

# ENTORNOS DE DESARROLLO

## TEMA 4:

### JUNIT

Desarrollo de Aplicaciones Multiplataforma



**Curso 2021/2022**

# Índice

Índice .....	2
1. Pruebas unitarias .....	3
2. JUnit.....	3
2.1. Crear una clase de pruebas.....	4
2.2. Comenzando a configurar la prueba .....	9
2.3. BeforeEach y AfterEach .....	12
2.4. Tipos de Asserts .....	14
2.5. assertThrows y Timeout.....	17
2.6. BeforeClass y AfterClass.....	20
2.7. Pruebas parametrizadas .....	22
2.8. Cobertura del código .....	22

# 1.Pruebas unitarias

Las pruebas unitarias consisten en la verificación de unidades del software de forma aislada, es decir, probar el correcto funcionamiento de una unidad de código.

Una **unidad de código** es considerada una unidad de programa, como una función o método de una clase que es invocada desde fuera y que puede invocar otras unidades. Es por ello por lo que hay que probar que cada unidad funcione separada de las demás unidades de código.

Estas pruebas suelen ser realizadas por los desarrolladores, ya que es muy recomendable conocer el código fuente del programa y generalmente se realizarán pruebas de caja blanca o se analizará el código para comprobar que cumple con las especificaciones del componente, no obstante, no solo se realizarán pruebas de la estructura del código, sino que también se generarán casos de prueba funcionales para comprobar el funcionamiento del componente.

JUnit es un entorno que nos permite crear pruebas unitarias.

## 2.JUnit

Es una biblioteca muy popular que está incluida en la mayoría de los entornos de desarrollo.

Realizar pruebas Unitarias con JUnit es mejor que hacer pruebas en un código Main ya que:

- El código de prueba que se hace en el código Main es molesto, ya que hay que comentarlo y descomentarlo constantemente o directamente añadirlo y eliminarlo.
- En proyectos grandes, en los que trabaja mucha gente, crear pruebas en el método Main puede ser un caos, ya que, si se realizan muchas pruebas, será difícil distinguir qué es código funcional y qué es código a eliminar.
- Puede que, si realizamos pruebas en el método Main, estas acaben apareciendo en el código final del programa.
- Las pruebas que se hagan en el método Main no están automatizadas y puede que estemos interpretando mal su resultado.

No es necesario instalar JUnit en Eclipse ya que el propio entorno contiene las librerías necesarias, pero si será necesario importarlas en las clases que se usen.

Para poder aprender a usar esta tecnología vamos a ir probándola sobre un proyecto calculadora para acompañar las explicaciones con práctica.

Por tanto, vamos a crear un proyecto de Java en Eclipse que represente una calculadora y, de momento, tenga una única clase que tenga la siguiente forma:

```
1 package edd.tema4.ejemplo.calculadora.fuente;
2
3 public class Calculadora {
4
5     static int suma (int a, int b) {
6         return a + b;
7     }
8
9     static int resta (int a, int b) {
10        return a - b;
11    }
12 }
```

Con los conocimientos que tenemos hasta ahora, para comprobar si esta clase funciona bien, lo normal sería que hiciéramos lo siguiente.

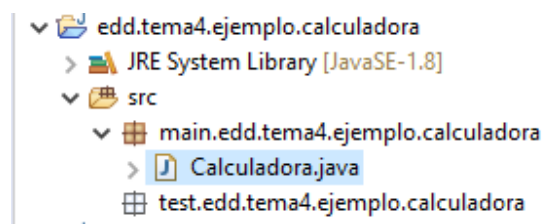
```
1 package edd.tema4.ejemplo.calculadora.fuente;
2
3 public class Calculadora {
4
5     static int suma (int a, int b) {
6         return a + b;
7     }
8
9     static int resta (int a, int b) {
10        return a - b;
11    }
12
13    public static void main(String[] args) {
14        int a = 5, b = 4;
15
16        int resultadoSuma = suma(a,b);
17        int resultadoResta = resta(a,b);
18
19        System.out.println("El resultado de la suma es " + resultadoSuma + " y el de la resta es " + resultadoResta + ".");
20    }
21 }
22 }
```

No obstante, esto tiene las desventajas ya mencionadas anteriormente.

Realmente, realizar las pruebas en el método main no es correcto ni funcional, para realizarlas vamos a aprender a usar la tecnología JUnit.

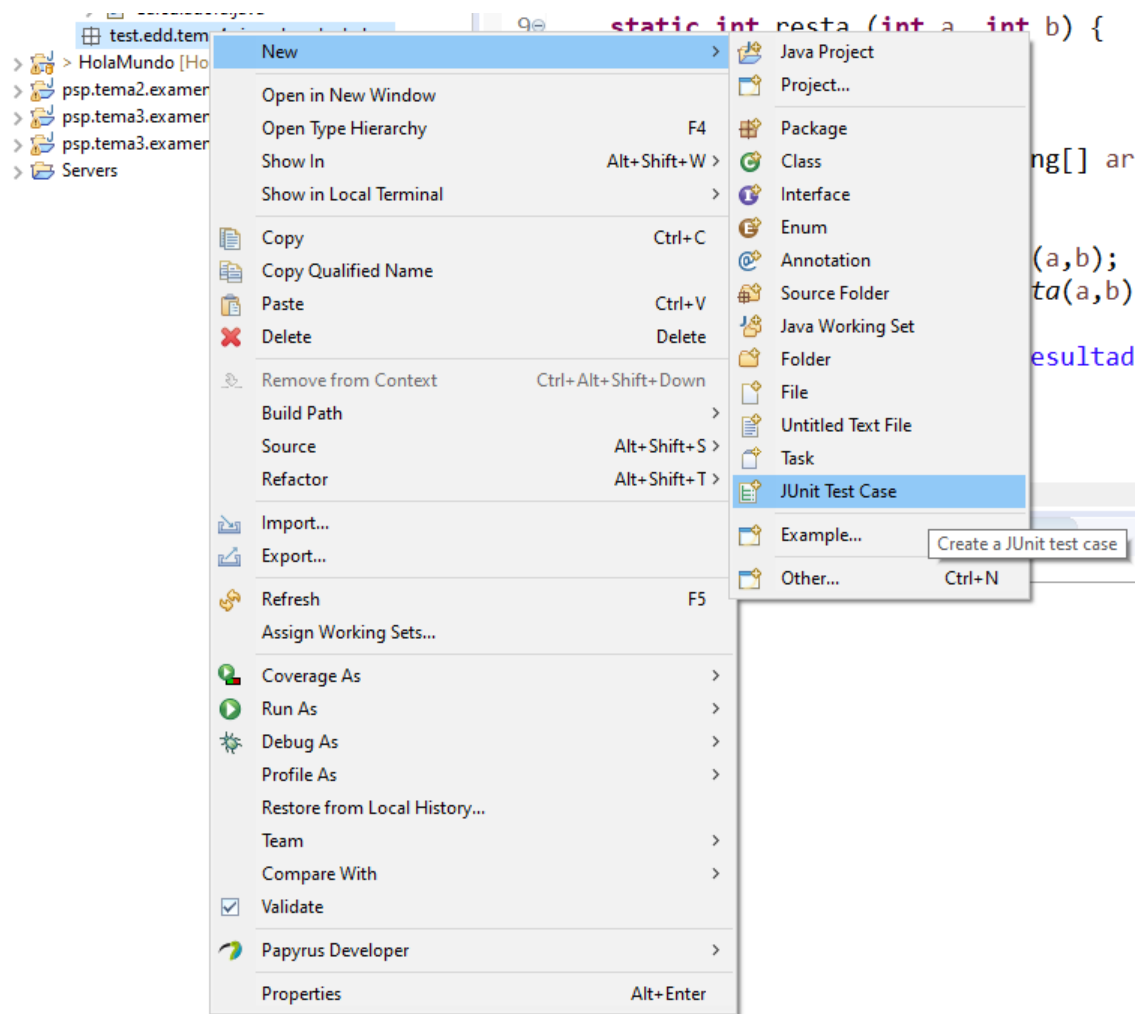
## 2.1. Crear una clase de pruebas

Para distribuir bien el código es adecuado crear un paquete en el que se almacenará el código funcional y otro en el que se almacenarán las pruebas.



Si el proyecto crece en tamaño estos paquetes se pueden dividir en otros paquetes más pequeños, pero la ubicación de las clases y sus pruebas tienen que ser análogas con la salvedad del comienzo del paquete (main o test).

Vamos a crear una clase de pruebas en el paquete adecuado.



Lo normal es llamar a las clases de prueba con el nombre de la clase a probar más la palabra test.

**New JUnit Test Case**

**JUnit Test Case**

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test
 ☐ New JUnit 4 test
 ☒ New JUnit Jupiter test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass()
 ☐ tearDownAfterClass()  
☐ setUp()
 ☐ tearDown()  
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

Si en el formulario anterior ponemos la clase sobre la que se aplicará el test se creará la clase con pruebas para todos los métodos que se indiquen.

**New JUnit Test Case**

**JUnit Test Case**

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test
 ☐ New JUnit 4 test
 ☒ New JUnit Jupiter test

Source folder:

Package:

Name:

Superclass:

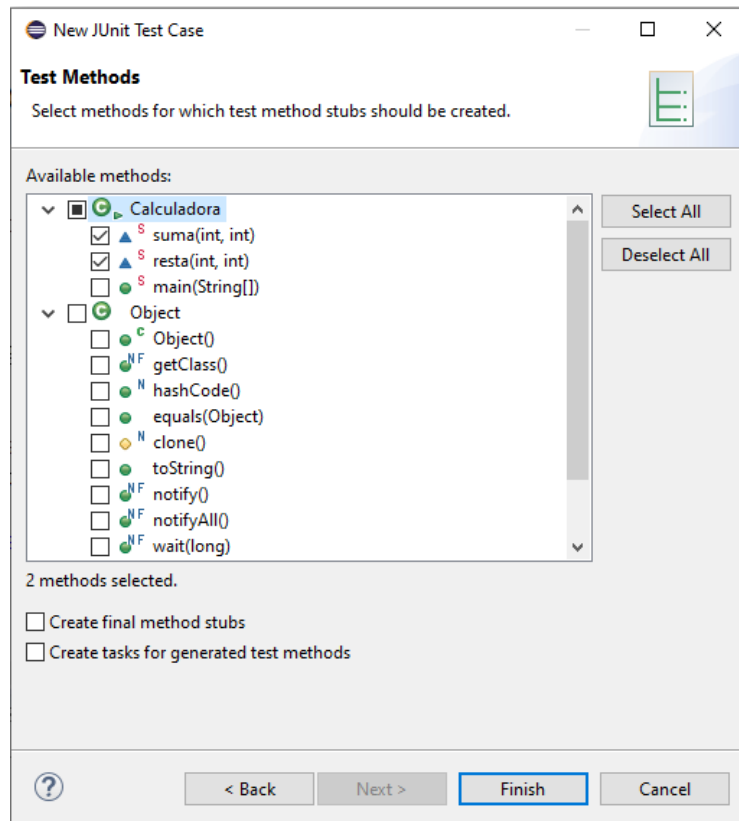
Which method stubs would you like to create?

☐ setUpBeforeClass()
 ☐ tearDownAfterClass()  
☐ setUp()
 ☐ tearDown()  
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

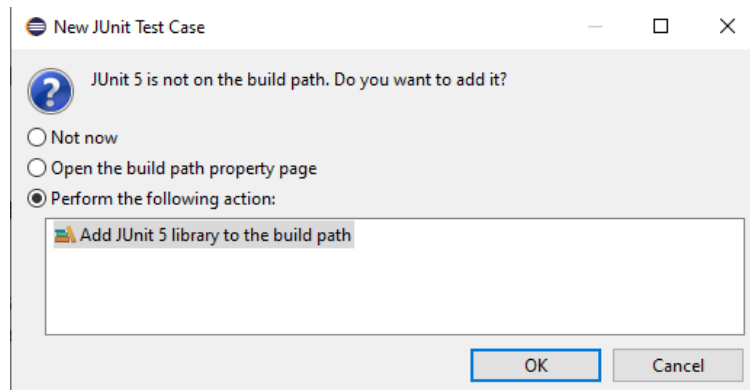
☐ Generate comments

Class under test:



El entorno me preguntará si quiero añadir JUnit como biblioteca para poder utilizarlo.

Es importante tener en cuenta que JUnit no forma parte de Java, y, por tanto, tendremos que añadirlo siempre antes de usarlo.



Se nos crea la siguiente clase

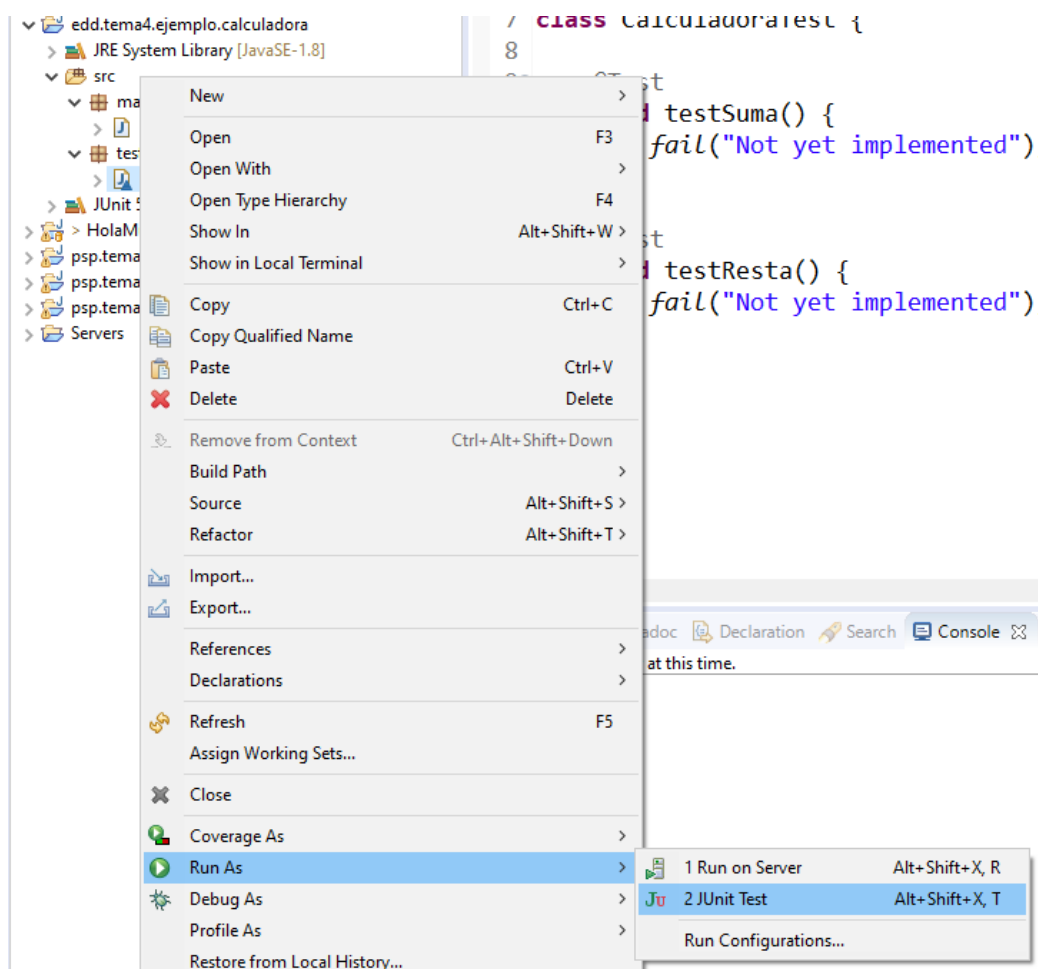
```

1 package test.edd.tema4.ejemplo.calculadora;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5
6
7 class CalculadoraTest {
8
9     @Test
10     void testSuma() {
11         fail("Not yet implemented");
12     }
13
14     @Test
15     void testResta() {
16         fail("Not yet implemented");
17     }
18
19 }
20

```

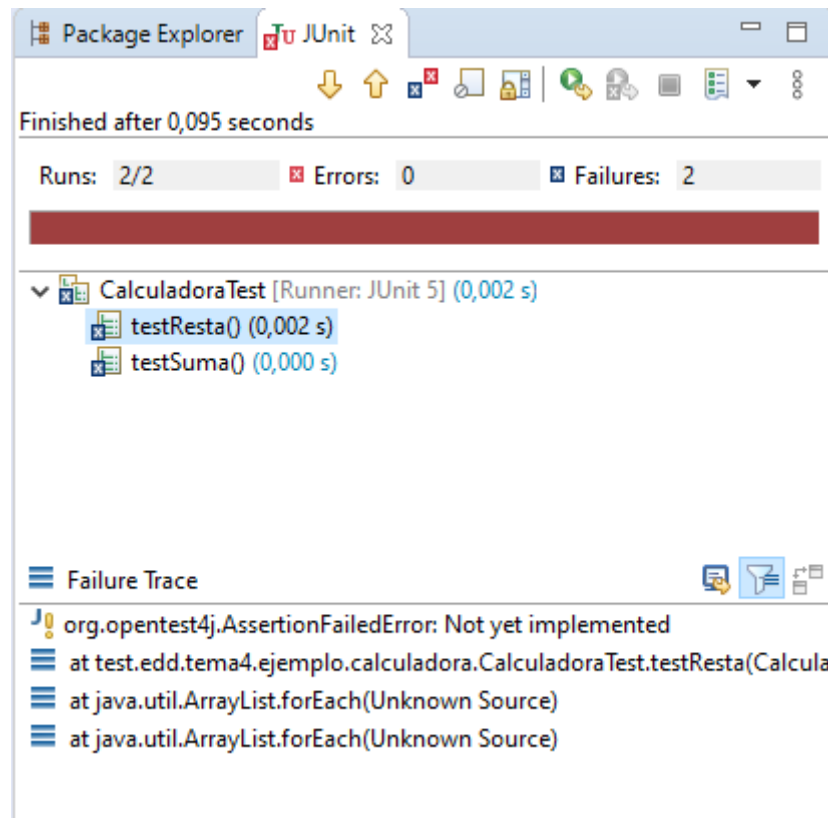
La anotación **test** indica que el método que tiene debajo es una prueba y, de esta manera, JUnit puede reconocer rápidamente métodos que son pruebas para ejecutarlos como tal.

Vamos a ejecutar la prueba unitaria.



Aparece en el entorno una vista nueva que nos muestra la información de la prueba.





La prueba va a fallar porque tiene una llamada al método fail directamente y no hemos configurado la prueba.

## 2.2. Comenzando a configurar la prueba

La biblioteca de JUnit, como es de esperar, tiene funciones y clases propias que permiten realizar las pruebas unitarias.

La más utilizada y la más básica se llama **assertEquals** y consiste en una función que se utiliza para ver si dos cosas son iguales y de esta manera comparar resultado esperado con resultado obtenido.

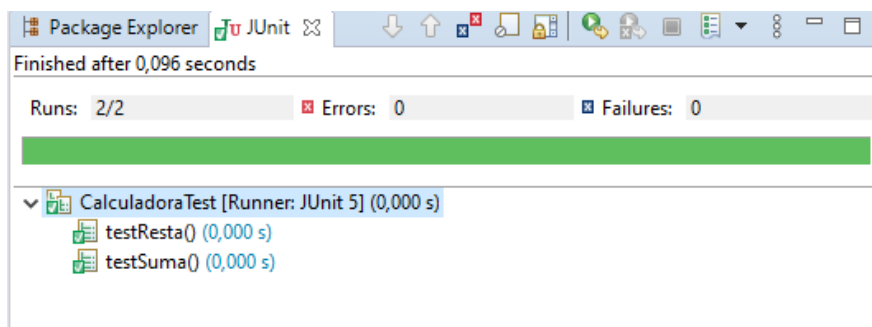
Por tanto, si queremos comprobar que las funciones que hemos hecho de la calculadora funcionan bien, tendremos que hacer unas pruebas parecidas a estas.

```

Calculadora.java  CalculadoraTest.java
1 package test.edd.tema4.ejemplo.calculadora;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5
6
7
8
9 class CalculadoraTest {
10
11     @Test
12     void testSuma() {
13         int resultado = Calculadora.suma(2,7);
14         int esperado = 9; // 2 + 7 = 9
15         assertEquals(esperado, resultado);
16     }
17
18     @Test
19     void testResta() {
20         int resultado = Calculadora.rest(4,2);
21         int esperado = 2; // 4 - 2 = 2
22         assertEquals(esperado, resultado);
23     }
24 }

```

Como se puede ver el resultado de las pruebas es satisfactorio.



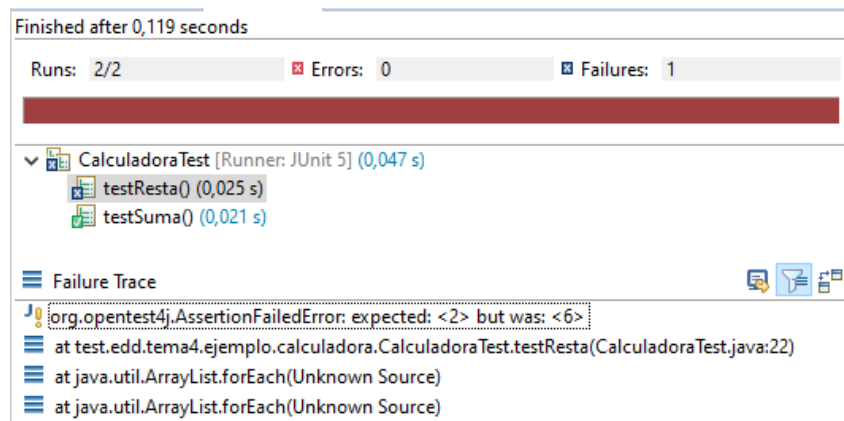
Si hubiera habido algún error en la implementación (como por ejemplo este)

```

Calculadora.java  CalculadoraTest.java
1 package main.edd.tema4.ejemplo.calculadora;
2
3 public class Calculadora {
4
5     public static int suma (int a, int b) {
6         return a + b;
7     }
8
9     public static int resta (int a, int b) {
10        return a + b;
11    }
12 }

```

La prueba devolvería el error.



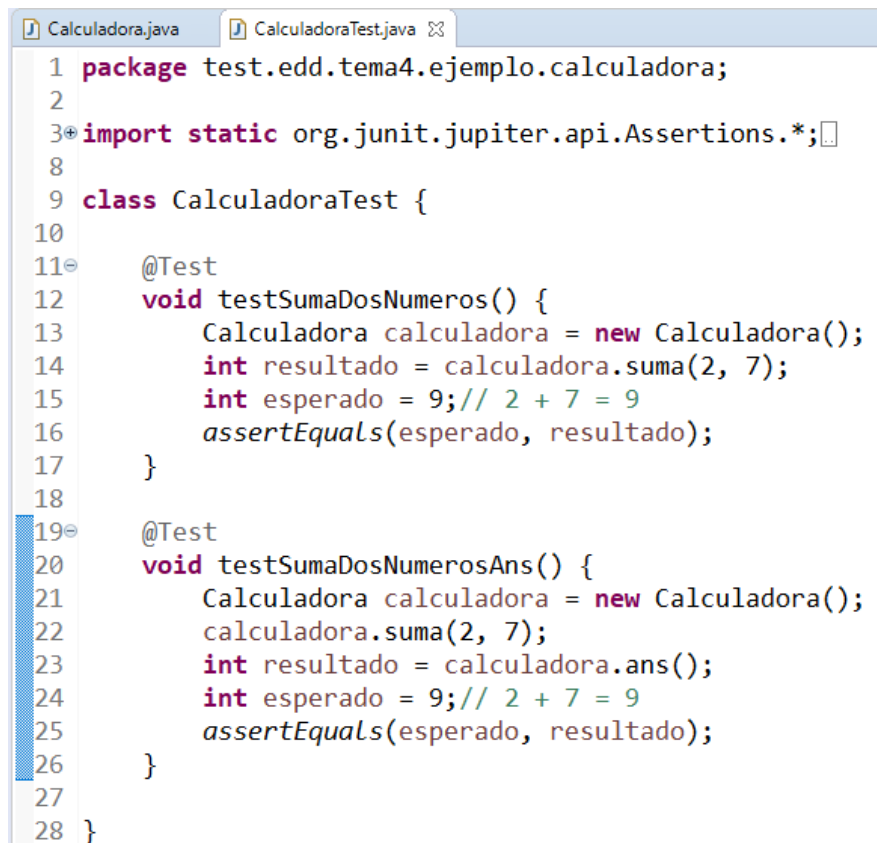
Para poder continuar explicando funciones más complejas de JUnit, vamos a modificar la clase Calculadora.

Hay que tener en cuenta que una prueba de JUnit puede devolver:

- Un tic verde que indica que el resultado de la prueba es el esperado.
- Una cruz azul que indica que el resultado de la prueba no es el esperado.
- Una cruz roja que indica error.

```
Calculadora.java x CalculadoraTest.java
1 package main.edd.tema4.ejemplo.calculadora;
2
3 public class Calculadora {
4
5     private int ans;
6
7     public Calculadora() {
8         ans = 0;
9     }
10
11     public int suma (int a, int b) {
12         ans = a + b;
13         return ans;
14     }
15
16     public int suma (int num) {
17         ans += num;
18         return ans;
19     }
20
21     public int resta (int a, int b) {
22         ans = a - b;
23         return ans;
24     }
25
26     public int ans() {
27         return ans;
28     }
29
30     public void clear () {
31         ans = 0;
32     }
33 }
```

Podemos empezar creando estas dos pruebas.



```
1 package test.edd.tema4.ejemplo.calculadora;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5
6
7
8
9 class CalculadoraTest {
10
11     @Test
12     void testSumaDosNumeros() {
13         Calculadora calculadora = new Calculadora();
14         int resultado = calculadora.suma(2, 7);
15         int esperado = 9; // 2 + 7 = 9
16         assertEquals(esperado, resultado);
17     }
18
19     @Test
20     void testSumaDosNumerosAns() {
21         Calculadora calculadora = new Calculadora();
22         calculadora.suma(2, 7);
23         int resultado = calculadora.ans();
24         int esperado = 9; // 2 + 7 = 9
25         assertEquals(esperado, resultado);
26     }
27
28 }
```

## 2.3. BeforeEach y AfterEach

Podemos ver que siempre, antes de hacer las pruebas, tenemos que inicializar la calculadora, y cuando hagamos el resto de las pruebas nos daremos cuenta de que también hay que inicializar la calculadora para poder ejecutarla.

La anotación **@BeforeEach** nos permite determinar una función que se ejecutará antes de todas las pruebas.

**@BeforeEach** funciona de una manera muy parecida a **@Before**, no obstante, está pensada también para **@ParametrizedTest** (pruebas parametrizadas, las veremos más adelante) y no solo para **@Test**.

Para entender cómo funciona vamos a hacer la siguiente prueba.

```

class CalculadoraTest {
    Calculadora calculadora;

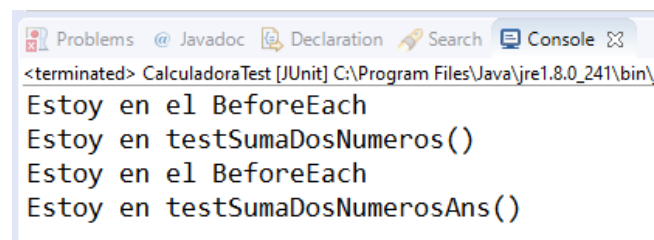
    @BeforeEach
    public void antes() {
        System.out.println("Estoy en el BeforeEach");
        calculadora = new Calculadora();
    }

    @Test
    void testSumaDosNumeros() {
        System.out.println("Estoy en testSumaDosNumeros()");
        int resultado = calculadora.suma(2, 7);
        int esperado = 9; // 2 + 7 = 9
        assertEquals(esperado, resultado);
    }

    @Test
    void testSumaDosNumerosAns() {
        System.out.println("Estoy en testSumaDosNumerosAns()");
        calculadora.suma(2, 7);
        int resultado = calculadora.ans();
        int esperado = 9; // 2 + 7 = 9
        assertEquals(esperado, resultado);
    }
}

```

Si le doy al run, se verá lo siguiente.



```

<terminated> CalculadoraTest [JUnit] C:\Program Files\Java\jre1.8.0_241\bin\
Estoy en el BeforeEach
Estoy en testSumaDosNumeros()
Estoy en el BeforeEach
Estoy en testSumaDosNumerosAns()

```

Esto nos permite no tener código duplicado.

De la misma manera existe la anotación **@AfterEach** (también, de manera análoga a como se explicó anteriormente, existe la anotación **@After**), que se ejecutará siempre después de hacer una prueba. Se suele utilizar para cerrar conexiones o destruir objetos.

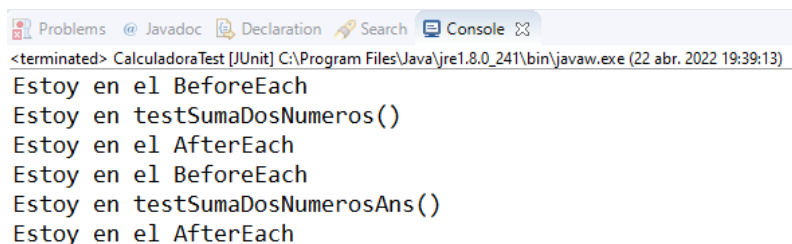
Por ejemplo, después de hacer una prueba podemos querer que se ejecute el método clear para que la calculadora quede limpia.

```

11 class CalculadoraTest {
12     Calculadora calculadora;
13
14     @BeforeEach
15     public void antes() {
16         System.out.println("Estoy en el BeforeEach");
17         calculadora = new Calculadora();
18     }
19
20     @AfterEach
21     public void despues() {
22         System.out.println("Estoy en el AfterEach");
23         calculadora.clear();
24     }
25
26     @Test
27     void testSumaDosNumeros() {
28         System.out.println("Estoy en testSumaDosNumeros()");
29         int resultado = calculadora.suma(2, 7);
30         int esperado = 9; // 2 + 7 = 9
31         assertEquals(esperado, resultado);
32     }
33
34     @Test
35     void testSumaDosNumerosAns() {
36         System.out.println("Estoy en testSumaDosNumerosAns()");
37         calculadora.suma(2, 7);
38         int resultado = calculadora.ans();
39         int esperado = 9; // 2 + 7 = 9
40         assertEquals(esperado, resultado);
41     }

```

El output obtenido es el siguiente.



```

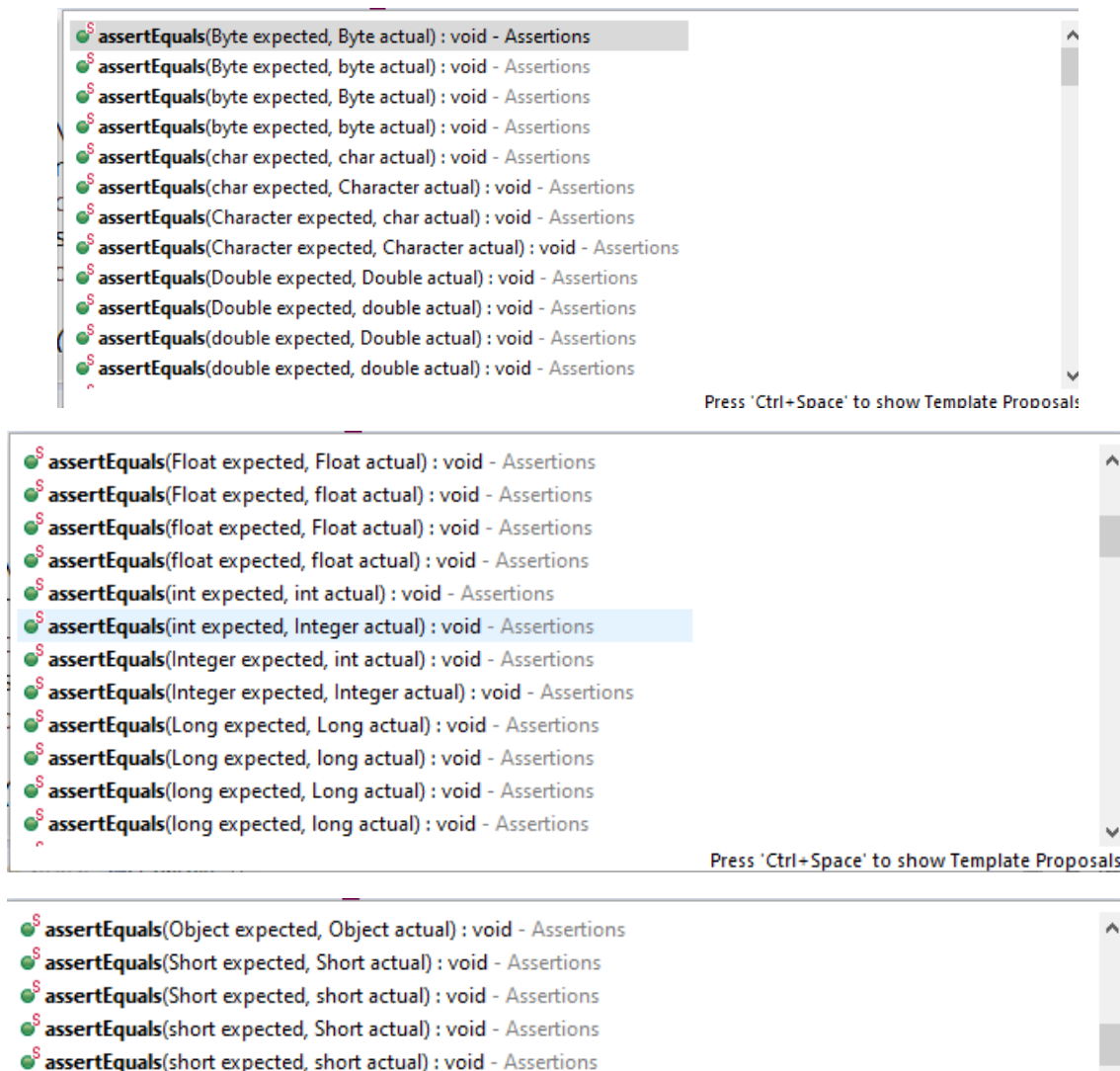
Problems Javadoc Declaration Search Console
<terminated> CalculadoraTest [JUnit] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (22 abr. 2022 19:39:13)
Estoy en el BeforeEach
Estoy en testSumaDosNumeros()
Estoy en el AfterEach
Estoy en el BeforeEach
Estoy en testSumaDosNumerosAns()
Estoy en el AfterEach

```

## 2.4. Tipos de Asserts

Hasta ahora solo hemos utilizado el **assertEquals()** pero existe una gran variedad de Asserts que nos permitirán hacer muchas pruebas que permite contemplar todos los resultados posibles desde el punto de vista del tipo de dato.

Hasta ahora hemos utilizado el **assertEquals()** que pide como parámetro dos números enteros, pero existe una función análoga casi para cualquier tipo de dato.



El de **assertEquals de Object** llama al método `equals` del objeto.

El **assertEquals de double y de float** que tiene dos parámetros no se suele usar, se usa el que tiene tres, ya que la variabilidad de los decimales es impredecible y puede dificultar las pruebas. El **assertEquals de double y de float** que tiene tres parámetros, pide un último dato llamado `delta` que permite establecer un margen de error, lo que facilita la comparación de decimales.

```
assertEquals(double expected, double actual, double delta) : void - Assertions  
assertEquals(float expected, float actual, float delta) : void - Assertions
```

Hay otros métodos con un primer parámetro que es un `String` y que permite mostrar un mensaje personalizado cuando la comprobación del `assert` no da el resultado esperado.

**AssertArray** se comporta de forma parecida, pero con `Arrays`.

```

S assertArrayEquals(boolean[] expected, boolean[] actual) : void - Assertions
S assertArrayEquals(byte[] expected, byte[] actual) : void - Assertions
S assertArrayEquals(char[] expected, char[] actual) : void - Assertions
S assertArrayEquals(double[] expected, double[] actual) : void - Assertions
S assertArrayEquals(float[] expected, float[] actual) : void - Assertions
S assertArrayEquals(int[] expected, int[] actual) : void - Assertions
S assertArrayEquals(long[] expected, long[] actual) : void - Assertions
S assertArrayEquals(Object[] expected, Object[] actual) : void - Assertions
S assertArrayEquals(short[] expected, short[] actual) : void - Assertions
S assertArrayEquals(boolean[] expected, boolean[] actual, String message) : void - Assertions
S assertArrayEquals(boolean[] expected, boolean[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(byte[] expected, byte[] actual, String message) : void - Assertions
S assertArrayEquals(byte[] expected, byte[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(char[] expected, char[] actual, String message) : void - Assertions
S assertArrayEquals(char[] expected, char[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(double[] expected, double[] actual, double delta) : void - Assertions
S assertArrayEquals(double[] expected, double[] actual, String message) : void - Assertions
S assertArrayEquals(double[] expected, double[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(float[] expected, float[] actual, float delta) : void - Assertions
S assertArrayEquals(float[] expected, float[] actual, String message) : void - Assertions
S assertArrayEquals(float[] expected, float[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(int[] expected, int[] actual, String message) : void - Assertions
S assertArrayEquals(int[] expected, int[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(long[] expected, long[] actual, String message) : void - Assertions
S assertArrayEquals(long[] expected, long[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(Object[] expected, Object[] actual, String message) : void - Assertions
S assertArrayEquals(Object[] expected, Object[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(short[] expected, short[] actual, String message) : void - Assertions
S assertArrayEquals(short[] expected, short[] actual, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(double[] expected, double[] actual, double delta, String message) : void - Assertions
S assertArrayEquals(double[] expected, double[] actual, double delta, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(float[] expected, float[] actual, float delta, String message) : void - Assertions
S assertArrayEquals(float[] expected, float[] actual, float delta, Supplier<String> messageSupplier) : void - Assertions
S assertArrayEquals(String message, boolean[] expecteds, boolean[] actuals) : void - org.junit.Assert
S assertArrayEquals(String message, byte[] expecteds, byte[] actuals) : void - org.junit.Assert
S assertArrayEquals(String message, char[] expecteds, char[] actuals) : void - org.junit.Assert
S assertArrayEquals(String message, int[] expecteds, int[] actuals) : void - org.junit.Assert
S assertArrayEquals(String message, long[] expecteds, long[] actuals) : void - org.junit.Assert
S assertArrayEquals(String message, Object[] expecteds, Object[] actuals) : void - org.junit.Assert
S assertArrayEquals(String message, short[] expecteds, short[] actuals) : void - org.junit.Assert
S assertArrayEquals(String message, double[] expecteds, double[] actuals, double delta) : void - org.junit.Assert
S assertArrayEquals(String message, float[] expecteds, float[] actuals, float delta) : void - org.junit.Assert

```

**AssertFalse** y **AssertTrue** se utilizan para comprobar si una determinada condición se cumple o no.

```

S assertFalse(boolean condition) : void - Assertions
S assertFalse(BooleanSupplier booleanSupplier) : void - Assertions
S assertFalse(boolean condition, String message) : void - Assertions
S assertFalse(boolean condition, Supplier<String> messageSupplier) : void - Assertions
S assertFalse(BooleanSupplier booleanSupplier, String message) : void - Assertions
S assertFalse(BooleanSupplier booleanSupplier, Supplier<String> messageSupplier) : void - Assertions
S assertFalse(String message, boolean condition) : void - org.junit.Assert

```



```

§ assertTrue(boolean condition) : void - Assertions
§ assertTrue(BooleanSupplier booleanSupplier) : void - Assertions
§ assertTrue(boolean condition, String message) : void - Assertions
§ assertTrue(boolean condition, Supplier<String> messageSupplier) : void - Assertions
§ assertTrue(BooleanSupplier booleanSupplier, String message) : void - Assertions
§ assertTrue(BooleanSupplier booleanSupplier, Supplier<String> messageSupplier) : void - Assertions
§ assertTrue(String message, boolean condition) : void - org.junit.Assert

```

**AssertNotEquals** funciona igual que **AssertEquals** pero en vez de comprobar que los valores son iguales comprueba que no lo sean.

**AssertNotNull** se utiliza para comprobar que algo no sea nulo.

```

§ assertNotNull(Object actual) : void - Assertions
§ assertNotNull(Object actual, String message) : void - Assertions
§ assertNotNull(Object actual, Supplier<String> messageSupplier) : void - Assertions
§ assertNotNull(String message, Object object) : void - org.junit.Assert

```

**AssertNull** comprueba que el resultado sea nulo.

```

§ assertNull(Object actual) : void - Assertions
§ assertNull(Object actual, String message) : void - Assertions
§ assertNull(Object actual, Supplier<String> messageSupplier) : void - Assertions
§ assertNull(String message, Object object) : void - org.junit.Assert

```

## 2.5. assertThrows y Timeout

En algunas ocasiones no queremos probar tan solo un resultado, queremos evaluar también el comportamiento.

Por ejemplo, **assertThrows** lo que hace es decirle al método que espera encontrarse una excepción y, por tanto, si no salta esa excepción la prueba dará error.

Es muy útil para comprobar que ciertas excepciones se controlan de manera correcta.

Para probarlo, vamos a crear en la calculadora la función dividir.

```

public int dividir (int a, int b) {
    ans = a/b;
    return ans;
}

```

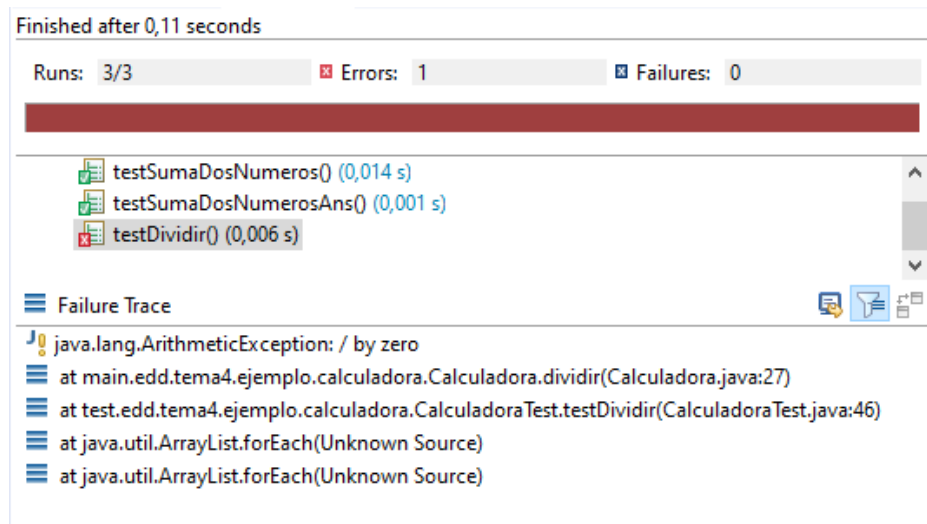
Y vamos a añadir una prueba de división.

```

42
43 @Test
44 void testDividir() {
45     System.out.println("Estoy en testDividir()");
46     calculadora.dividir(5, 0);
47 }

```

Como sabéis, 5 no se puede dividir entre 0 y, por tanto, la prueba dará error.



Vamos a controlar la excepción en el método.

```

public int dividir (int a, int b) {
    if(b== 0)
        throw new ArithmeticException("No puedes dividir por 0");
    ans = a/b;
    return ans;
}

```

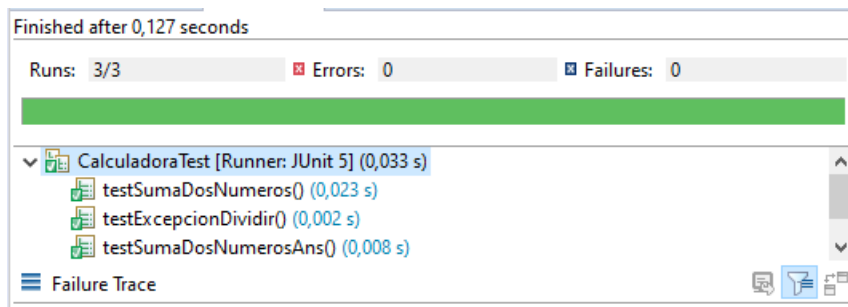
Si yo ahora quiero comprobar que cuando divido por 0 salta esa excepción, tendré que hacer lo siguiente.

```

@Test
void testExcepcionDividir() {
    System.out.println("Estoy en testExcepcionDividir()");
    assertThrows(ArithmeticException.class, ()->{calculadora.dividir(5, 0);});
}

```

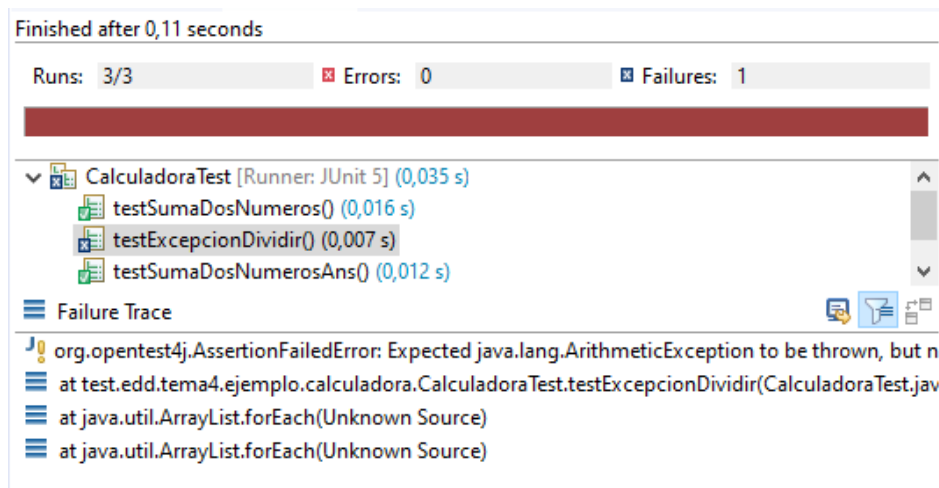
Ahora la prueba permitirá comprobar que esa excepción está controlada.



Si, en este caso, cambiamos la prueba y no se lanza la excepción.

```
@Test
void testExcepcionDividir() {
    System.out.println("Estoy en testExcepcionDividir()");
    assertThrows(ArithmeticException.class, ()->{calculadora.dividir(5, 2);});
}
```

La prueba dará error.



Existe otra anotación, llamada **@Timeout** cuyo objetivo es que la prueba falle si un método tarda más tiempo de lo normal en hacer la prueba.

Este método es muy útil en funciones con pruebas de paralelización, optimización o funciones con conexiones a fuentes de datos online.

Imaginad que creamos un método en la calculadora que ejecuta un algoritmo, y, por lo que sea, no nos ha salido bien y entra tarda más de lo deseado.

```

public void algoritmo() {

    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}

```

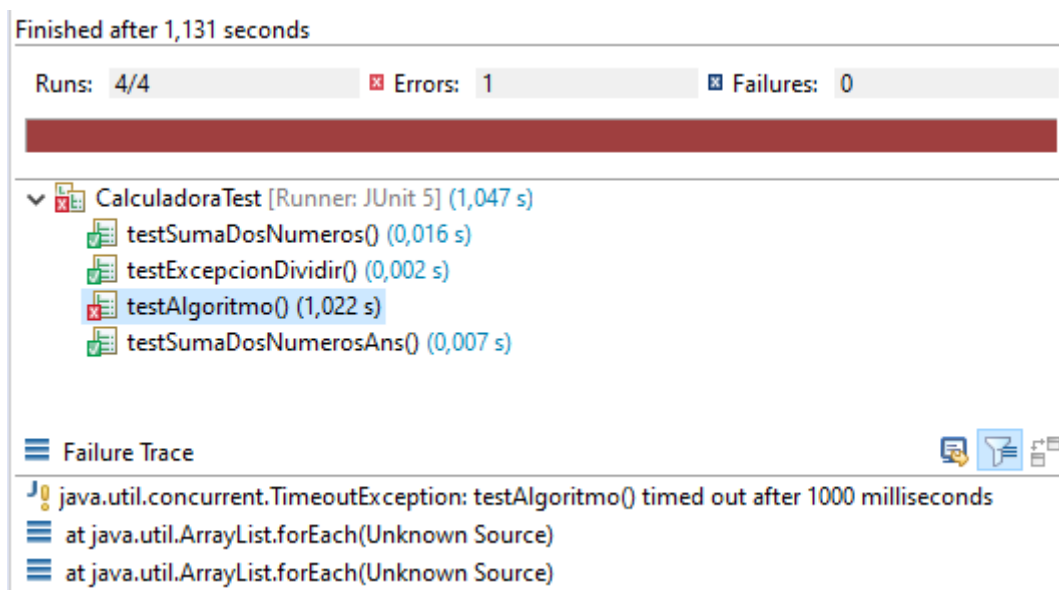
JUnit nos permite decir que si la ejecución del algoritmo tarda más de 1000 milisegundos se cancele la prueba del mismo.

```

@Test
@Timeout(value = 1000, unit = TimeUnit.MILLISECONDS)
void testAlgoritmo() {
    calculadora.algoritmo();
}

```

Aquí se puede ver como se obtiene el error de la prueba.



## 2.6. BeforeClass y AfterClass

Del mismo modo que en JUnit 5 se implementó **@BeforeEach** y **@AfterEach** para complementar el funcionamiento de **@Before** y **@After**, también se implementó **@BeforeAll** y **@AfterAll** para complementar **@BeforeClass** y **@AfterClass**.

Se comportan igual que **@BeforeEach** y **@AfterEach** pero, en este caso, solo se van a ejecutar una vez, al principio del todo y al final.

**@BeforeAll** se ejecutará antes que el primer **@BeforeEach**, y **@AfterAll** se ejecutará después del último **@AfterEach**.

Realmente, lo que estamos haciendo hasta ahora, es instanciar un objeto calculadora en cada prueba, y esto no es eficiente, ya que un mismo objeto valdría para todas las pruebas, solo tendríamos que asegurarnos de que ans llega limpio.

Por tanto, ahora podremos hacer lo siguiente:

```
static Calculadora calculadora;

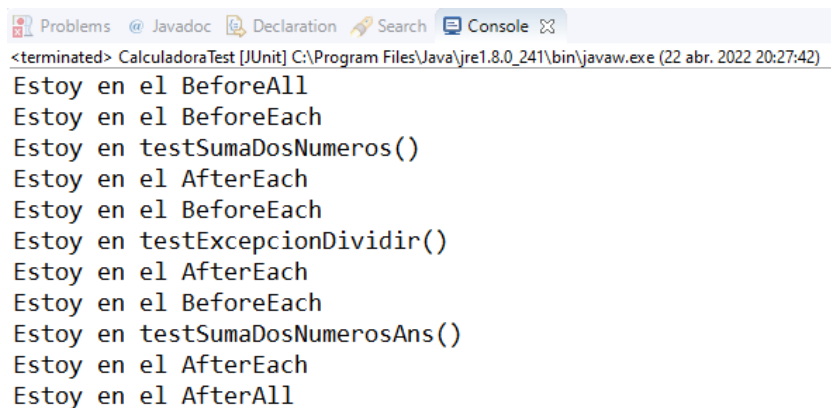
@BeforeAll
public static void antesDeTodo() {
    System.out.println("Estoy en el BeforeAll");
    calculadora = new Calculadora();
}

@AfterAll
public static void despuesDeTodo() {
    System.out.println("Estoy en el AfterAll");
    calculadora = new Calculadora();
}

@BeforeEach
public void antes() {
    System.out.println("Estoy en el BeforeEach");
    calculadora.clear();
}

@AfterEach
public void despues() {
    System.out.println("Estoy en el AfterEach");
}
```

El resultado sería este



```
<terminated> CalculadoraTest [JUnit] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (22 abr. 2022 20:27:42)
Estoy en el BeforeAll
Estoy en el BeforeEach
Estoy en testSumaDosNumeros()
Estoy en el AfterEach
Estoy en el BeforeEach
Estoy en testExcepcionDividir()
Estoy en el AfterEach
Estoy en el BeforeEach
Estoy en testSumaDosNumerosAns()
Estoy en el AfterEach
Estoy en el AfterAll
```

Y las pruebas se ejecutarían sin problemas.

**@AfterAll** es muy útil para cerrar conexiones, eliminar objetos de memoria o similar.

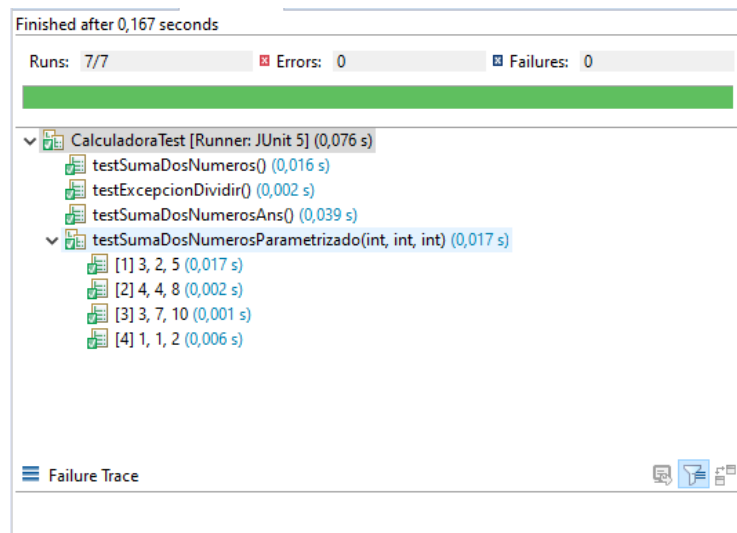
## 2.7. Pruebas parametrizadas

Las pruebas parametrizadas son una forma cómoda de ejecutar pruebas unitarias en las que queremos comprobar que para varios valores obtenemos varios resultados.

Se utiliza la anotación **@ParameterizedTest** en vez de **@Test** y **@CsvSource** para indicar la fuente de datos sobre los que se va a realizar las pruebas.

```
@ParameterizedTest
@CsvSource({ "3,2,5", "4,4,8", "3,7,10", "1,1,2" })
void testSumaDosNumerosParametrizado(int a, int b, int esperado) {
    System.out.println("Estoy en testSumaDosNumerosParametrizado()");
    int resultado = calculadora.suma(a, b);
    assertEquals(esperado, resultado);
}
```

De esta manera se podrá realizar la prueba sobre todos estos datos.



## 2.8. Cobertura del código

También se puede comprobar cuanto porcentaje de código cubren nuestras pruebas.

CalculadoraTest (26-abr-2022 11:39:26)		
Element	Coverage	Covered Instructio
▼ edd.tema4.ejemplo.calculadora	74,0 %	1
▼ src	74,0 %	1
▼ main.edd.tema4.ejemplo.calculadora	46,7 %	
▼ Calculadora.java	46,7 %	
▼ Calculadora	46,7 %	
● suma(int)	0,0 %	
● dividir(int, int)	46,7 %	
● resta(int, int)	0,0 %	
● algoritmo()	0,0 %	
● Calculadora()	100,0 %	
● ans()	100,0 %	
● clear()	100,0 %	
● suma(int, int)	100,0 %	
▼ test.edd.tema4.ejemplo.calculadora	93,0 %	
▼ CalculadoraTest.java	93,0 %	
▼ CalculadoraTest	93,0 %	
● antesDeTodo()	100,0 %	
● despuesDeTodo()	100,0 %	
● antes()	100,0 %	
● despues()	100,0 %	
▲ testExcepcionDividir()	100,0 %	
▲ testSumaDosNumeros()	100,0 %	
▲ testSumaDosNumerosAns()	100,0 %	
▲ testSumaDosNumerosParametriza	100,0 %	

**Ejercicio 1:** Intenta conseguir el máximo porcentaje de cobertura de código en este ejemplo (fijándote tanto en la herramienta coverage como en los conocimientos de programación que tienes).

**Ejercicio 2:** Crea un proyecto que tenga, al menos, 8 métodos de los ejercicios de la UT4 (String y StringBuilder) de programación. Realiza pruebas unitarias sobre ellos e intenta llegar a la máxima cobertura posible (fijándote tanto en la herramienta coverage como en los conocimientos de programación que tienes).

**Ejercicio 3:** Crea un proyecto que tenga, al menos, 4 de los métodos de los ejercicios de la UT4 (Recursividad). Realiza pruebas unitarias sobre ellos e intenta llegar a la

máxima cobertura posible (fijándote tanto en la herramienta coverage como en los conocimientos de programación que tienes).

**Ejercicio 4:** Crea un proyecto que tenga, al menos, 5 de los métodos de los ejercicios hechos en programación (me valen de cualquier tema). Intenta que sean variados y que permitan mostrar todos los conocimientos que has adquirido a lo largo de este tema (fijándote tanto en la herramienta coverage como en los conocimientos de programación que tienes).