” a la lista mockeada

----

List list = new ArrayList();

List spy = spy(list);

Spy.add(1) // llama al metodo add de lista y agrega “1” al objeto spy

* **@MockBean:** Esta anotación añade mocks al SpringApplicationContext y además los inyecta en los campos correspondientes. Se puede utilizar para mockear servicios.

# Métodos más usados:

## JUnit:

**Assert:**

Este método se usa para realizar las comprobaciones que queremos en nuestros tests, existen distintas variantes de assert como por ej assertNotNull, AssertThat, AssertEquals, etc.

assertEquals(60d,suma(30d,30d));

assertNotNull( traerInformacion() );

assertThrows(ArithmeticException.class,

()->{

calculatorService.operate(mock);

});

## Mockito:

**When:**

Este método se usa para mockear resultados de otros métodos (ajenos a lo que queramos testear)

when(calculatorService.operate(myQueue)).thenReturn(3d);

// En este caso decimos que cuando calculatorService ejecute el método operate recibiendo myQueue como parámetro este método devuelva un 3 double, sin importar el contenido de myQueue. Este método fue testeado con valores reales en su clase correspondiente.

# Links útiles:

<http://static.javadoc.io/org.mockito/mockito-core/2.21.0/org/mockito/Mockito.html#1>

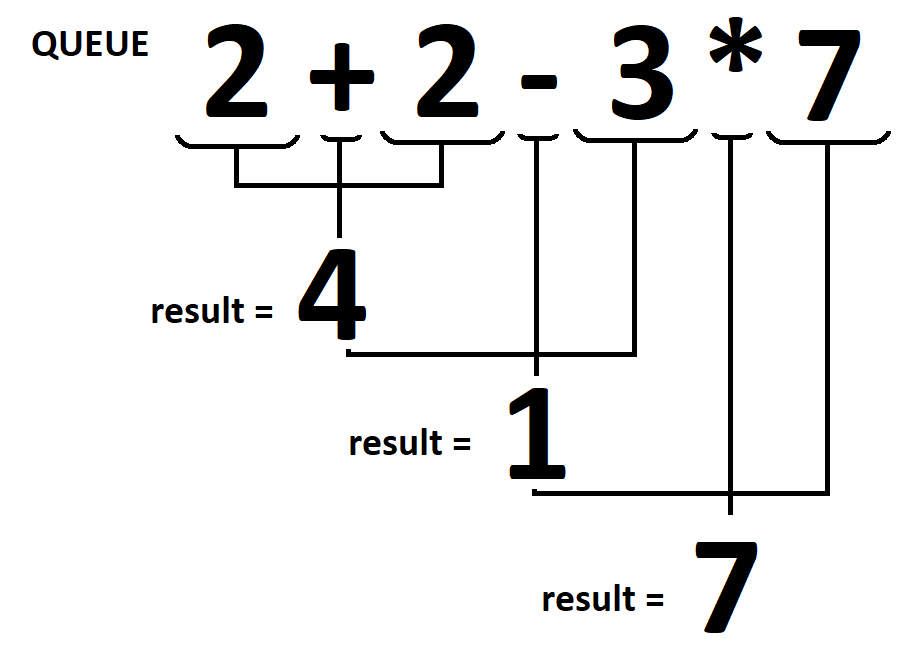
<https://junit.org/junit5/>

**Calculadora**

En el siguiente ejemplo utilizamos los frameworks ya comentados para implementar una calculadora y sus respectivas unidades de prueba.

El modelo principal consiste en una petición Http que envia al programa un JSON con una coleccion de Strings, esta colección sera interpretada por el programa como una Queue para resolver las operaciones matemáticas pedidas.

El proyecto cuenta con dos carpetas de paquetes principales, Source Packages y Test Packages. Dentro de Source packages se encuentran la aplicación, el controlador y el servicio, y en Test Packages tenemos las pruebas correspondientes a cada uno.



El programa opera de izquierda a derecha, no tiene en cuenta las prioridades de operaciones matemáticas. Dentro del programa, se encuentran los algoritmos que realizan las validaciones sobre la Queue, ya sea un tamaño inválido, o la data incorrecta.

Los métodos de prueba testSum (suma), testSub (substracción), testProd (producto) y testDiv (división) funcionan de la misma manera, se crea una Queue con datos y se valida que el resultado del respectivo servicio es correcto, utilizando AssertEquals. Para el caso de la división por cero, se utiliza el método assertThrows llamando al servicio y probando que este devuelve una excepción aritmética.

La calculadora retorna un tipo de dato Opcional<Double>, esto es porque el usuario puede enviar una operación invalida o una Queue de datos no procesables. Si la Queue fue procesada correctamente, el resultado se encontrará dentro del opcional.

El programa realiza una validación previa sobre la queue, verificando que esta es del tamaño correcto y que sus datos son procesables. Deberá contar con un número impar de datos, y estos deben estar intercalados entre número y operación. El paquete de prueba cuenta con cuatro módulos que validan este método, la primera utiliza una queue válida y verifica que el opcional cuenta con un dato, las otras tres chequean que el servicio esta retornando una expeción de argumentos (Illegal argument exception), pasando cada una como parámetro una queue de tamaño inválido, una queue vacía y una queue con datos erróneos.

El paquete además cuenta con cuatro métodos que emplean la anotación @RepeatedTest, dentro de ellas es crean dos números aleatorios y se verifica que cuando son procesados con una operación, la calculadora retorna algún dato (para este tipo de prueba no nos interesa si el dato es el correcto, solamente que los argumentos son procesados).