

JUnit 4

Pruebas Unitarias

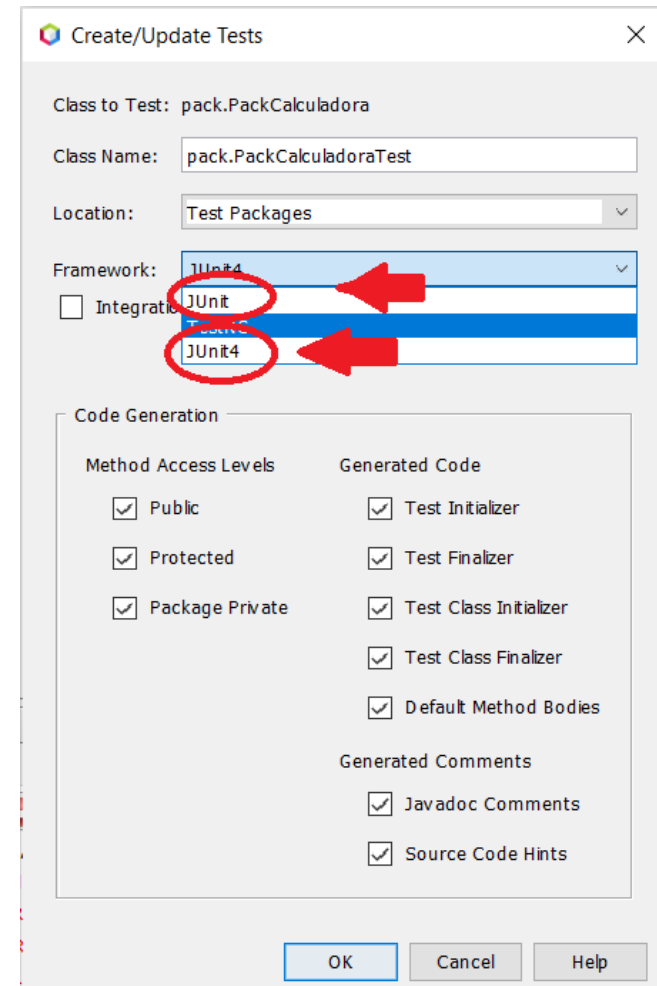
JUnit



- JUnit es una herramienta para realizar pruebas unitarias automatizadas
- JUnit está integrado en NetBeans y Eclipse
- Aquí se presenta JUnit 4.0
 - ▣ Presta atención a la versión que tienes instalada en tu NetBeans o Eclipse

JUnit 4 vs. JUnit 5

- En Apache NetBeans 12 se ofrece por defecto JUnit 5
- Siempre puedo solicitar los Test de Prueba para la versión JUnit 4, tal como ves e la imagen
- En otro documento se presentarán las diferencias con JUnit 5.0

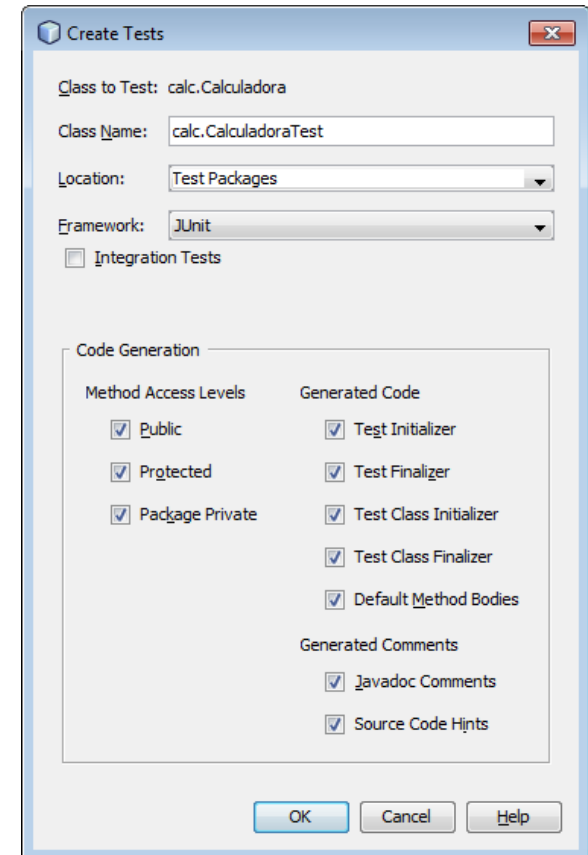


JUnit

- 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**

JUnit

- 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
<code>assertNull (Object objeto)</code> <code>assertNull (string mensaje, Object objeto)</code>	Comprueba que el objeto sea Null. Si no es null y se incluye el String al producirse el error devolverá el mensaje.
<code>assertNotNull (Object objeto)</code> <code>assertNotNull (string mensaje, Object objeto)</code>	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 <u>true</u> . 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 <u>false</u> . 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.

assertEquals (ValorEspDouble, valorRealDobouble, precision)

Comprueba dos valores double o float con la precisión indicada

Métodos JUnit

Métodos	Misión
<code>assertSame (Object objetoEsperado, Object objetoReal)</code> <code>assertSame (String mensaje, Object objetoEsperado, Object objetoReal)</code>	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.
<code>assertNotSame (Object objetoEsperado, Object objetoReal)</code> <code>assertNotSame (String mensaje, Object objetoEsperado, Object objetoReal)</code>	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.
<code>fail()</code> <code>fail (String mensaje)</code>	Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el mensaje.

Más información y opciones:

<https://junit.org/junit4/javadoc/latest/org/junit/Assert.html>

Otros Métodos JUnit

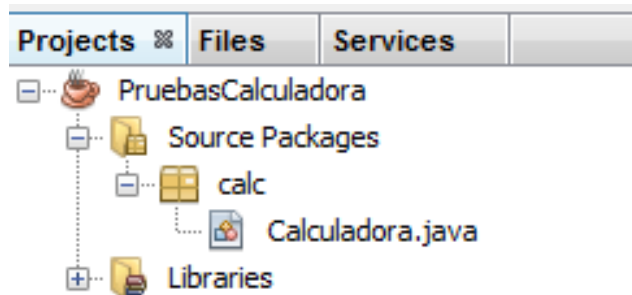
- Inicializadores y Finalizadores:
 - ▣ En algunos casos no es necesario utilizar estos métodos pero siempre se suelen incluir
- El **método setUp** es un método de inicialización de la prueba y se ejecuta antes de cada caso de prueba en la clase de prueba
 - ▣ No es necesario para ejecutar pruebas
 - ▣ 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

Ejercicio Guiado

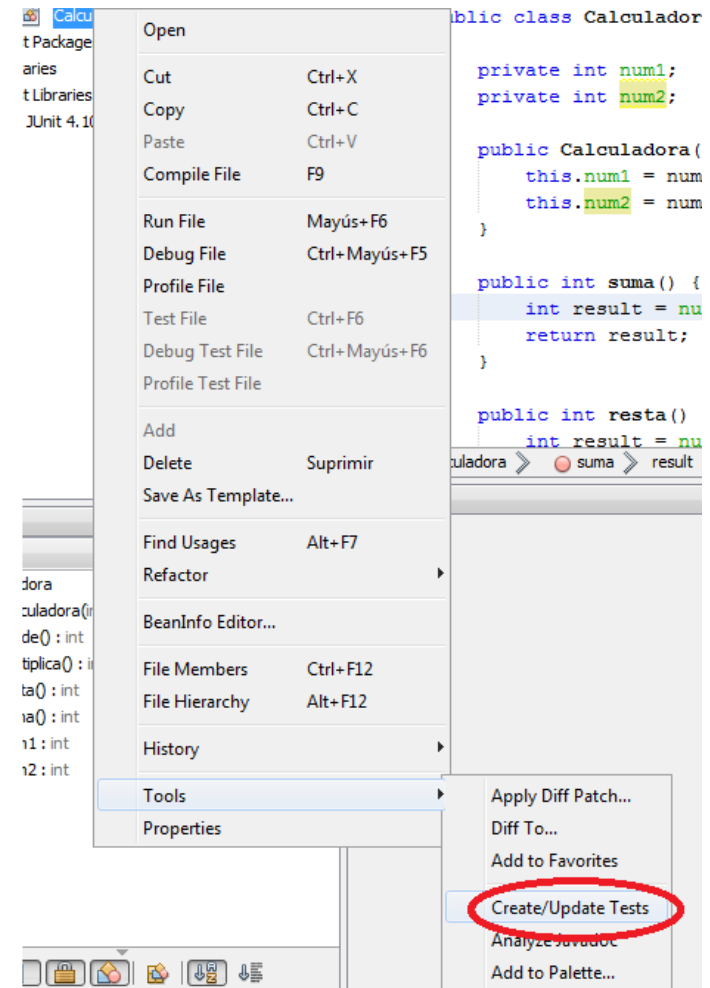
- En NetBeans creamos un nuevo proyecto y creamos la clase Calculadora



```
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    }
}
```

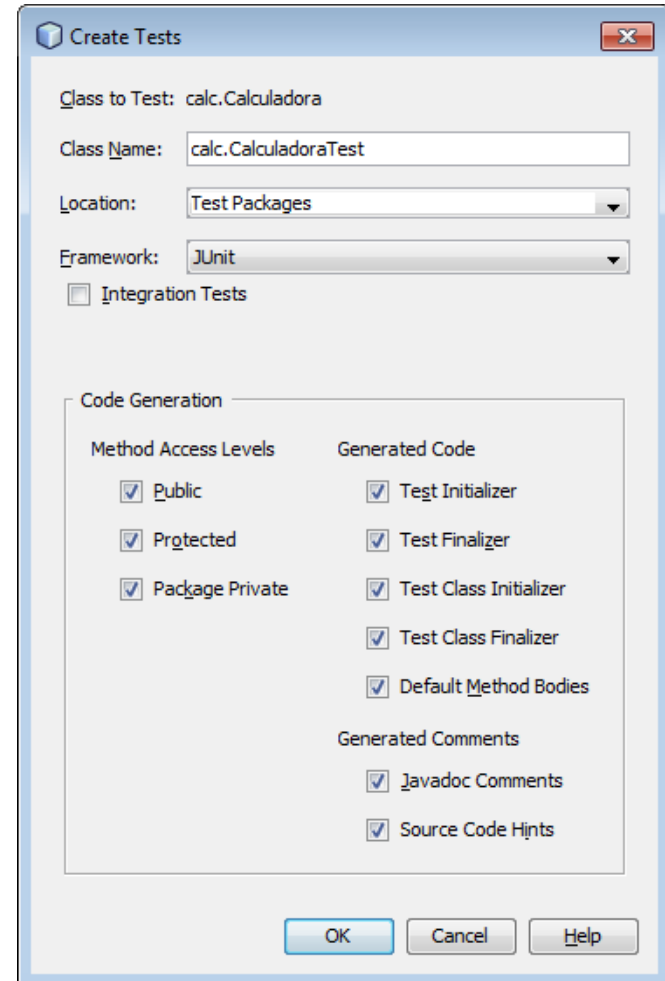
Ejercicio Guiado

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



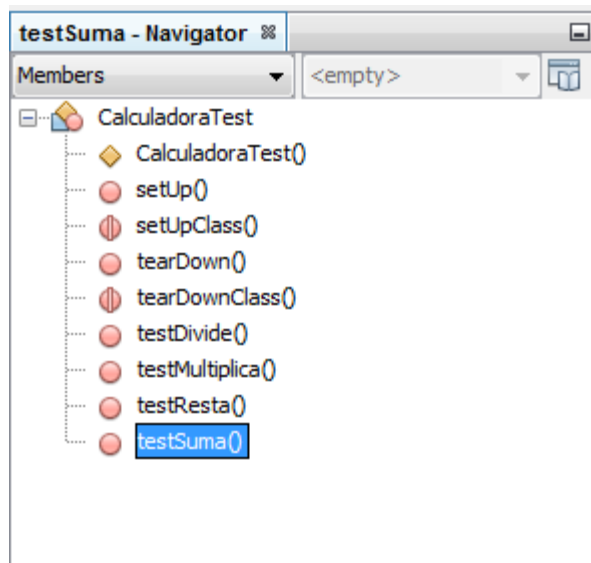
Ejercicio Guiado

- ❑ 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  [Icons]
18  /*
19  public class CalculadoraTest {
20
21  +    public CalculadoraTest() {...2 lines }
22
23
24  @BeforeClass
25  +    public static void setUpClass() {...2 lines }
26
27
28  @AfterClass
29  +    public static void tearDownClass() {...2 lines }
30
31
32  @Before
33  +    public void setUp() {...2 lines }
34
35
36  @After
37  +    public void tearDown() {...2 lines }
38
39
40  +    /** Test of suma method, of class Calculadora ...3 lines */
41  +    @Test
42  +    public void testSuma() {...9 lines }
43
44
45  +    /** Test of resta method, of class Calculadora ...3 lines */
46  +    @Test
47  +    public void testResta() {...9 lines }
48
49
50  +    /** Test of multiplica method, of class Calculadora ...3 lines */
51  +    @Test
52  +    public void testMultiplica() {...9 lines }
53
54
55  +    /** Test of divide method, of class Calculadora ...3 lines */
56  +    @Test
57  +    public void testDivide() {...9 lines }
58
59
60  }
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
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()`

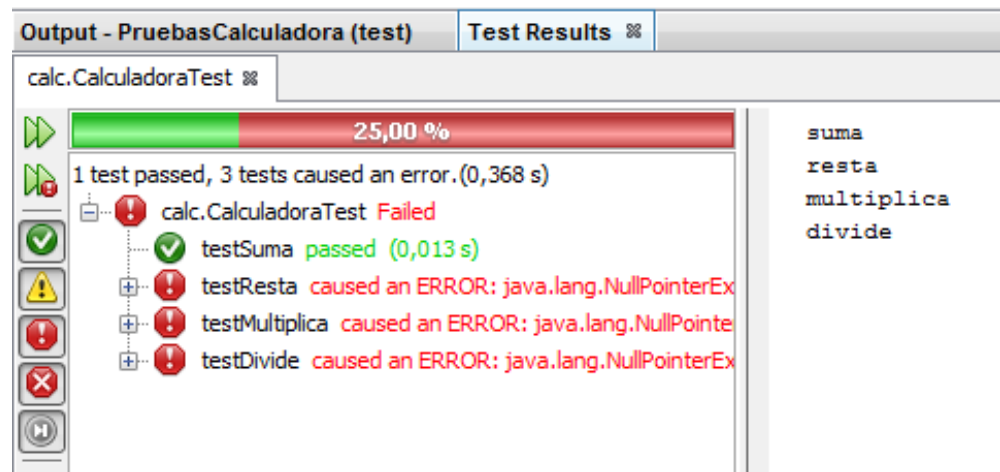
Ejercicio Guiado

□ Modifica testSuma()

```
public void testSuma() {  
    /* System.out.println("suma");  
    Calculadora instance = null;  
    int expectedResult = 0;  
    int result = instance.suma();  
    assertEquals(expectedResult, result);  
    // TODO review the generated test code and remove the default call to fail  
    fail("The test case is a prototype.");  
    */  
  
    System.out.println("suma");  
    Calculadora instance = new Calculadora(20,10);  
    int expectedResult = 30;  
    int result = instance.suma();  
    assertEquals(expectedResult, result);  
    // TODO review the generated test code and remove the default call to fail  
    //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

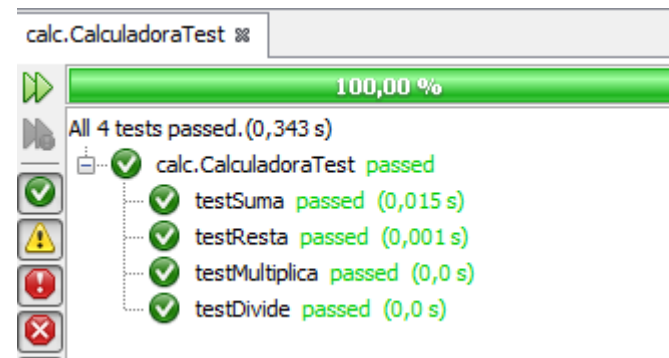
Ejercicio Guiado

- 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 expectedResult = 10;
    int result = instance.resta();
    assertEquals(expectedResult, 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 expectedResult = 200;
    int result = instance.multiplica();
    assertEquals(expectedResult, 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 expectedResult = 2;
    int result = instance.divide();
    assertEquals(expectedResult, result);
}
```



Diferencia fallo y error

- Provocamos fallo en el método multiplica()
 - ▣ hacemos que el valor esperado no coincida con el resultado
 - ▣ 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);
}
```

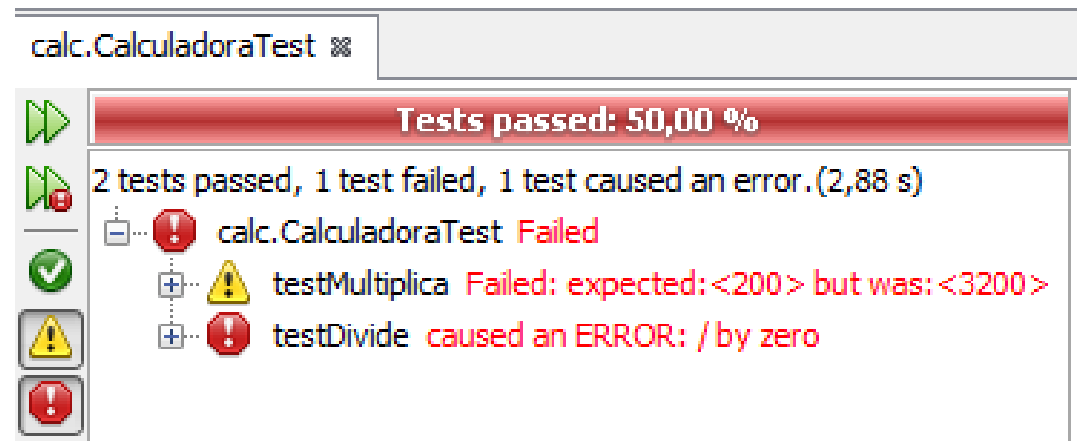
Diferencia fallo y error

- 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:

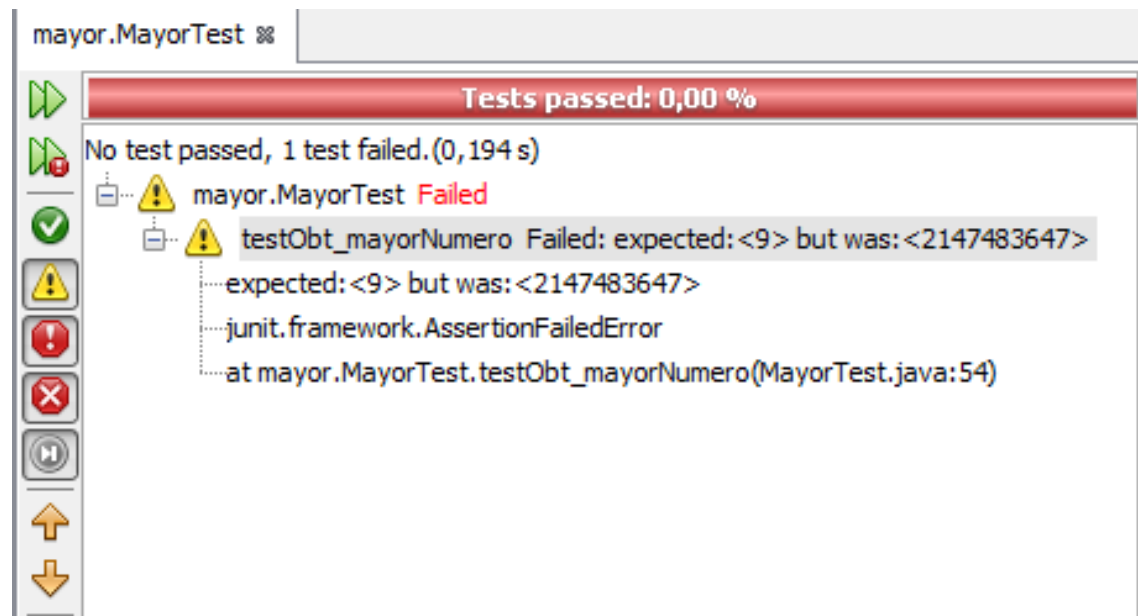
```
@Test
public void testObt_mayorNumero() {
    System.out.println("obt_mayorNumero");

    //si lista es null disparará una NullPointerException
    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:



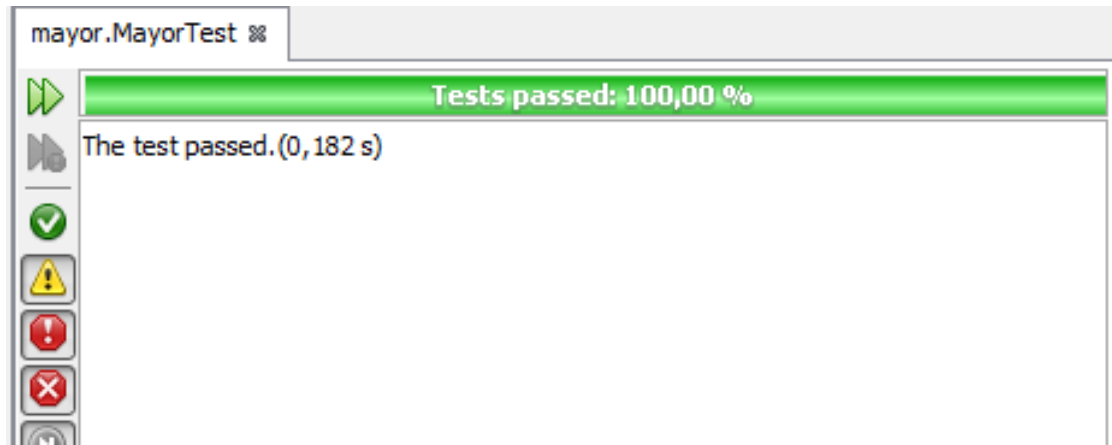
Ejercicio Resuelto

□ 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

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 calculu
calculu=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 calculu;
    private int resultado;

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

    @Test
    public void testSuma() {
        resultado= calculu.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 Calculadortest2

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

public class CalculadoraTest2 {

    private Calculadora calculo;
    private int resultado;

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

    @After
    public void borraCalculadora(){
        calculo= 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 calculu;
    private int resultado;

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

    @AfterClass
    public static void borraCalculadora(){
        calculu= 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
- ❑ ○ si el tiempo de ejecución del método de test es demasiado largo para ser incluido



Pruebas Parametrizadas

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 clase `test`
 - 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 $\text{num1} / \text{num2} = \text{resul}$, esto es, $20 / 10 = 2$ (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:

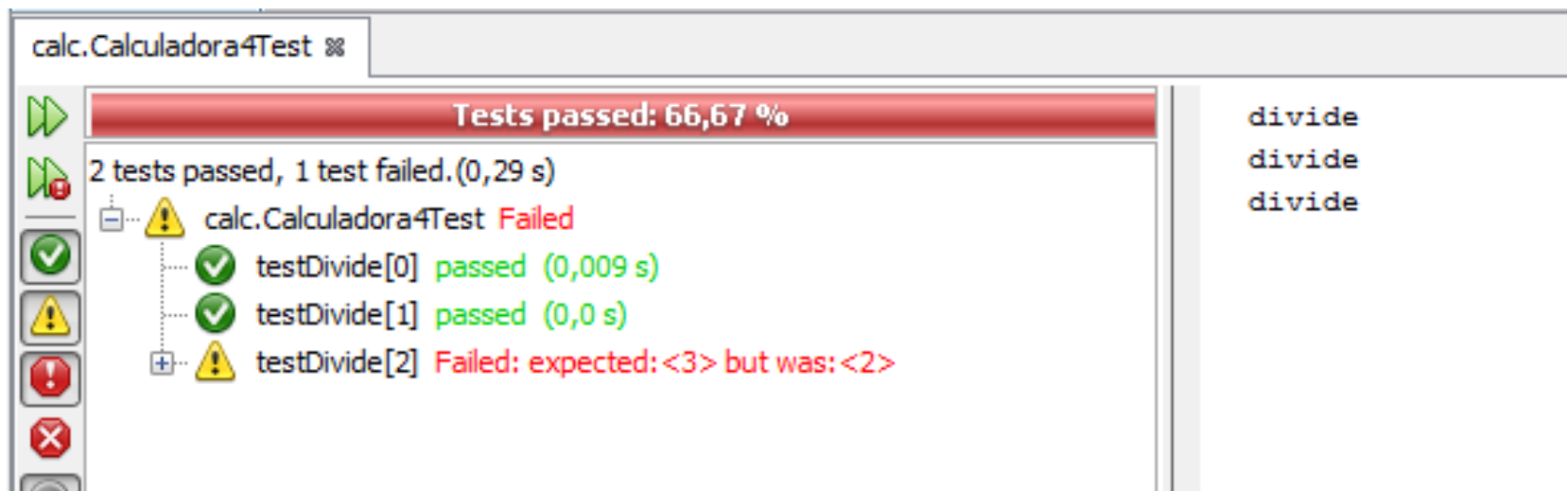
```
@Test
public void testDivide() {

    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

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

CalculadoraRestaTest

CalculadoraMultiplicaTest

- Atención! Debes crear las otras clases siguiendo el ejemplo

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

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

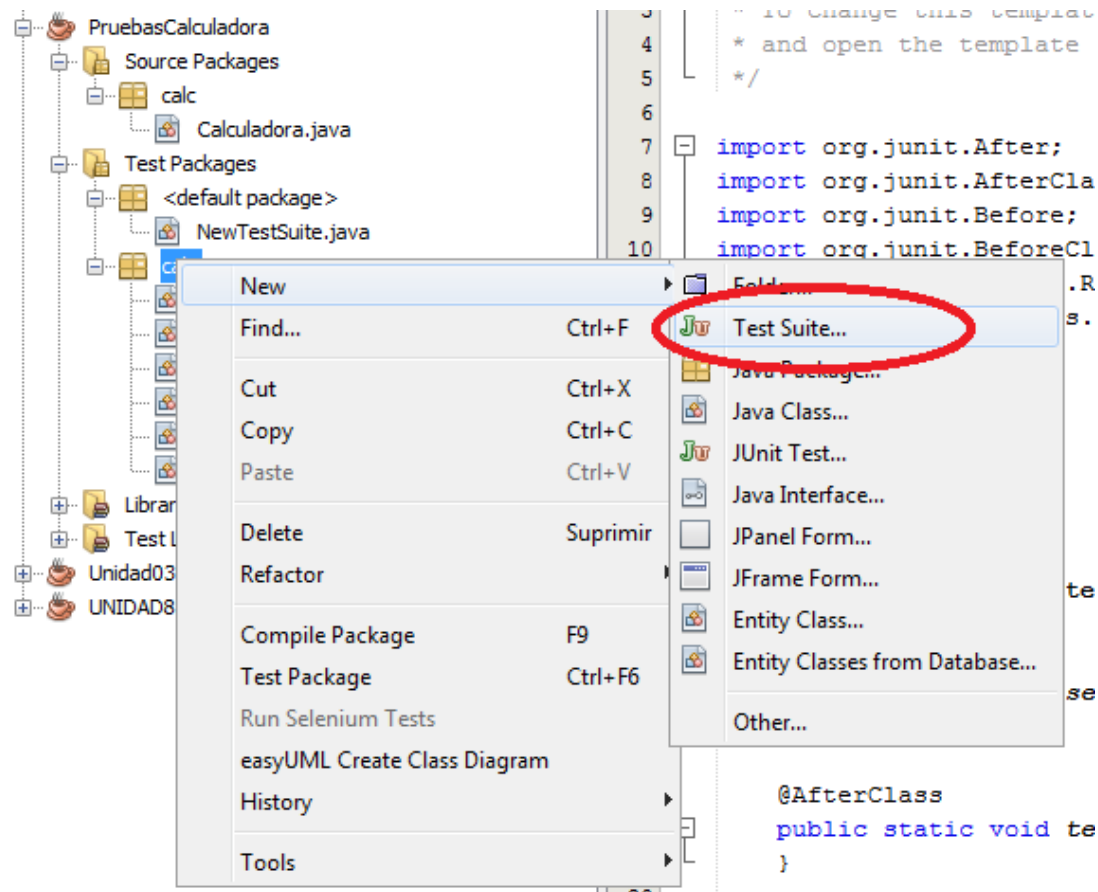
    public CalculadoraSumaTest(int num1, int num2, int resul) {
        this.num1 = num1;
        this.num2 = num2;
        this.resul = resul;
    }

    @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

New Test Suite

Steps

1. Choose File Type
2. **Name and Location**

Name and Location


Class Name:

Project:

Location:

Package:

Created File:

The created test suite will comprise tests for all classes in the selected package. 

Generated Code

- ☒ Test Initializer
- ☒ Test Finalizer
- ☒ Test Class Initializer
- ☒ Test Class Finalizer

Generated Comments

- ☒ Source Code Hints

< Back Next > **Finish** Cancel Help

Ejercicio Guiado

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

```
@RunWith(Suite.class)
```

@RunWith(Suite.class)

Indica a Junit que es una suite de pruebas

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

@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

