UNIDAD 4. JUNIT

VICENT MARTÍ

OBJETIVOS

- Conocer las principales características de JUnit
- Realizar los ejercicios guiados para conocer como funciona

INTRODUCCIÓN

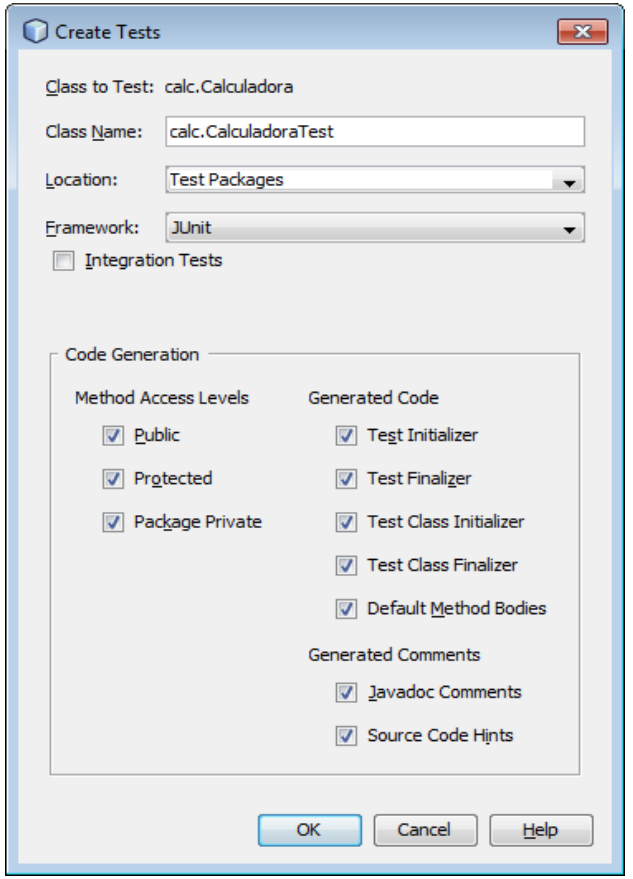
- JUnit es una herramienta para realizar pruebas unitarias automatizadas
- JUnit está integrado en NetBeans y Eclipse
- Se presenta JUnit
  - Presta atención a la versión que tienes instalada en NetBeans o Eclipse

CLASE PRUEBA

- Las pruebas unitarias se realizan sobre una Clase (por ejemplo, Calculadora) para probar su comportamiento de forma aislada independientemente del resto de clases de la aplicación
  - Se creará una Clase de Prueba (por ejemplo, Calculadora**Test**)
  - Esta clase se va a insertar en un nuevo paquete de nuestro proyecto denominado **Test Packages**

MÉTODOS DE PRUEBA

- En la Clase de Prueba se van a generar los métodos de prueba con un código prototipo:
  - El método de prueba llamará al método que va a ser evaluado
  - Mediante los métodos "assert" de JUnit se verificará si están realizando la actividad deseada



MÉTODOS JUNIT

- Aserciones
  - Los método assertXXX() se utilizan para hacer las pruebas
  - Estos métodos permiten comprobar si la salida del método que se está probando concuerda con los valores esperados
  - Todos devuelven tipo void

Métodos	Misión
assertNull (Object objeto) **assertNull (string mensaje, Object objeto)**	Comprueba que el objeto sea Null. Si no es null y se incluye el String al producirse el error devolverá el mensaje.
assertNotNull (Object objeto) **assertNotNull (string mensaje, Object objeto)**	Comprueba que el objeto no sea Null. Si es null y se incluye el String al producirse el error devolverá el mensaje.

MÉTODOS JUNIT

Métodos	Misión
assertTrue (boolean expresión) **assertTrue (String mensaje, boolean expresión)**	Comprueba que la expresión se evalúe a true. Si no es true y se incluye String, devolverá mensaje de producirse error
assertFalse (boolean expresión) **assertFalse (String mensaje, boolean expresión)**	Comprueba que la expresión se evalúe a false. Si no es false y se incluye String, devolverá mensaje de producirse error
assertEquals (valorEsperado, valorReal) **assertEquals (String mensaje, valorEsperado, valorReal)**	Comprueba que el valorEsperado sea igual al valorReal. Si no son iguales y se incluye el String, entonces se lanzará el mensaje. ValorEsperado y valorReal pueden ser de diferentes tipos.

MÉTODOS JUNIT

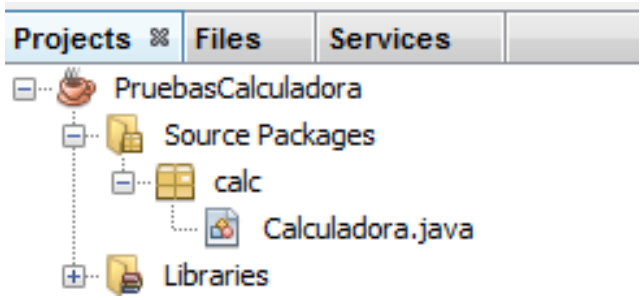
Métodos	Misión
assertSame (Object objetoEsperado, Object objetoReal) **assertSame (String mensaje, Object objetoEsperado, Object objetoReal)**	Comprueba que el objetoEsperado y objetoReal sean el mismo objeto. Si no son el mismo y se incluye el String, al producirse error se lanzará el mensaje.
assertNotSame (Object objetoEsperado, Object objetoReal) **assertNotSame (String mensaje, Object objetoEsperado, Object objetoReal)**	Comprueba que el objetoEsperado y objetoReal no sean el mismo objeto. Si son el mismo y se incluye el String, al producirse error se lanzará el mensaje.
fail() **fail (String mensaje)**	Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el mensaje.

OTROS MÉTODOS JUNIT

- Inicializadores y Finalizadores:
  - En algunos casos no es necesario utilizar estos métodos pero siempre se suelen incluir
- El **método setUp** es un método de inicialización de la prueba y se ejecuta antes de cada caso de prueba en la clase de prueba
  - No es necesario para ejecutar pruebas
  - Puede ser necesario para inicializar algunas variables antes de iniciar la prueba
- El **método tearDown** es un método finalizador de prueba y se ejecutará después de cada test en la clase prueba
  - No es necesario para ejecutar las pruebas
  - Puede ser necesario para limpiar algún dato que fue requerido en la ejecución de los casos de prueba

EJERCICIO GUIADO

- En NetBeans creamos un nuevo proyecto y creamos la clase Calculadora



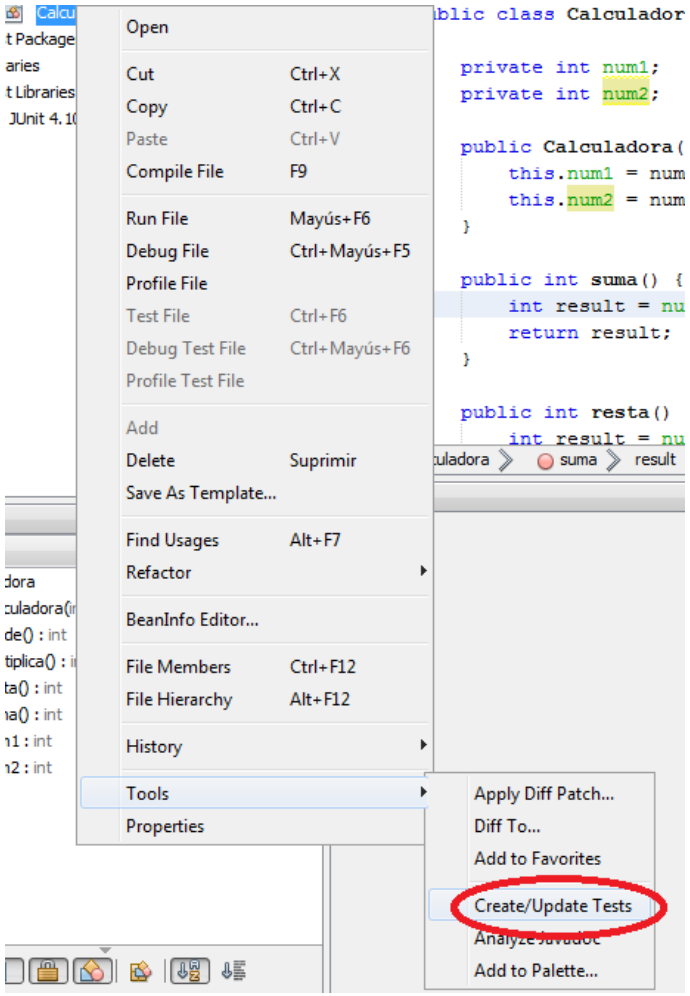
Calculadora.java

Source | History

```
1  package calc;
2  public class Calculadora {
3      private int num1;
4      private int num2;
5
6      public Calculadora(int num1, int num2) {
7          this.num1 = num1;
8          this.num2 = num2;
9      }
10     public int suma() {
11         int result = num1 + num2;
12         return result;
13     }
14     public int resta() {
15         int result = num1-num2;
16         return result;
17     }
18     public int multiplica() {
19         int result = num1 * num2;
20         return result;
21     }
22     public int divide() {
23         int result = num1 / num2;
24         return result;
25     }
```

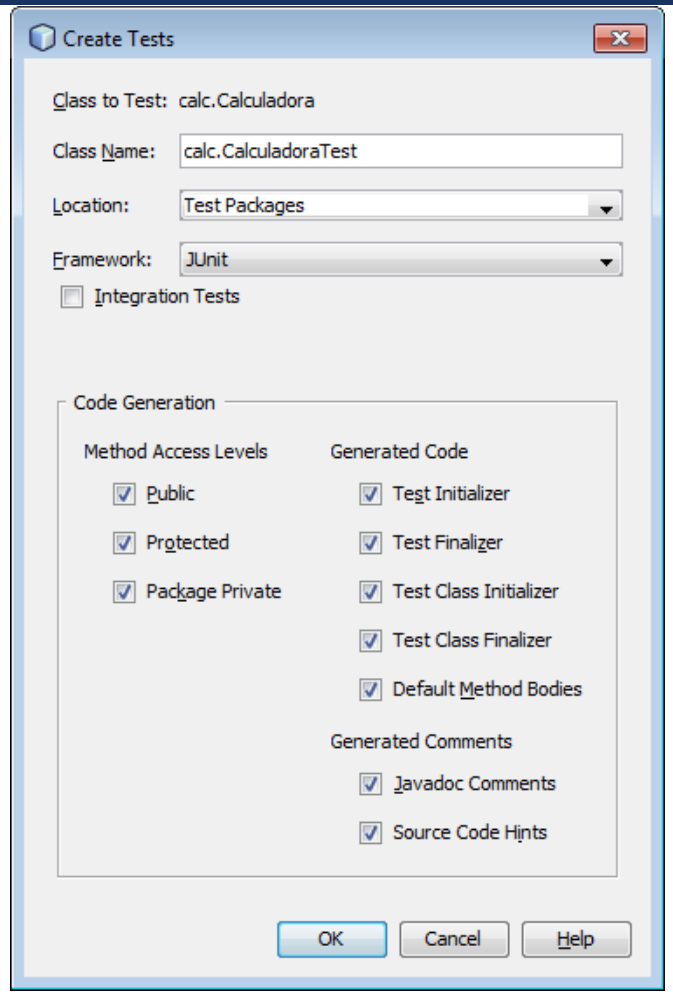
EJERCICIO GUIADO

- Creamos la Clase de Tests:
  - Seleccionamos el dase Calculadora > BotónDerecho > Tools > Create/UpDate Tests
  - (También desde el menu Tools-> Create/UpDate Tests)



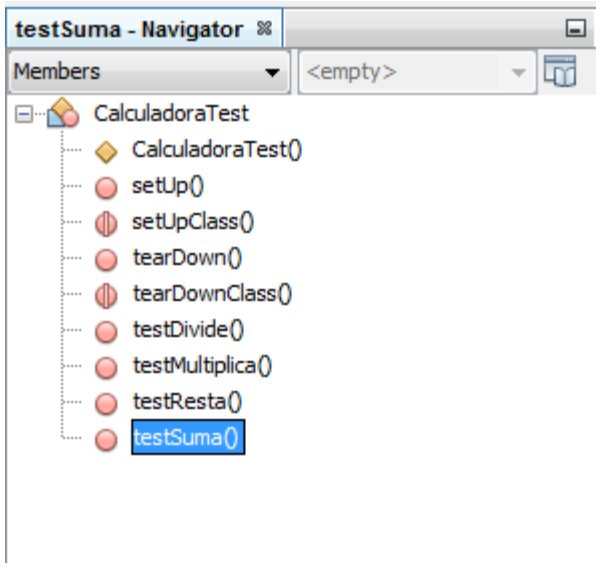
EJERCICIO GUIADO

- Puesto que vamos a probar la clase Calculadora, por convenio es recomendable llamar a la clase de prueba CalculadoraTest
  - Esta clase se va a insertar en un nuevo paquete de nuestro proyecto denominado **Test Packages** (*Paquete de pruebas*)
- Nos abre la ventana Create Tests
  - Rellenamos como en la imagen y pulsamos OK



EJERCICIO GUIADO

- Nos ha generado la clase CalculadoraTest:



Calculadora.java | CalculadoraTest.java

Source | History

```
18      */
19     public class CalculadoraTest {
20
21         public CalculadoraTest() {...2 lines }
23
24         @BeforeClass
25         public static void setUpClass() {...2 lines }
27
28         @AfterClass
29         public static void tearDownClass() {...2 lines }
31
32         @Before
33         public void setUp() {...2 lines }
35
36         @After
37         public void tearDown() {...2 lines }
39
40         /** Test of suma method, of class Calculadora ...3 lines */
43         @Test
44         public void testSuma() {...9 lines }
53
54         /** Test of resta method, of class Calculadora ...3 lines */
57         @Test
58         public void testResta() {...9 lines }
67
68         /** Test of multiplica method, of class Calculadora ...3 lines */
71         @Test
72         public void testMultiplica() {...9 lines }
81
82         /** Test of divide method, of class Calculadora ...3 lines */
85         @Test
86         public void testDivide() {...9 lines }
95
96     }
97
```

EJERCICIO GUIADO

Vemos que:

- Se crea un método de prueba por cada método
  - Estos 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(), testDivide()
- Sobre cada método aparece la notación @Test que indica al compilador que es un método de prueba
- Cada método tiene una llamada a fail() con mensaje fallo
  - Este método hace que el test termine con fallo y lanzando el mensaje

EJERCICIO GUIADO

Vemos que:

- El método testSuma() es un método para probar el método suma de la clase Calculadora

Lo modificamos:

- Creamos una instancia de la clase Calculadora con los valores a operar (20 y 10)
- Llamamos al método Suma
- Comprobamos los resultados utilizando método JUnit assertEquals()

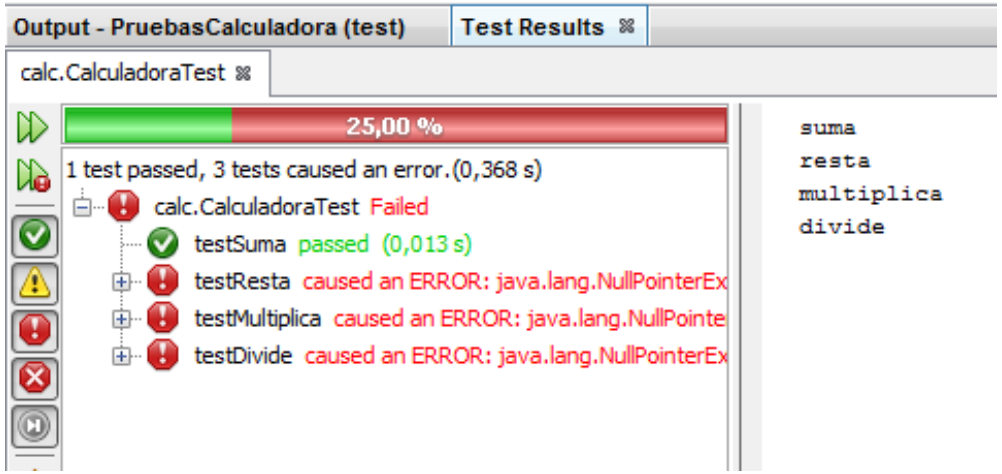
EJERCICIO GUIADO

- Modifica testSuma()

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

     System.out.println("suma");
    Calculadora instance = new Calculadora(20,10);
    int expResult = 30;
    int result = instance.suma();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to f
    //fail("The test case is a prototype.");

}
```

EJERCICIO GUIADO

- Ejecutamos el test:
  - Seleccionamos la clase Calculadora > BotónDerecho> Test File
- Se nos abre ventana Junit donde aparecen resultados
- Solo se ha realizado satisfactoriamente la prueba con el metodo testSuma()



EJERCICIO GUIADO

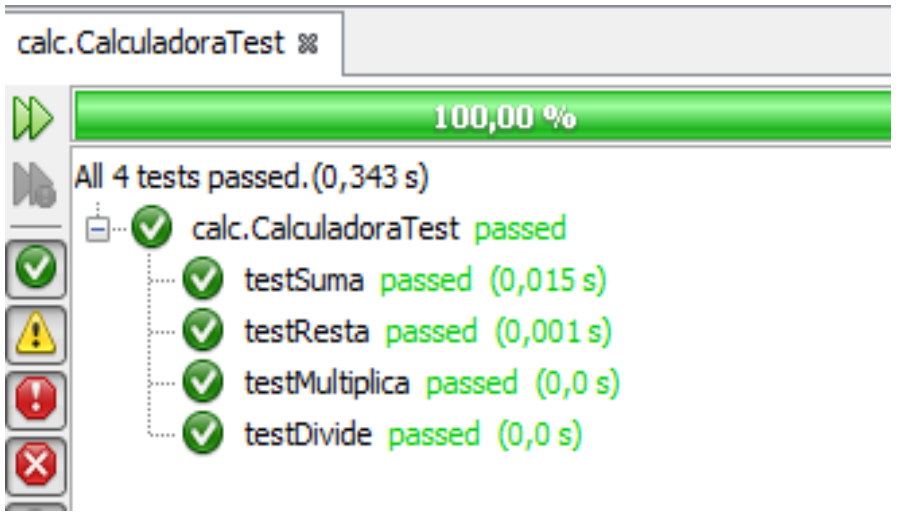
- Al lado de cada prueba aparece icono con una marca:
  - Verde indica exitosa
  - Amarillo indica fallo
  - Rojo indica error

- Ojo no es lo mismo fallo que error!!
  - fallo es una comprobación que no se cumple
  - error es una excepción durante la ejecución de código

EJERCICIO GUIADO

- Implementa los métodos para probar los métodos resta, multiplica y divide de la clase Calculadora

```
public void testResta() {
    System.out.println("resta");
    Calculadora instance = new Calculadora(20, 10);
    int expResult = 10;
    int result = instance.resta();
    assertEquals(expResult, result);
}
/** Test of multiplica method, of class Calculadora ...3 li
@Test
public void testMultiplica() {
    System.out.println("multiplica");
    Calculadora instance = new Calculadora(20, 10);
    int expResult = 200;
    int result = instance.multiplica();
    assertEquals(expResult, result);
}
/** Test of divide method, of class Calculadora ...3 lines
@Test
public void testDivide() {
    System.out.println("divide");
    Calculadora instance = new Calculadora(20, 10);
    int expResult = 2;
    int result = instance.divide();
    assertEquals(expResult, result);
}
```



DIFERENCIA FALLO Y ERROR

- Provocamos fallo en el método multiplica()
  - hacemos que el valor esperado no coincida con el esultado
  - incluimos String con mensaje para si se produce fallo muestre el mensaje

```
/**
 * Test of multiplica method, of class Calculadora.
 */
@Test
public void testMultiplica() {
    System.out.println("multiplica");
    Calculadora instance = new Calculadora(320, 10);
    int expResult = 200;
    int result = instance.multiplica();
    assertEquals("Fallo en el Producto", expResult, result);
}
```

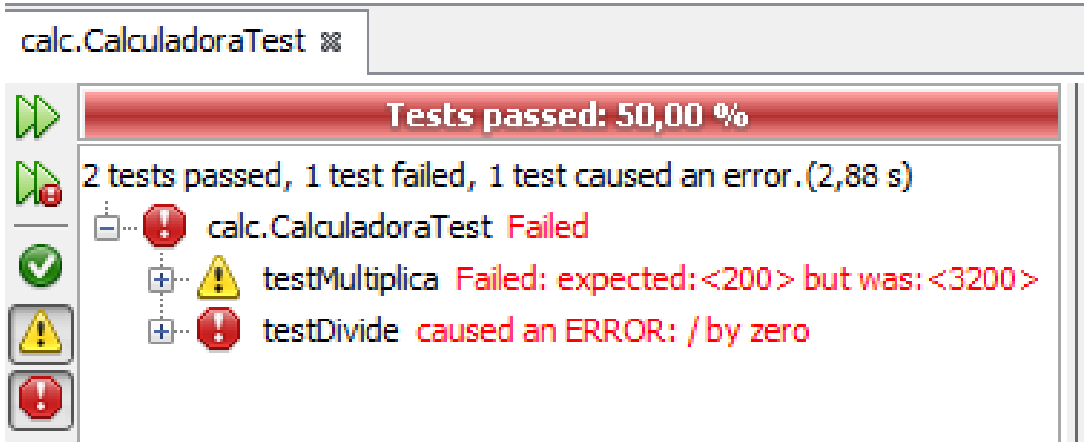
DIFERENCIA FALLO Y ERROR

- Provocamos error en el método divide()
  - al crear objeto calculadora asignamos 0 al segundo parámetro
  - Al dividir por 0 producirá excepción

```
/**
 * Versión 1. Test of divide method, of class Calculadora.
 */
@Test
public void testDivide() {
    System.out.println("divide");
    Calculadora instance = new Calculadora(20, 0);
    int expResult = 2;
    int result = instance.divide();
    assertEquals(expResult, result);
}
```

DIFERENCIA FALLO Y ERROR

- Ojo no es lo mismo fallo que error!!
  - fallo es una comprobación que no se cumple
  - error es una excepción durante la ejecución de código



CAPTURANDO EXCEPCIÓN

- Nos interesa capturar la excepción siempre que se produzca una división entre cero
  - Vemos la implementación:

```
@Test
 public void testDivide() {
    System.out.println("divide");
    try {
        Calculadora instance = new Calculadora(20, 0);
        int resultado = instance.divide();
        //assertEquals(expResult, result);
        fail("Fallo, Deberia lanzar la excepcion");
    } catch (ArithmeticException e) {
        //prueba satisfactoria
    }
}
```

EJERCICIO RESUELTO

Desarrollar un método estático que tome un array de enteros como argumento y devuelva el mayor valor encontrado en el array

EJERCICIO RESUELTO

- Partimos de esta versión:

```
public class Mayor {

    /** Devuelve el elemento de mayor valor de una lista ...9 lines */
    public static int obt_mayorNumero(int lista[]) {
        int indice, max = Integer.MAX_VALUE;
        for (indice = 0; indice < lista.length - 1; indice++) {
            if (lista[indice] > max) {
                max = lista[indice];
            }
        }
        return max;
    }
```

EJERCICIO RESUELTO

- ¿Qué pruebas se te ocurren para el método **obt_mayorNumero?**
  - ¿Qué ocurre para un *array con valores cualesquiera (el caso* normal)?
    - [3, 7, 9, 8] > 9
  - ¿Qué ocurre si el mayor elemento se encuentra al principio, en medio o al final del *array*?
    - [9, 7, 8] > 9; [7, 9, 8] > 9; [7, 8, 9] > 9
  - ¿Y si el mayor elemento se encuentra duplicado en el *array*?
    - [9, 7, 9, 8] > 9
  - ¿Y si sólo hay un elemento en el *array*?
    - [7] > 7
  - ¿Y si el *array está compuesto por elementos negativos*?
    - [-4, -6, -7, -22] > -4
  - ¿Y si el array es null?
    - Disparará una excepción

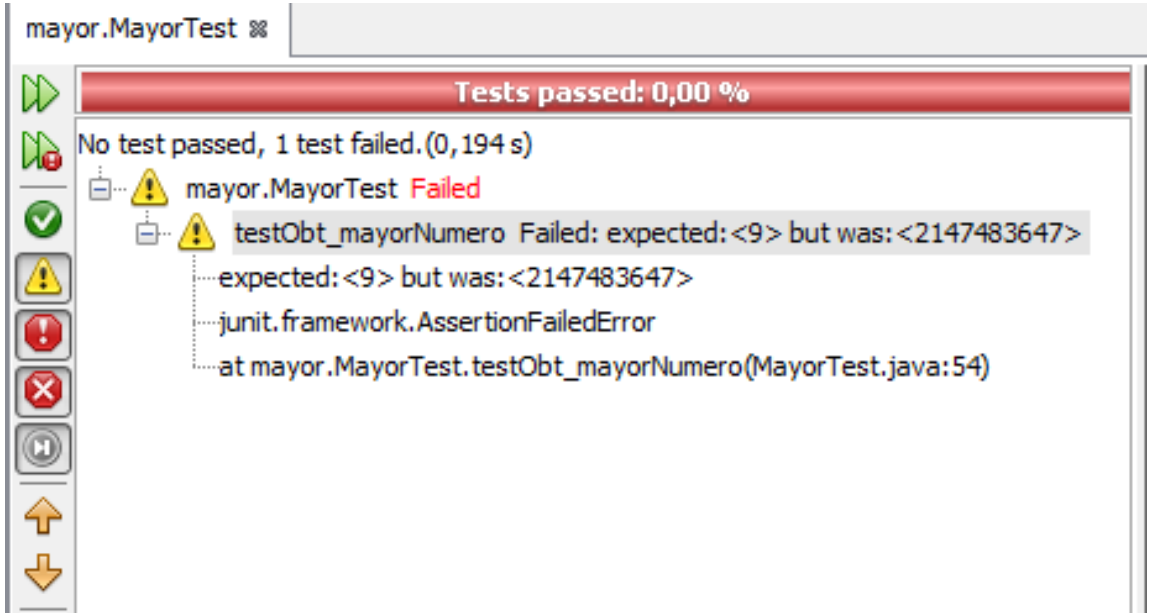
EJERCICIO RESUELTO

- Preparo las pruebas:

```
@Test
public void testObt_mayorNumero() {
    System.out.println("obt_mayorNumero");

    //si lista es null disparará una NullPointerExcepcion
    try {
        assertEquals(0, Mayor.obt_mayorNumero(null));
        fail("deberia haber lanzado una NullPointerException");
    } catch (NullPointerException e) {
    }

    assertEquals(9, Mayor.obt_mayorNumero(new int[]{3, 7, 9, 8}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{9, 7, 8}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{7, 9, 8}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{7, 8, 9}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{9, 7, 9, 8}));
    assertEquals(7, Mayor.obt_mayorNumero(new int[]{7}));
    assertEquals(-4, Mayor.obt_mayorNumero(new int[]{-4, -6, -7, -22}));
}
```

EJERCICIO RESUELTO

- No las pasamos:



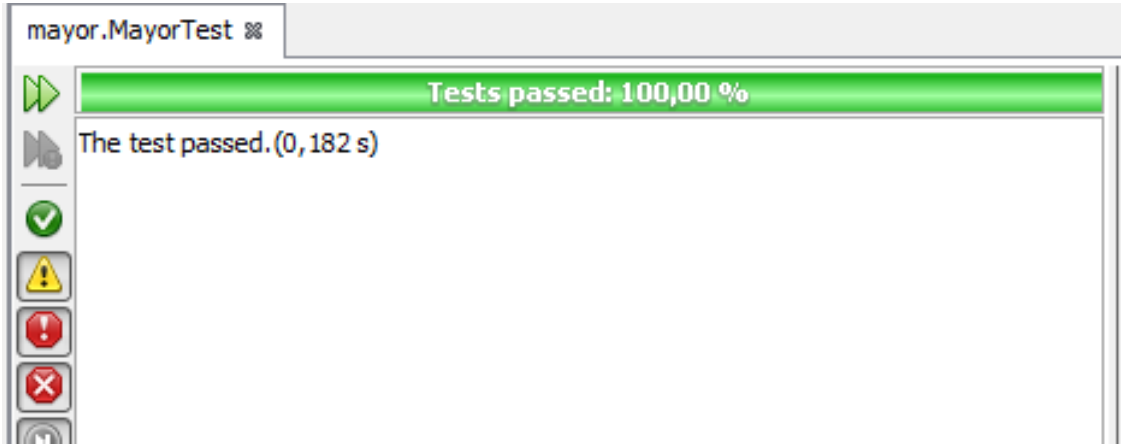
EJERCICIO RESUELTO

- Reviso el código:

```
public static int obt_mayorNumero(int lista[]) {
    //   int indice, max = Integer.MAX_VALUE;
    int indice, max = Integer.MIN_VALUE;
    // for (indice = 0; indice < lista.length - 1; indice++) {
    for (indice = 0; indice < lista.length; indice++) {
        if (lista[indice] > max) {
            max = lista[indice];
        }
    }
    return max;
}
```

EJERCICIO RESUELTO

- Ahora pasamos las pruebas:



EJERCICIO PROPUESTO

- Aunque dentro de un método de prueba podemos poner tantos assert como queramos **es recomendable crear un método de prueba diferente por cada caso de prueba** que tengamos
  - Modifica el método testObt_mayorNumero() que contiene muchos casos de prueba creando varios métodos de prueba distintos: testObt_mayorNumero1(), testObt_mayorNumero2(), …
  - De esta forma cuando se presenten los resultados de las pruebas podremos ver exactamente qué caso de prueba es el que ha fallado

ANOTACIONES JUNIT

- En versiones anteriores de Junit no existían
  - Se han incluido en la versión 4
- Se trata de palabras clave que se colocan antes de los métodos de test e indican a las librerías Junit instrucciones concretas:
- @RunWith, @Before , @After , @Test

@TEST

- La anotación @Test identifica el método que sigue como método de prueba o método test

@Test

public void method()

@Test (expected = Exception.class)

Falla si el método no lanza la excepción esperada

@Test(timeout=100)

Falla si el método tarda más de 100 milisegundos

@BEFORE

- @Before
  - Si anotamos a un método con esta etiqueta el código será ejecutado antes de cada método de prueba
  - Puede haber varios métodos con esta anotación
  - Se usa para preparar el entorno de test
    - Por ejemplo, leer datos de entrada, inicializar la clase, etc

@BEFORE

- ¿Cómo podríamos utilizar @Before en Clase Calculadora?
  - Fíjate que en todos los métodos de test repetimos la instrucción de creación del objeto calcu

    calcu=new Calculadora()
  - Podríamos crear el método creaCalculadora() con la etiqueta @Before
- Creo una clase nueva CalculadoraTest2

```
import org.junit.Test;
import org.junit.After;
import org.junit.Before;

public class CalculadoraTest2 {

    private Calculadora calcu;
    private int resultado;

    @Before
    public void creaCalculadora(){
        calcu=new Calculadora(20,10);

    }

    @Test
    public void testSuma() {
         resultado= calcu.suma();
        assertEquals(30,resultado);
    }
```

@AFTER

- @After
  - Si anotamos un método con esta etiqueta el código será ejecutado después de cada método de prueba
  - Podemos tener varios métodos con esta anotación
  - Se usa para limpiar el entorno de test
    - Por ejemplo borrar datos temporales, restaurar valores por defecto,…
    - Se puede usar también para ahorrar memoria limpiando estructuras de memoria pesadas

@AFTER

- En la clase CalculadoraTest2 vamos a añadir un método que limpie los objetos creados y se ejecute después de las pruebas
- El método se llamará borraCalculadora() y se ejecutará al final de las pruebas de la clase Calculadoratest2

```
import org.junit.Test;
import org.junit.After;
import org.junit.Before;

public class CalculadoraTest2 {

    private Calculadora calcu;
    private int resultado;

    @Before
    public void creaCalculadora(){
        calcu=new Calculadora(20,10);

    }

    @After
    public void borraCalculadora(){
        calcu= null;

    }
```

@BEFORECLASS Y @AFTERCLASS

- @BeforeClass
  - Solo puede haber un método con esta etiqueta
  - El método marcado con esta anotación es invocado una vez antes de ejecutar todas las pruebas
  - Se usa para ejecutar actividades intensivas como conectar a una base de datos

- @AfterClass
  - Solo puede haber un método con esta anotación
  - Este método será invocado una sola vez cuando finalicen todas las pruebas
  - Se usa para actividades de limpieza, como por ejemplo, desconectar de la base de datos

- Los métodos marcados @BeforeClass y @AfterClass necesitan ser definidos como static

@BEFORECLASS Y @AFTERCLASS

- Los métodos anotados como @BeforeClass y @AfterClass deben ser **static** y por tanto los atributos a los que acceden también
- Creamos CalculadoraTest3 y añadimos métodos creaCalculadora y borraCalculadora con @BeforeClass y@AfterClass
- Ojo son de tipo static!!

```
import static org.junit.Assert.*;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

public class CalculadoraTest3 {

    private static Calculadora calcu;
    private int resultado;

    @BeforeClass
    public static void creaCalculadora(){
        calcu=new Calculadora(20,10);

    }

    @AfterClass
    public static void borraCalculadora(){
        calcu= null;

    }
```

@IGNORE

- @Ignore Ignora el método de test
- Es útil cuando el código a probar ha cambiado y el caso de uso no ha sido todavía adaptado
- O si el tiempo de ejecución del método de test es demasiado largo para ser incluido

PRUEBAS PARAMETRIZADAS

- Supongamos queremos ejecutar una prueba varias veces con distintos valores de entrada
  - Por ejemplo: vamos a probar el método *divide() con diferentes valores*
- JUnit nos permite generar parámetros para lanzar varias veces una prueba con dichos parámetros
- Para conseguir esto seguimos dos pasos:

PRUEBAS PARAMETRIZADAS

- Paso 1:
  - Añadimos la etiqueta @RunWith(Parameterized.class) a la clasetest
    - Requerirá nuevos import
  - Con esto indicamos a la clase que va a ser usada para realizar una batería de pruebas
  - En esta clase se debe declarar
    - un atributo por cada uno de los parámetros de la prueba
    - y un constructor con tantos argumentos como parámetros en cada prueba
- Ejemplo:
  - Para probar el método divide (o cualquier otro) definiremos 3 parámetros, dos de ellos para los números con los que se realiza la operación y el tercero para recoger el resultado

PRUEBAS PARAMETRIZADAS

- Creamos una nueva clase Calculadora4Test

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class Calculadora4Test {

    private int num1;
    private int num2;
    private int resul;

    public Calculadora4Test(int num1, int num2, int resul) {
        this.num1 = num1;
        this.num2 = num2;
        this.resul = resul;
    }
```

PRUEBAS PARAMETRIZADAS

- Paso 2:
  - Definimos un método anotado con la etiqueta @Parameters que será el encargado de devolver la lista de valores a probar
  - En este método se definirán filas de valores para num1, num2 y resul (en el mismo orden que en el constructor)

PRUEBAS PARAMETRIZADAS

- Ejemplo:
  - Un grupo de valores de prueba seria {20, 10, 2}
    - Para la división equivale a resul=num1/num2 , esto es, 2=20/10 (sería un caso de prueba correcto)
  - Otros grupos de valores de prueba seria {30, -2, -15}(correcto) y {5, 2, 3} (incorrecto)

```
@Parameters
public static Collection<Object[]> numeros() {
    return Arrays.asList(new Object[][]{{20, 10, 2}, {30, -2 - 15}, {5, 2, 3}});

}
```

Atención! Quédate con la idea aunque no entiendas todo el código Java utilizado

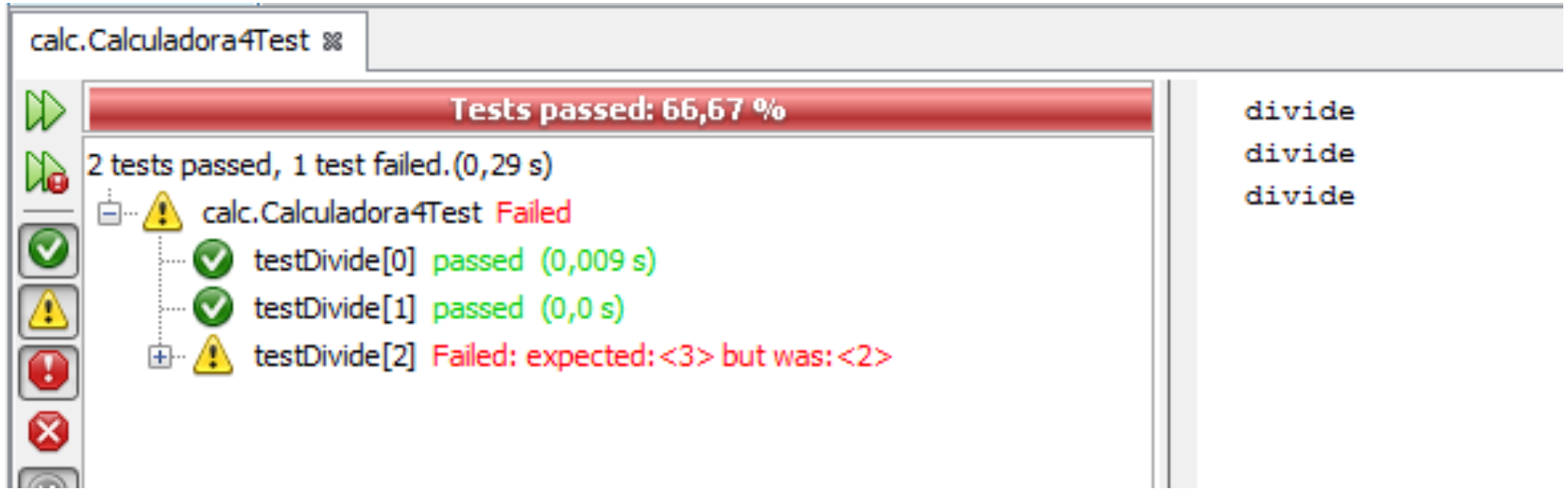
PRUEBAS PARAMETRIZADAS

- El método testDivide() de la clase Calculadora4Test podría ser:

```
@Test
public void testDivide() {

    System.out.println("divide");
    Calculadora instance = new Calculadora(this.num1, this.num2);
    int division = instance.divide();
    assertEquals( this.resul, division);

}
```

PRUEBAS PARAMETRIZADAS

- La ejecución produce la salida siguiente:



- Al lado del método se muestra entre corchetes la prueba que se trata

SUITE DE PRUEBAS

- JUnit nos proporciona el mecanismo llamado Test Suites que agrupa varias Clases de Prueba para que se ejecuten una tras otra
- Vamos a ver el procedimiento de creación de la suite de pruebas con un Ejercicio Guiado:
  1. Creamos distintas Clases de Prueba
  2. Creamos el Test Suite

EJERCICIO GUIADO

- Creamos pruebas parametrizadas para los métodos suma(), resta() y multiplica()
- El nombre para las clases de prueba es:

CalculadoraSumaTest

CalcultadoraRestaTest

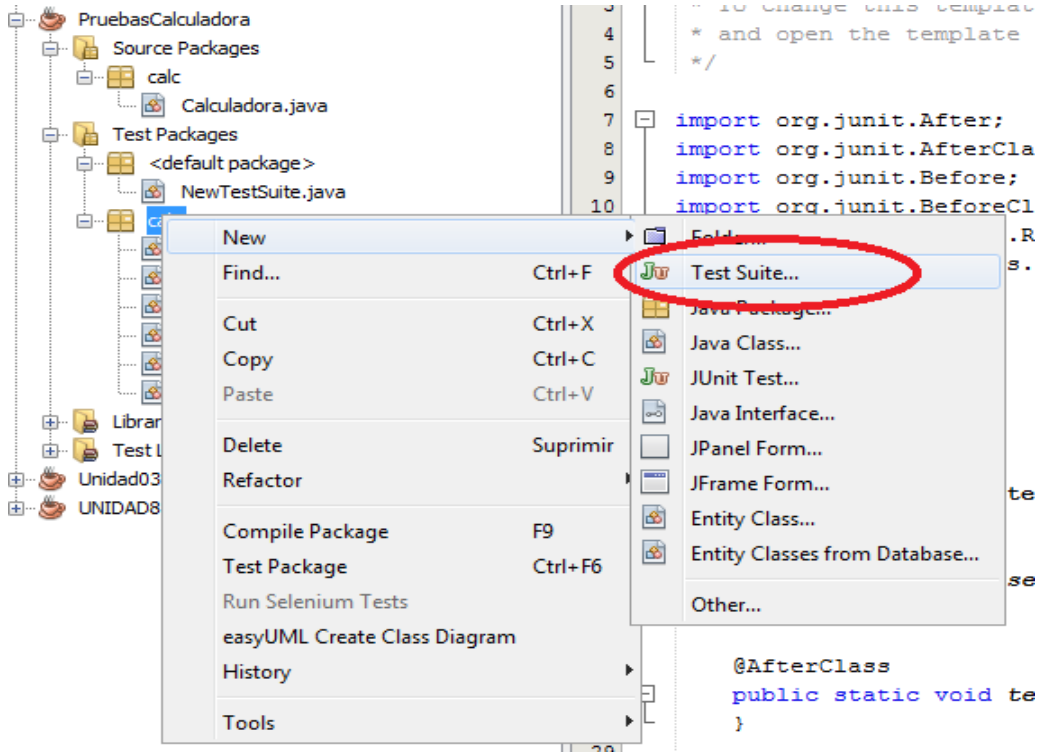
CalculadoraMultiplicaTest

- Atención! Debes crear las otras clases siguiendo el ejemplo

```java
@RunWith(Parameterized.class)
public class CalculadoraSumaTest {

    private int num1;
    private int num2;
    private int resul;

    public CalculadoraSumaTest(int num1, int num2, int resul) {
        this.num1 = num1;
        this.num2 = num2;
        this.resul = resul;
    }

    @Parameters
    public static Collection<Object[]> numeros() {
        return Arrays.asList(new Object[][]{
            {20, 10, 30}, {30, -2, 28}, {5, 2, 7}
        });

    }

    /** Test of suma method, of class Calculadora ...3 lines */
    @Test
    public void testSuma() {
        System.out.println("Suma");
        Calculadora instance = new Calculadora(this.num1, this.num2);
        int calculo = instance.suma();
        assertEquals(this.resul, calculo);
    }
```

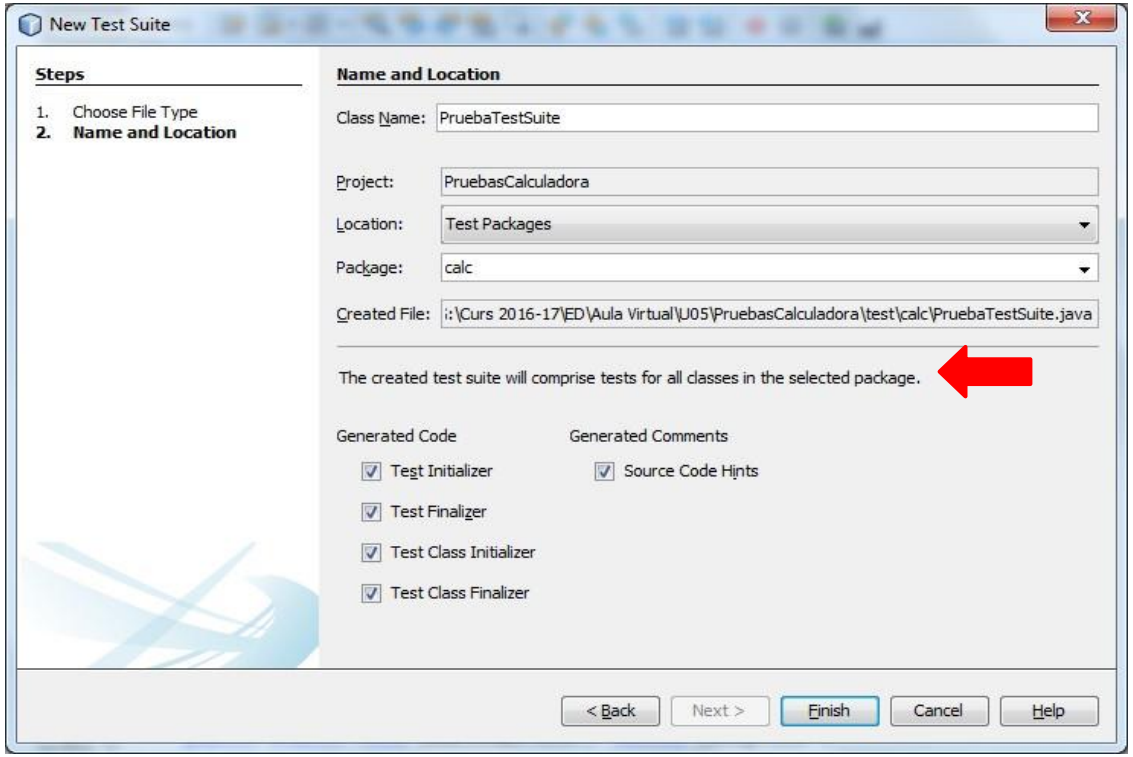
EJERCICIO GUIADO

- Creamos la Suite de Pruebas (Test Suite)



EJERCICIO GUIADO

- Damos nombre a la clase: PruebaTestSuite



EJERCICIO GUIADO

- El asistente genera una SuiteTest con todas las Clases de Prueba del Paquete
- Podemos revisar las que nos interesan:

```
@RunWith(Suite.class)

@Suite.SuiteClasses({calc.CalculadoraSumaTest.class,
    calc.CalculadoraMultiTest.class, calc.CalculadoraRestaTest.class})
```

@RunWith(Suite.class)
Indica a Junit que es una suite de pruebas

@Suite.SuiteClasses()
Indica las clases que forman parte del conjunto de pruebas y que son las que se van a ejecutar

EJERCICIO GUIADO

- Vemos que dentro de la clase no se genera ninguna línea de código
- Únicamente resta ejecutar y se ejecutarán las clases una detrás de otra