Pruebas Unitarias con JUNIT

1. INTRODUCCIÓN

- Hasta ahora hemos estado haciendo pruebas de forma manual a partir de una especificación o de un código.

- Vamos a utilizar una herramienta para implementar pruebas que verifiquen que nuestro programa genera los resultados que de él esperamos.

- Junit es una herramienta para realizar pruebas unitarias automatizadas. Las pruebas se realizan sobre una clase para probar su comportamiento de modo aislado del resto de clases de la aplicación (aunque en ocasiones como sabemos el funcionamiento de una clase depende de otras clases para poder llevar a cabo su función).

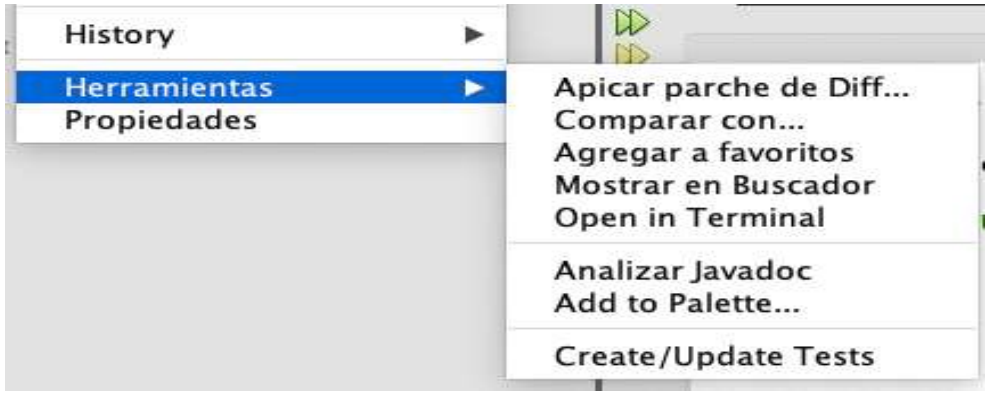
2. CREACIÓN DE UNA CLASE DE PRUEBA

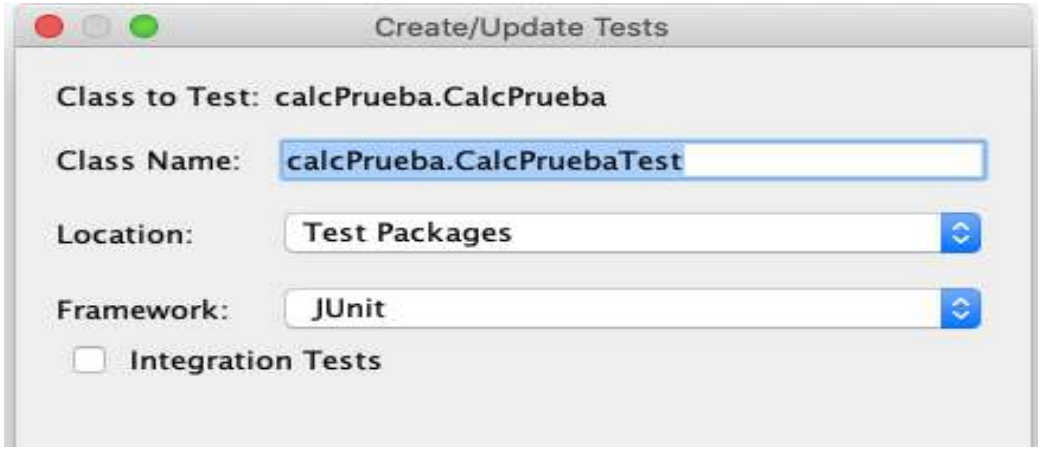
- Para empezar a usar Junit crearemos un nuevo proyecto. **OJO: Vamos a crear este proyecto de tipo Java With Maven**

- Creamos la clase a probar que vamos a llamar CalcPrueba o cualquier otro nombre. Esta nueva clase tendrá:
  - Dos variables de clase: num1 y num2 de tipo int
  - Cuatro métodos (suma, resta, multiplica y divide)que devolverán un entero
  - Método constructor (CalcPrueba) que será el único que recibirá parámetros (dos enteros) pero que no devolverá nada.

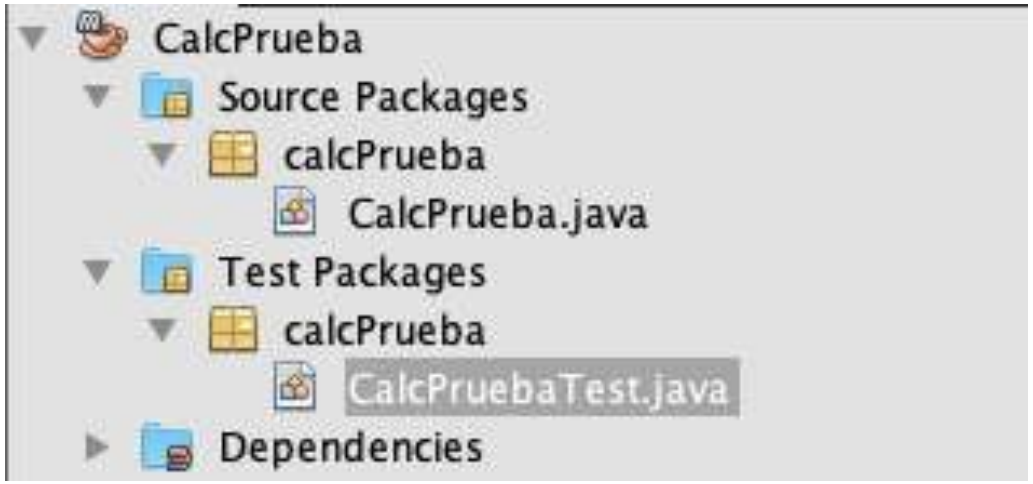
```java
package calcPrueba;

/**
 *
 * @author guillermopalazoncano
 */
public class CalcPrueba {

    private int num1;
    private int num2;

    public CalcPrueba(int a, int b){
        num1 = a;
        num2 = b;
    }

    public int suma(){
        return num1 + num2;
    }

    public int resta(){
        return num1 - num2;
    }

    public int multiplica(){
        return num1 * num2;
    }

    public int divide(){
        return num1 / num2;
    }
```

- A continuación, hay que crear la clase de prueba. Con la clase CalcPrueba seleccionada pulsamos el botón derecho y seleccionamos Herramientas y la opción Create/Update Test.
- A continuación dejamos seleccionado JUnit en el Framework y le damos a aceptar.
- OJO: Es posible que si se cambia el nombre y no finaliza en Test NO funcionen las pruebas





- Se nos ha abierto una nueva clase y podemos comprobar que tenemos código en Paquetes de Prueba que también se ha generado.




```java
package calcPrueba;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

/**
 *
 * @author guillermopalazoncano
 */
public class CalcPruebaTest {

    public CalcPruebaTest() {
    }

    @BeforeAll
    public static void setUpClass() {
    }
```

- Podemos destacar las siguientes características sobre esta clase de prueba que se ha generado automáticamente.
  - Se crean 4 métodos de prueba (@Test), uno para cada método seleccionado anteriormente:
  - Los métodos son públicos, no devuelven nada y no reciben ningún argumento.
  - El nombre de cada método incluye la palabra test al principio: testSuma, testResta, testMultiplica y testDivide.
  - Sobre cada uno de los métodos aparece la anotación @Test que indica al compilador que es un método de prueba.
  - Termina con un mensaje de tipo fail

```
@Test
public void testSuma() {
```

```
 */
@Test
public void testSuma() {
    System.out.println("suma");
    CalcPrueba instance = null;
    int expResult = 0;
    int result = instance.suma();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
```

3. ANOTACIONES

- BeforeAll
  - Sólo puede haber un método con este marcador.
  - Este método será invocado una vez al principio del lanzamiento de todas las pruebas. Se suele utilizar para inicializar los atributos comunes a todas las pruebas o para realizar acciones que tardan un tiempo considerarse en ejecutarse.
- AfterAll
  - Sólo puede haber un método con este marcador. Este método será invocado un sola vez cuando finalicen todas las pruebas.
- BeforeEach:
  - Los métodos marcados con está anotación, serán invocados antes de iniciarse cada prueba. Puede haber varios métodos en la clase de prueba con esta anotación.

- AfterEach
  - Los métodos marcados con está anotación, serán invocados después de ejecutarse cada prueba.
- Test
  - Representa un test que se debe ejecutar. Puede tener dos tipos de modificadores
- DisplayName
  - Nos permite hacer una mejor descripción al test. Suele acompañar a @Test
  - @DisplayName("Vamos a realizar la prueba de la suma de 2 + 2 es igual 4")

- Disabled
  - Los métodos marcados con esta anotación no serán ejecutados. Se suelen poner para desactivar las pruebas que no pueden ser realizadas por algún motivo.
  - Puede tener un parámetro que indica un texto que se visualizará en la plataforma donde se ejecuten los test.
  - Acompaña siempre a @Test
- NOTA: Algunas de estas anotaciones dan problemas con NetBeans (@disabled). Para solucionarlo, si nuestro proyecto es maven debemos añadir en nuestro fichero pom.xml, la parte de <dependencies> y <build> del fichero pom.xml que se encuentra en el AulaVirtual

4. ASERCIONES (ASSERT)

- Antes de preparar el código para los métodos de prueba vamos a ver una serie de métodos de Junit para hacer las comprobaciones. Todos estos se podrían ver en: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

- JUnit proporciona métodos para poner a prueba ciertas condiciones, estos métodos suelen empezar asserts permitiendo especificar el mensaje de error, el valor esperado y el real. En la siguiente tabla mostramos algunos de los más utilizados (si hemos visitado la ruta indicado, podemos ver, que existen muchísimos más).

Método	Misión (qué comprueba)
fail (String)	Hace que la prueba falle. Si se incluye un String la prueba lanzando el mensaje. Ahora mismo todos nuestros métodos (testSuma, testResta,…) fallarían.
assertTrue(boolean expresion, [String mensaje])	Comprueba si expresión es true. Si no es true la prueba fallará y se mostrará mensaje (en caso de incluirse ya que no es obligatorio)
assertFalse(boolean expresion, [String mensaje])	Comprueba si expresión es false. Si no es false la prueba fallará y se mostrará mensaje (en caso de incluirse ya que no es obligatorio)
assertsEquals(esperado, real, [String mensaje)	Comprueba que el valor Esperado sea igual al valor real. Si no son iguales la prueba fallará y se mostrará mensaje (en caso de incluirse ya que no es obligatorio). Esperado y real pueden ser de diferentes tipos

Método	Misión (qué comprueba)
assertNull(objeto, [String mensaje])	Comprueba que el objeto es null. En caso de no ser nulo la prueba falla y se mostrará el mensaje si se ha metido
assertNotNull(objeto, [String mensaje])	Comprueba que el objeto sea not null. En caso de ser nulo la prueba fallará y se mostrará el mensaje si se ha metido
assertSame(obj_esperado, obj_real, [String mensaje])	Comprueba que obj_esperado y obj_real sean el mismo objeto. Si no son el mismo objeto la prueba fallará y se incluye el mensaje al producirse el error se lanzará el mensaje
assertNotSame(obj_esperado, obj_real, [String mensaje])	Comprueba que obj_esperado y obj_real no sean el mismo objeto. Si son el mismo la prueba fallará y se incluye el mensaje al producirse el error se lanzará el mensaje

5. PREPARACIÓN Y EJECUCIÓN DE LAS PRUEBAS

- Si lanzo la ejecución de la clase de prueba (mayúsculas + F6) vemos que los 4 test fallan (he borrado todos los métodos a excepción de las pruebas).

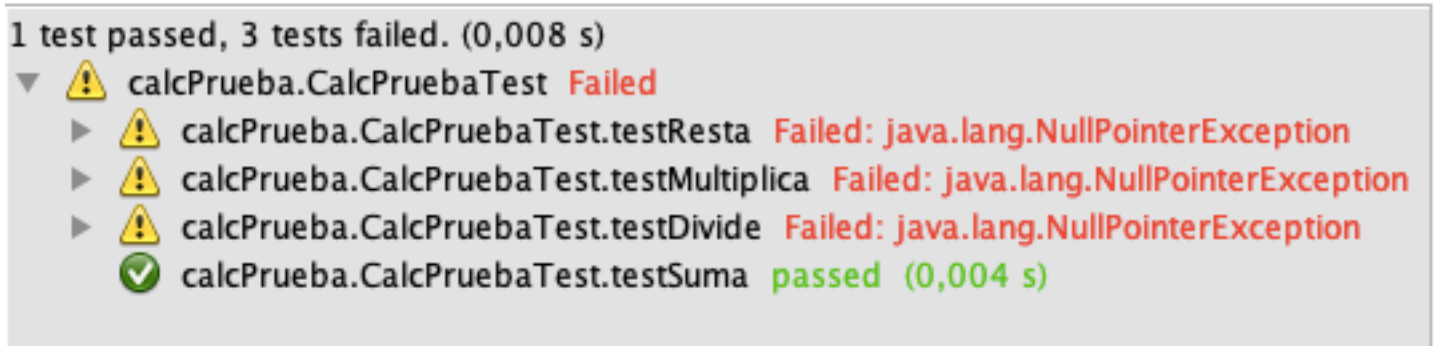
- Vamos a crear el código de prueba para hacer el método testSuma() que probará el método suma() de la clase CalcPrueba.

- Lo primero que hacemos es crear una instancia de la clase CalcPrueba, llamamos al método suma() llevando los valores a sumar, por ejemplo, 20 y 10, y comprobamos los resultados con el método assertEquals().
- En el último parámetro de este último método escribimos el mensaje de error si no funciona bien "El test de suma ha dado mal". Antes hemos indicado como parámetros el resultado esperado al realizar el método suma(), en este caso es 30 y como último parámetro asignamos el resultado obtenido al llam

```
@Test
@DisplayName("Vamos a realizar la prueba de la suma de 20 y 10")
public void testSuma() {
    CalcPrueba calcu = new CalcPrueba(20,10);
    int resultado = calcu.su
    assertEquals(30, resultado, "El test de la suma ha dado mal");
}
```

int resultado

- Si volvemos a ejecutar el test, se siguen mostrando algunos fallos ya que no se han implementado todos los test de pruebas, pero podemos ver que ya se ha solucionado el prueba de testSuma

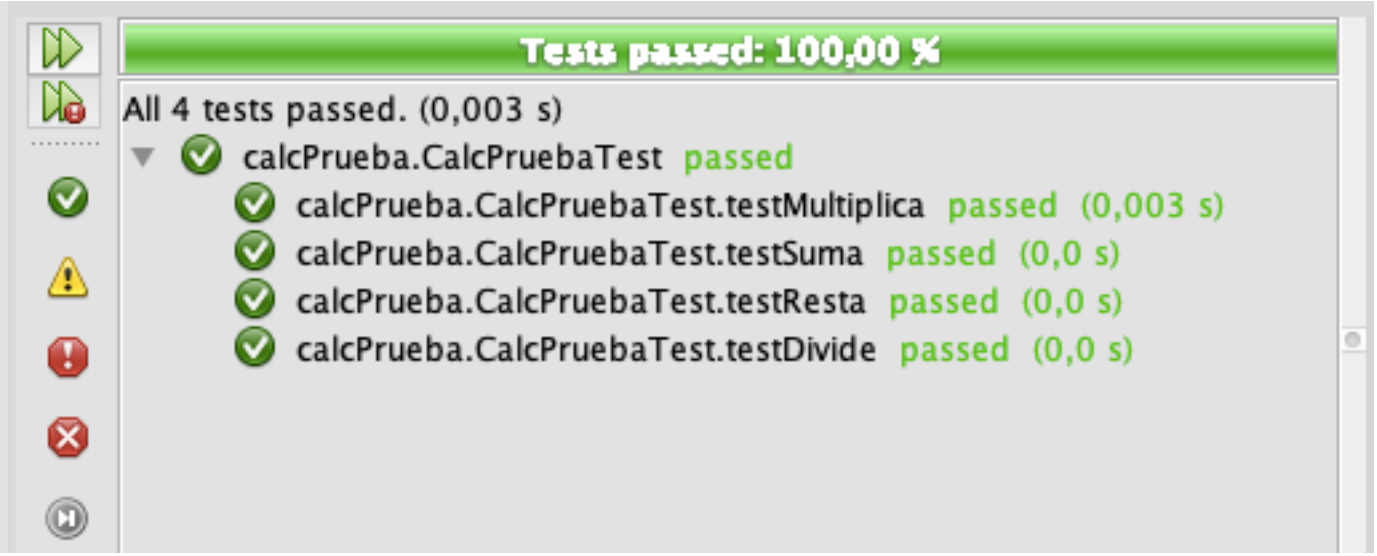


- Al lado de cada prueba aparece un icono con una marca:
  - Una marca de verificación verde indica prueba exitosa.
  - Una marca de exclamación en amarillo indica un fallo.
  - Una marca de exclamación en rojo indica un error.
- En nuestro caso un test ha resultado exitoso y tres han causado fallo.
- Antes de Junit5, había una distinción clara entre fallo y error
  - Un fallo es una comprobación que no se cumple (si esperamos que la suma sea un número pero da otro diferente) mientras que un error es una excepción durante la ejecución del código (por ejemplo si se dividiera entre 0 o que de un java.lang.NullPointerException)
- Junit5, clasifica tanto errores en programación como fallos de pruebas como Fallos

- Rellenamos el resto de los métodos de prueba escribiendo en los métodos assertEquals()el valor esperado y el resultado de realizar la operación con los números 20 y 10.

```java
/**
 * Test of resta method, of class CalcPrueba.
 */
@Test
public void testResta() {
    CalcPrueba calcu = new CalcPrueba(20,10);
    int resultado = calcu.resta();
    assertEquals(10, resultado, "El test de la resta ha dado mal");
}

/**
 * Test of multiplica method, of class CalcPrueba.
 */
@Test
public void testMultiplica() {
    CalcPrueba calcu = new CalcPrueba(20,10);
    int resultado = calcu.multiplica();
    assertEquals(200, resultado, "El test de la multiplicación ha dado mal");
}

/**
 * Test of divide method, of class CalcPrueba.
 */
@Test
public void testDivide() {
    CalcPrueba calcu = new CalcPrueba(20,10);
    int resultado = calcu.divide();
    assertEquals(2, resultado, "El test de la división ha dado mal");
}
```

- Ahora al ejecutar nuestra clase de prueba podemos comprobar que no tenemos ningún fallo ni ningún error y todas nuestras pruebas han resultado exitosas.



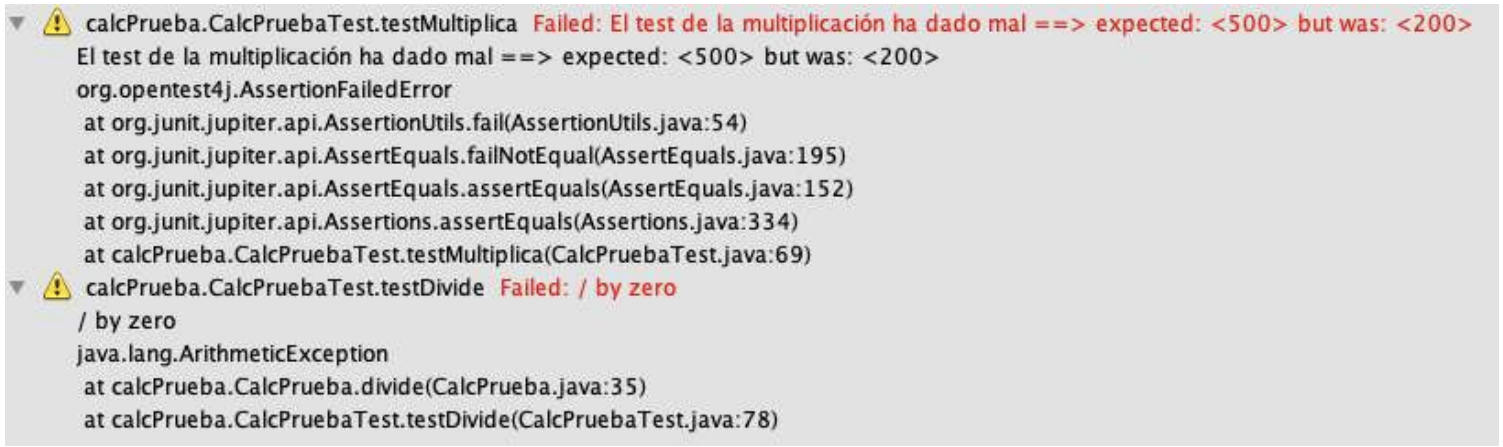
- Vamos a cambiar ahora los códigos del caso de multiplicación y división
  - Para hacer que el método multiplica() produzca un fallo hacemos que el valor esperado no coincida con el resultado.
  - Para hacer que el método divide() produzca un fallo, al crear el objeto calculadora asignamos un valor 0 al segundo parámetros (será el denominador de la división, al dividir por 0 se produce una excepción), Junit 4 nos hubiera catalogado este fallo como Error.

```
@Test
public void testMultiplica() {
    CalcPrueba calcu = new CalcPrueba(20,10);
    int resultado = calcu.multiplica();
    assertEquals(500, resultado, "El test de la multiplicación ha dado mal");
}
```

```
@Test
public void testDivide() {
    CalcPrueba calcu = new CalcPrueba(20,0);
    int resultado = calcu.divide();
    assertEquals(2, resultado, "El test de la división ha dado mal");
}
```

- Lanzamos la ejecución

- Se pueden desplegar estos test que han dado problemas y obtener más información sobre ellos



- Vemos que en testMultiplica se esperaba obtener un 500, pero hemos obtenido 200 y sin embargo en testDivide el problema está en que nuestro código para esos valores da un problema en la línea 35 (Clase CalcPrueba) debido a la llamada de la línea 78 de la clase del test.

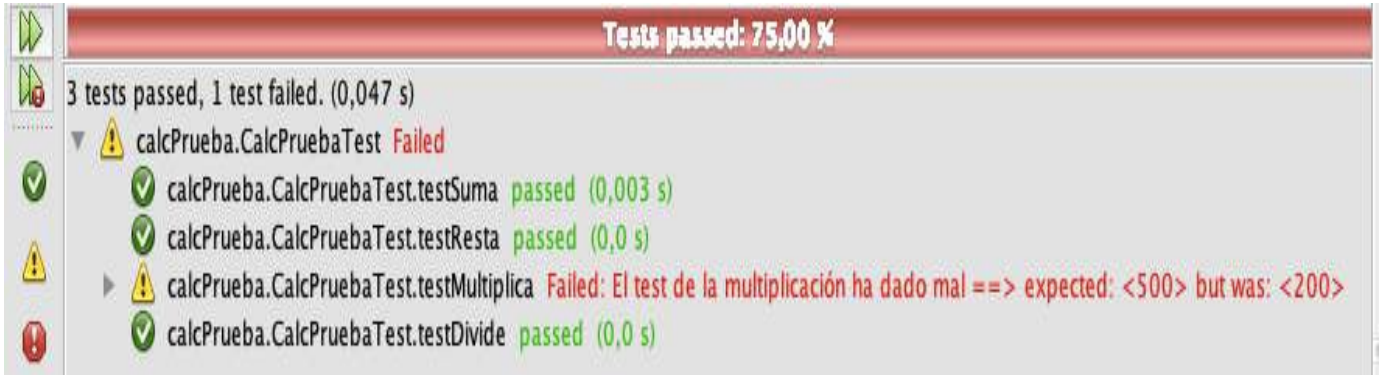
- Dejamos nuestras clases de pruebas tal y como estaban, para que no den fallo y errores. Como hemos detectado un error en Calculadora vamos a solucionar el mismo.
- Tras cambiar el tipo que devuelve el método de int a Integer (lo tenemos que hacer para poder devolver null), tenemos el siguiente código:

```
public Integer divide(){
    if (num2 == 0){
        return null;
    } else {
        return num1 / num2;
    }
}
```

- Cambiamos ahora la invocación en el test para probar assertNull

```
@Test
public void testDivide() {
    CalcPrueba calcu = new CalcPrueba(20,0);
    Integer resultado = calcu.divide();
    assertNull(resultado, "El test de la división ha dado mal");
}
```

- Esta prueba es ahora exitosa



- En la vista de Junit se muestran varios botones:
  - Next failure  . Navega a la siguiente prueba que ha producido fallo o error
  - Previous failure  . Navega a la anterior prueba que ha producido fallo o error
  - Show error  . Muestra las pruebas que tienen error.
  - Show failed  . Muestra las pruebas que han dado fallo.
  - Show skipped  . Muestra las pruebas omitidas.
  - Show aborted  . Muestra las pruebas abortadas
  - Rerun  . Lanza otra vez la ejecución de las pruebas.
  - Rerun failed  . Lanza otra vez la ejecución pero solo de las pruebas con fallo o error

- En todos los métodos de las pruebas anteriores se repite la línea "CalcPrueba calcu = new CalcPrueba(20,10);". Esta sentencia de inicialización se puede escribir una sola vez dentro de la clase (tras definir calcu como variable de clase). Creamos un nuevo método creaCalculadora y le indicamos la anotación @BeforeAll. Añadimos esa línea en el método y la borramos de los métodos Test
- También podemos aprovechar la anotación @After (también se podría indicar en un @AfterAll) para hacer una limpieza de datos. Creamos un nuevo método borraCalculadora que lo único que hará será poner calcu como null.

```
public class CalcPruebaTest {

    private static CalcPrueba calcu;
```

```
@BeforeAll
public static void creaCalculadora(){
    calcu = new CalcPrueba(20,10);
}
```

```
@AfterAll
public static void borraCalculadora(){
    calcu = null;
}
```

- Si lanzamos ahora la ejecución tenemos el siguiente resultado

- testDivide da un fallo en la prueba porque tiene aún puesto el assertNull. Sin embargo con los valores de 20,10, el resultado es 2 y no nulo.

```
@Test
public void testDivide() {
    Integer resultado = calcu.divide();
    assertNull(resultado, "El test de la división ha dado mal");
}
```

6. PRUEBAS PARAMETRIZADAS

- Supongamos que queremos ejecutar una prueba varias veces con distintos valores de entrada, por ejemplo, queremos probar el método divide() con diferentes valores.
- Junit nos permite generar parámetros para lanzar varias veces una prueba con dichos parámetros. Para poder hacer esto seguiremos estos pasos:
  - 1. Debemos añadir la etiqueta @ParameterizedTest al método de prueba que vamos a usar para la batería de pruebas batería de pruebas.

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

- 2. Este irá acompañada de otra anotación en la que se indicará la forma en la que le llegaran los parámetros al método que va a realizar todas estas pruebas.
- Existen diversas formas: https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources
- En nuestro caso vamos a utilizar @CsvSource que nos permite expresar listas de argumentos como valores separados por comas.

```
@ParameterizedTest
@CsvSource({
        "1, 1, 2",
        "2, 3, 5",
        "4, 3, 9"
})
void sumaParametrizada(int num1, int num2, int num3){
    CalcPrueba calcu2 = new CalcPrueba(num1, num2);
    int resultado = calcu2.suma();
    assertEquals(num3, resultado, "El test de la suma ha dado mal");
}
```

- 3. En el ejemplo podemos ver que el nuevo método sumaParametrizada, ahora tiene tres parámetros de tipo entero. Estos los leeremos del @csvSource y además, al tener tres filas estaremos indicando tres pruebas.
  - En la primera prueba num1 toma el valor 1, num2 toma el valor 1 y num3 el valor 2
  - En la segunda prueba num1 toma el valor 2, num2 toma el valor 3 y num3 el valor 5
  - En la tercera prueba num1 toma el valor 4, num2 toma el valor 3 y num3 el valor 7

```
@ParameterizedTest
@CsvSource({
        "1, 1, 2",
        "2, 3, 5",
        "4, 3, 9"
})
void sumaParametrizada(int num1, int num2, int num3)
```

- Observando el código del método sumaParametrizada

```
void sumaParametrizada(int num1, int num2, int num3){
    CalcPrueba calcu2 = new CalcPrueba(num1, num2);
    int resultado = calcu2.suma();
    assertEquals(num3, resultado, "El test de la suma ha dado mal");
}
```

  - Coge los dos primeros parámetros y crea una nueva instancia (objeto) de la lase CalcPrueba con estos valores.
  - Lanza el método suma y lo almacena en resultado
  - Comprueba si ese resultado es igual al tercer parámetro
- Por tanto, se habían diseñado tres prueba: Las dos primeras serán éxito y la última fallará.

- Teniendo en cuenta que tenemos omitidas las pruebas anteriores (testSuma, testResta, testDivide y testMultiplica), si lanzamos estas pruebas este sería el resultado que obtenemos