

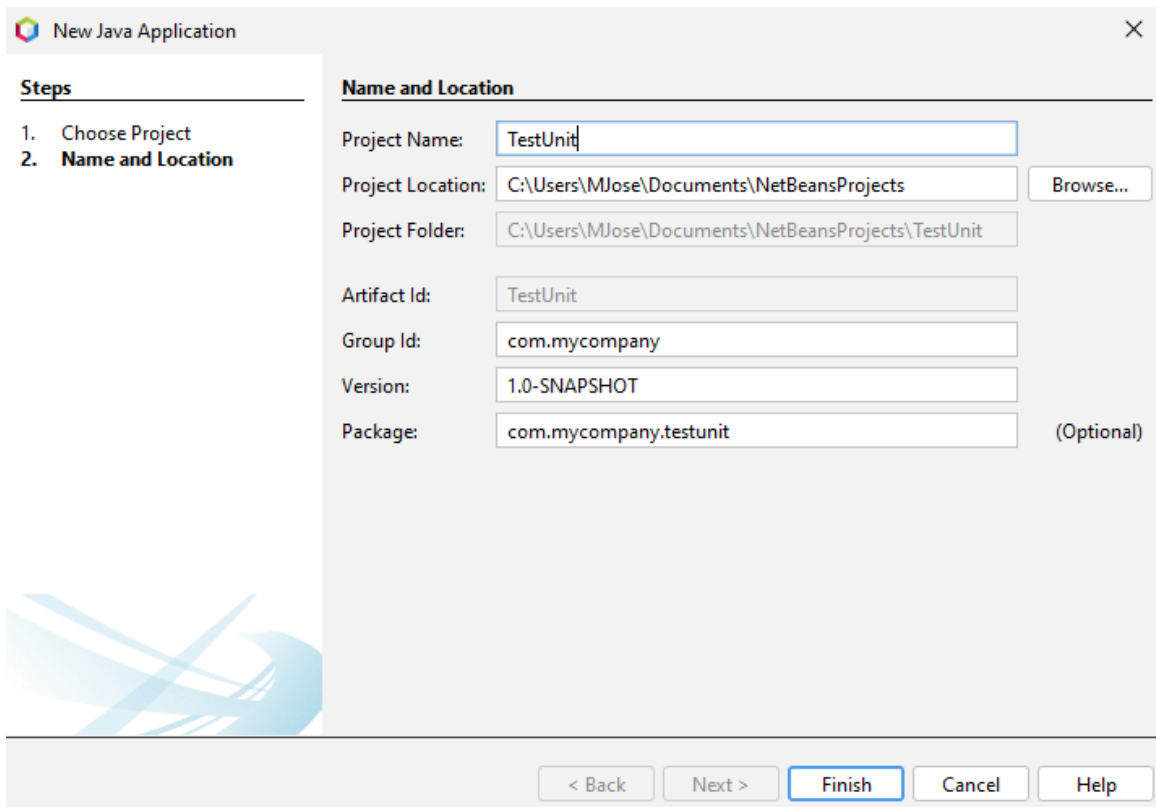
UD03 – PRÁCTICA 2

JUNIT Y PRUEBAS AUTOMÁTICAS

Hasta ahora hemos estado haciendo pruebas de forma manual a partir de una especificación de un código. En esta práctica aprenderemos 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. Está integrada en Eclipse por lo que no es necesario descargarse ningún paquete para poder usarla. Las pruebas unitarias se realizan sobre una clase para probar su comportamiento de modo aislado independientemente del resto de clases de la aplicación (aunque esto no siempre es así porque una clase a veces depende de otras clases para poder llevar a cabo su función).

Para empezar a usar *JUnit* crearemos un nuevo proyecto en Netbeans:

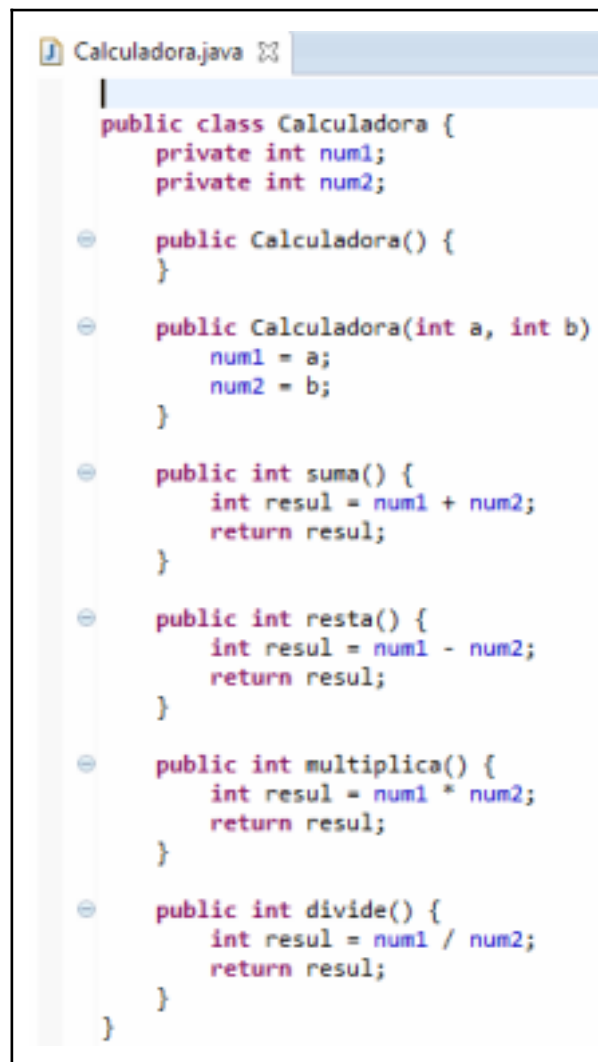


The screenshot shows the 'New Java Application' dialog box in NetBeans. The 'Steps' panel on the left indicates the current step is '2. Name and Location'. The 'Name and Location' tab is active, displaying the following fields:

- Project Name:** TestUnit
- Project Location:** C:\Users\MJose\Documents\NetBeansProjects (with a 'Browse...' button)
- Project Folder:** C:\Users\MJose\Documents\NetBeansProjects\TestUnit
- Artifact Id:** TestUnit
- Group Id:** com.mycompany
- Version:** 1.0-SNAPSHOT
- Package:** com.mycompany.testunit (Optional)

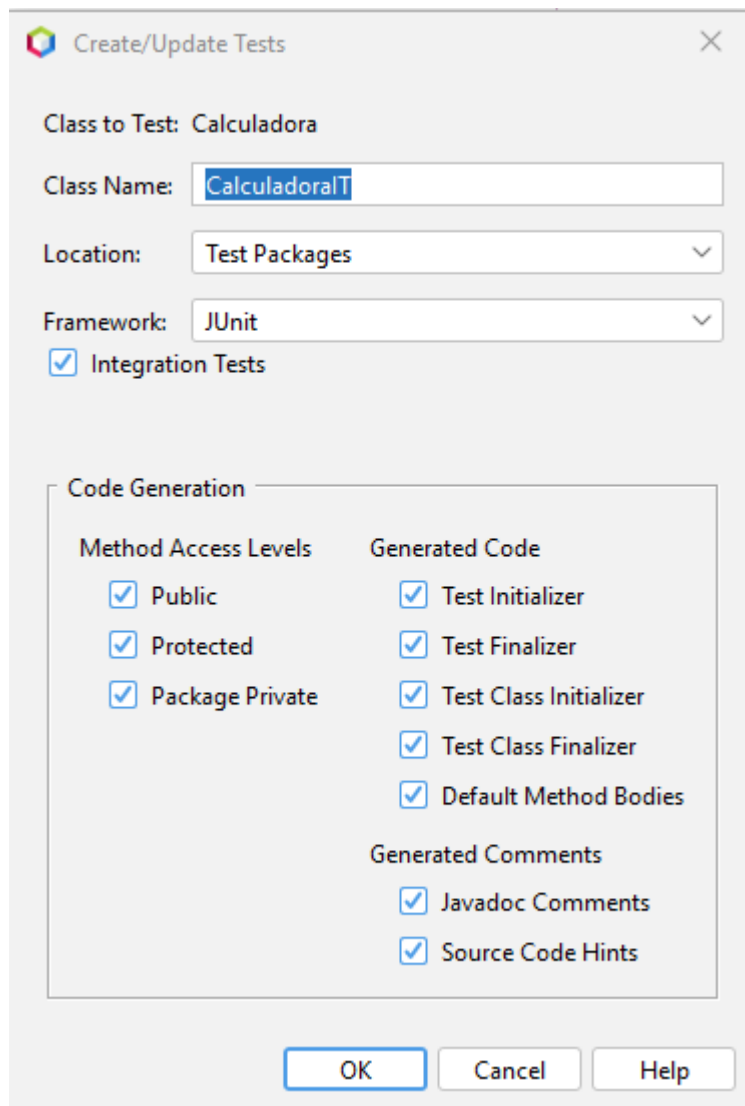
At the bottom of the dialog, there are five buttons: '< Back', 'Next >', 'Finish' (highlighted with a blue border), 'Cancel', and 'Help'.

Crearemos una clase para probar, que en este caso se llamará *Calculadora*:

A screenshot of a Java IDE window titled 'Calculadora.java'. The code defines a public class 'Calculadora' with two private integer attributes, 'num1' and 'num2'. It includes two constructors: a default constructor and a parameterized constructor that takes two integers 'a' and 'b'. There are four public methods: 'suma()' for addition, 'resta()' for subtraction, 'multiplica()' for multiplication, and 'divide()' for division. Each method calculates the result using the attributes and returns it.

```
public class Calculadora {  
    private int num1;  
    private int num2;  
  
    public Calculadora() {  
    }  
  
    public Calculadora(int a, int b) {  
        num1 = a;  
        num2 = b;  
    }  
  
    public int suma() {  
        int resul = num1 + num2;  
        return resul;  
    }  
  
    public int resta() {  
        int resul = num1 - num2;  
        return resul;  
    }  
  
    public int multiplica() {  
        int resul = num1 * num2;  
        return resul;  
    }  
  
    public int divide() {  
        int resul = num1 / num2;  
        return resul;  
    }  
}
```

A continuación hay que crear la clase de prueba. Con la clase *Calculadora* seleccionada pulsaremos el botón derecho del ratón y seleccionaremos tools>create/update tests



Dejamos las opciones con sus valores por defecto. Como nombre de clase se generará el nombre *CalculadoraT*. Pulsaremos *OK*. A continuación hemos de seleccionar los métodos que queremos probar: marcaremos los 4 métodos.

La clase de prueba se crea automáticamente, observándose una serie de características:

- Se crean 4 métodos de prueba, uno por 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 de su nombre original.
- Sobre cada uno de los métodos aparece la anotación *@Test* que indica al compilador que es un método de prueba.
- Cada uno de los métodos de prueba tiene una llamada al método *fail()* con un mensaje indicando que todavía no se ha implementado el método. Este método hace que el test termine con fallo lanzando el mensaje y habrá que dejarlo comentado antes

de proceder a la prueba.

```
@Test
public void testSuma() {
    System.out.println("suma");
    Calculadora instance = new Calculadora();
    int expectedResult = 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.");
}

/**
 * Test of resta method, of class Calculadora.
 */
@Test
public void testResta() {
    System.out.println("resta");
    Calculadora instance = new Calculadora();
    int expectedResult = 0;
    int result = instance.resta();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}

/**
 * Test of multiplicacion method, of class Calculadora.
 */
```

Antes de preparar el código para los métodos de prueba veamos una serie de métodos de *JUnit* para hacer las comprobaciones:

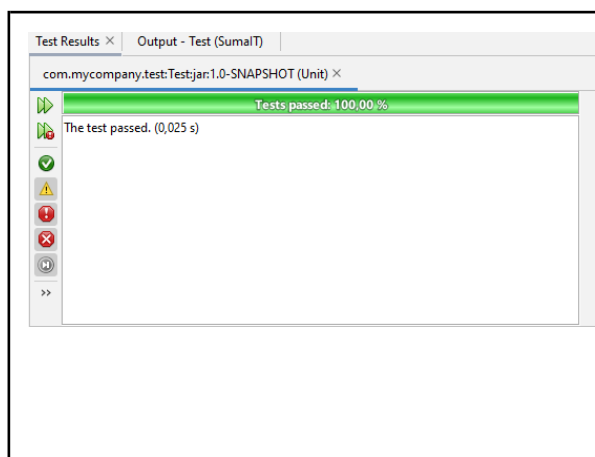
MÉTODOS	MISIÓN
<i>assertTrue(boolean expresión)</i> <i>assertTrue(String mensaje, boolean expression)</i>	Comprueba que la expresión se evalúe a true. Si no es true y se incluye el String, al producirse error se lanzará el mensaje.
<i>assertFalse(boolean expresión)</i> <i>assertFalse(String mensaje, boolean expression)</i>	Comprueba que la expresión se evalúe a false. Si no es false y se incluye el String, al producirse error se lanzará el mensaje.
<i>assertEquals(valorEsperado, valorReal)</i> <i>assertEquals(String mensaje, valorEsperado, valorReal)</i>	Comprueba que el valorEsperado sea igual que el valorReal. Si no son iguales y se incluye el String, entonces se lanzará el mensaje. valorEsperado y valorReal pueden ser de diferentes tipos.
<i>assertNull(Object objeto)</i> <i>assertNull(String mensaje, Object objeto)</i>	Comprueba que el objeto sea null. Si no es null y se incluye el String, al producirse error se lanzará el mensaje.
<i>assertNotNull(Object objeto)</i> <i>assertNotNull(String mensaje, Object objeto)</i>	Comprueba que el objeto no sea null. Si es null y se incluye el String, al producirse error se lanzará el mensaje.

<i>assertSame(Object objetoEsperado, Object objetoReal)</i> <i>assertSame(String mensaje, Object objetoEsperado, Object objetoReal)</i>	Comprueba que objetoEsperado y objetoReal sean el mismo objeto. Si no son el mismo y se incluye el String, al producirse error se lanzará el mensaje.
<i>assertNotSame(Object objetoEsperado, Object objetoReal)</i> <i>assertNotSame(String mensaje, Object objetoEsperado, Object objetoReal)</i>	Comprueba que objetoEsperado y objetoReal no sean el mismo objeto. Si son el mismo y se incluye el String, al producirse error se lanzará el mensaje.
<i>fail()</i> <i>fail(String mensaje)</i>	Hace que la prueba falle. Si se incluye el String, la prueba fallará lanzando el mensaje.

Vamos a crear el código de prueba para el método *testSuma()*, que probará el método *suma()* de la clase *Calculadora*. Lo primero que haremos será crear una instancia de la clase *Calculadora*. Llamaremos al método *suma()* llevando los valores a sumar, por ejemplo 20 y 10, y comprobaremos los resultados con el método *assertEquals()*. En el primer parámetro de este último método escribiremos el resultado esperado al realizar el método *suma()*, en este caso es 30, y como segundo parámetro asignaremos el resultado obtenido al llamar a dicho método:

```
@Test
public void testSuma() {
    Calculadora calculo = new Calculadora(20, 10);
    int resultado = calculo.suma();
    assertEquals(30, resultado);
}
```

Si pulsamos ahora el botón *Run* para ejecutar el test, se mostrarán algunos errores ya que no se han implementado todos los tests de pruebas. También se puede ejecutar la clase de prueba pulsando sobre la clase con el botón derecho del ratón y seleccionando *test file*. En ambos casos se abrirá la pestaña de *JUnit* donde se muestran los resultados de ejecución de las pruebas.



Al lado de cada prueba aparecerá un icono con una marca: una marca de verificación verde indica prueba exitosa, un aspa azul indica fallo y un aspa rojo indica error.

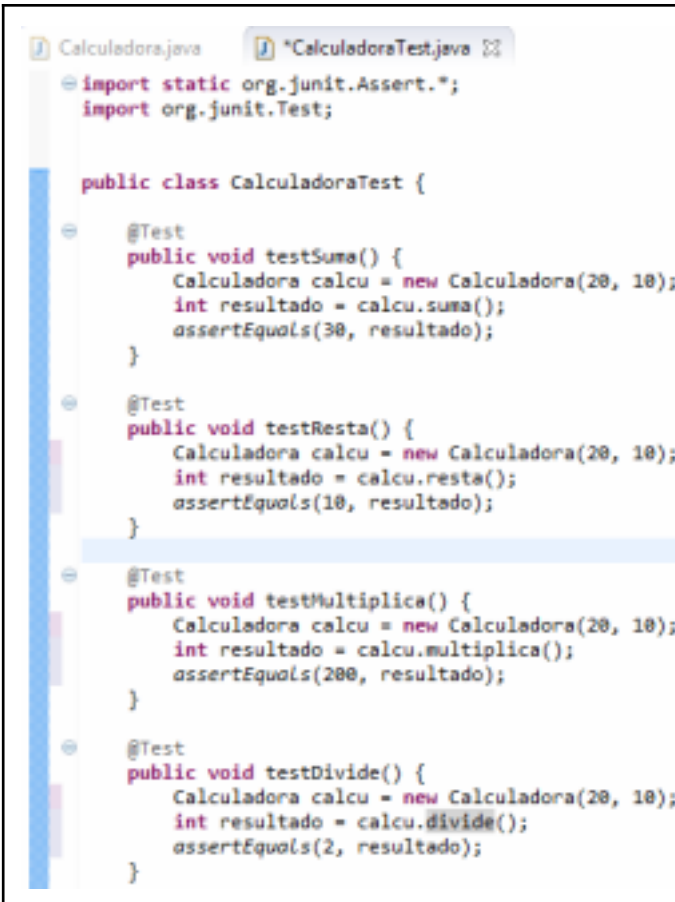
El resultado de la ejecución de la prueba mostrará:

Runs: 4/4 Errors: 0 Failures: 3

Esto nos indica que se han realizado 4 pruebas, ninguna de ellas ha provocado error y 3 de ellas han provocado fallo.

En el contexto de *JUnit* un fallo es una comprobación que no se cumple y un error es una excepción durante la ejecución del código. En esta prueba solo se ha realizado satisfactoriamente la prueba con el método *testSuma()* que mostrará un icono con una marca de verificación al lado. El resto de pruebas habrán fallado y mostrarán el icono con aspa azul (todos los métodos inicialmente incluyen el método *fail()* que hace fallar la prueba).

Rellenaremos 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 previstos (20 y 10):



```
Calculadora.java  *CalculadoraTest.java
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculadoraTest {

    @Test
    public void testSuma() {
        Calculadora calculo = new Calculadora(20, 10);
        int resultado = calculo.suma();
        assertEquals(30, resultado);
    }

    @Test
    public void testResta() {
        Calculadora calculo = new Calculadora(20, 10);
        int resultado = calculo.resta();
        assertEquals(10, resultado);
    }

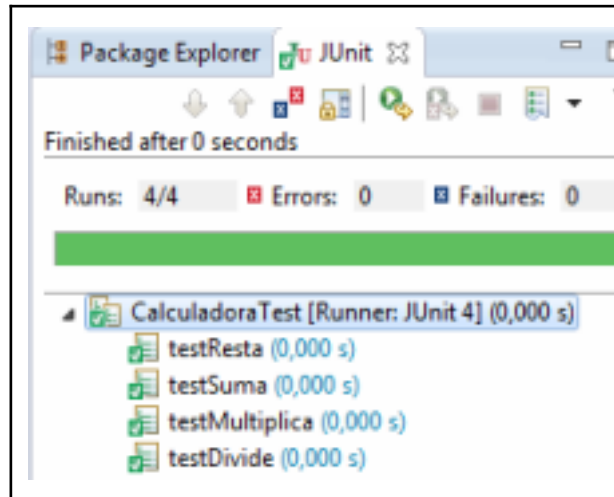
    @Test
    public void testMultiplica() {
        Calculadora calculo = new Calculadora(20, 10);
        int resultado = calculo.multiplica();
        assertEquals(200, resultado);
    }

    @Test
    public void testDivide() {
        Calculadora calculo = new Calculadora(20, 10);
        int resultado = calculo.divide();
        assertEquals(2, resultado);
    }
}
```

Ahora el ejecutar la clase de prueba el resultado que se mostrará es:

Runs: 4/4 Errors: 0 Failures: 0

Eso nos indica que se han realizado 4 pruebas, ninguna ha provocado error y ninguna ha provocado fallo.

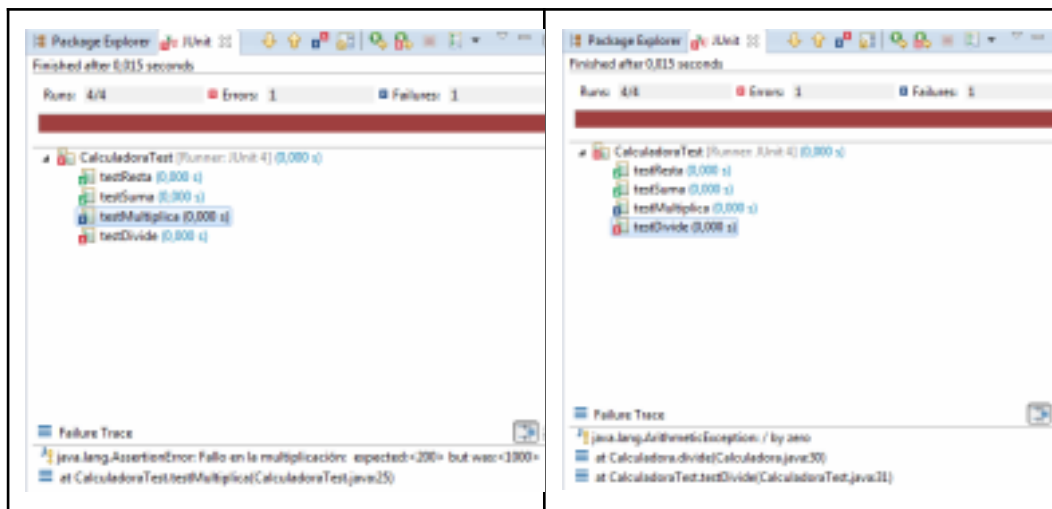


Para ver la diferencia entre un fallo y un error cambiaremos el código de dos de los métodos de prueba. Para hacer que el método *multiplica()* produzca un fallo haremos que el valor esperado no coincida con el resultado; se incluirá un *String* en el método *assertEquals()* para que si se produce el fallo se lance el mensaje. Para que el método *divide()* produzca un error, al crear el objeto calculadora asignaremos el valor 0 al segundo parámetro (será el denominador de la división y, al dividir por cero, producirá una excepción). El código de los métodos es el siguiente:

```
@Test
public void testMultiplica() {
    Calculadora calculo = new Calculadora(20, 50);
    int resultado = calculo.multiplica();
    assertEquals("Fallo en la multiplicación: ", 200, resultado);
}

@Test
public void testDivide() {
    Calculadora calculo = new Calculadora(20, 0);
    int resultado = calculo.divide();
    assertEquals(2, resultado);
}
```

Al ejecutar la prueba, el botón *Filter Stack Trace* mostrará la traza completa de ejecución. Al pulsar en los test que han producido fallos o errores se mostrara la traza de ejecución.



El siguiente test comprobará que la llamada al método *divide()* devuelve la excepción *ArithmeticException* al dividir por cero. Por tanto saldrá del mismo por la cláusula *match* correspondiente. Si no se lanza la excepción, se lanzará el método *fail()* con un mensaje indicando que se ha producido un fallo al probar el test. La prueba tiene éxito si se produce la excepción y falla en caso contrario:

```
@Test
public void testDivide() {
    try {
        Calculadora calculo = new Calculadora(20, 0);
        int resultado = calculo.divide();
        fail("FALLO, debería haber lanzado la excepción");
    }
    catch (ArithmeticException e) {
        // PRUEBA satisfactoria
    }
}
```

ACTIVIDAD PROPUESTA

Modificar el método *resta()* de la clase *Calculadora* y añadir los métodos *resta2()* y *divide2()* que se exponen a continuación. Crear después los test para probar los 3 métodos:

```
public int resta() {
    int resul;
    if (resta2())
        resul = num1 - num2;
    else
        resul = num2 - num1;
    return resul;
}

public boolean resta2() {
```



```

        if (num1 >= num2)
            return true;
        else
            return false;
    }

    public Integer divide2() {
        if (num2 == 0)
            return null;
        int resul = num1 / num2;
        return resul;
    }

```

Utiliza los métodos *assertTrue()*, *assertFalse()*, *assertNull()*, *assertNotNull()* o *assertEquals()* según convenga.

En el caso de *divide2* se producirá un error al utilizar el método *assertNull()* ya que el primero no devuelve un objeto de la clase *Integer* en todos los casos. Cambia el tipo *int* a la variable local *resul* poniéndole *Integer* en su lugar.

Para probar un método que puede lanzar excepciones se utiliza el parámetro *expected* con la anotación *@Test*. Por ejemplo, para hacer que el método *divide()* lance la excepción *ArithmeticException* si el denominador es 0, se puede utilizar la instrucción *throw* de la siguiente manera:

```

    public int divide0() {
        if (num2 == 0)
            throw new Java.lang.ArithmeticException("Division por 0");    else
        {
            int resul = num1 / num2;
        }
        return resul;
    }

```

En la clase de prueba, para poder verificar que se lanza esa excepción se utilizó el parámetro *expected* de la siguiente manera:

```

@Test (expected = java.lang.ArithmeticException.class)
public void testDivide0() {
    Calculadora calcu = new Calculadora(20, 0);
    Integer resultado = calcu.divide0();
}

```

La prueba fallará si no se produce la excepción.

En la vista de *JUnit* se muestran varios botones:

- *Next Failed Test* navega a la siguiente prueba que ha producido fallo o error.
- *Previous Failed Test* navega a la anterior prueba que ha producido fallo o error.
- *Show Failures Only* muestra solo las pruebas que han producido fallo o error.
- *Scroll Lock* activa o desactiva el bloqueo de desplazamiento de pantalla.
- *Rerun*

Test vuelve a ejecutar las pruebas.

- *Rerun Test – Failures First* vuelve a ejecutar las pruebas, ejecutando en primer lugar los fallos y errores.
- *Stop JUnit Test Run* detiene la ejecución de las pruebas.
- *Test Run History* muestra el historial de las pruebas realizadas anteriormente.

En todos los métodos de prueba anteriores se repetía la línea:

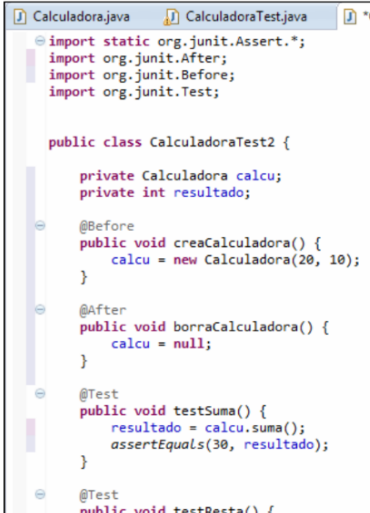
```
Calculadora calculo = new Calculadora(20, 10);
```

Esta sentencia de inicialización se puede escribir una sola vez dentro de la clase.

JUnit dispone de una serie de anotaciones que permiten ejecutar código antes y después de las pruebas:

- *@Before*: si anotamos un método con esta etiqueta, el código será ejecutado antes de cualquier método de prueba. Este método se puede utilizar para inicializar datos: por ejemplo, en una aplicación de acceso a base de datos se puede preparar ésta última y si vamos a utilizar un *array* para las pruebas se puede inicializar aquí. Puede haber varios métodos en la clase de prueba con esta anotación.
- *@After*: si anotamos un método con esta etiqueta el código será ejecutado después de la ejecución de todos los métodos de prueba: por ejemplo, se puede utilizar para limpiar datos. Puede haber varios métodos en la clase de prueba con esta anotación.

La clase *CalculadoraTest*, incluyendo dos métodos con las anotaciones *@Before* y *@After* quedaría de la siguiente manera (completar con lo que falte):



```
Calculadora.java  CalculadoraTest2.java  "C"

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class CalculadoraTest2 {

    private Calculadora calculo;
    private int resultado;

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

    @After
    public void borraCalculadora() {
        calculo = null;
    }

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

    @Test
    public void testResta() {
```

Otras anotaciones a destacar son `@BeforeClass` y `@AfterClass`. Tienen algunas diferencias respecto a las anteriores:

- `@BeforeClass`: solo puede haber un método con esta etiqueta. El método marcado con esta anotación es invocado una vez al principio del lanzamiento de todas las pruebas. Se suele utilizar para inicializar atributos comunes a todas las pruebas o para realizar acciones que tardan un tiempo considerable en ejecutarse.
- `@AfterClass`: solo puede haber un método con esta anotación. Este método será invocado una sola vez cuando finalicen todas las pruebas.

En este caso los métodos anotados deben ser *static* y por tanto los atributos a los que acceden también:



Supongamos ahora 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:

Debemos añadir la etiqueta `@RunWith(Parameterized.class)` a la clase de prueba. Con esto indicamos que la clase 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 haya en cada prueba. Para probar el método `divide()` o cualquiera de los demás métodos definiremos 3 parámetros, dos de ellos para los números con los que se realiza la operación y el tercero para recoger el resultado; el constructor tendrá el siguiente aspecto:

```
public CalculadoraTest4(int nume1, int nume2, int resul)
```

Lo siguiente es definir 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 *nume1*, *nume2* y *resul* (en el mismo orden en que están definidos en el constructor). Por ejemplo, un grupo de valores de prueba sería {20, 10, 2}, que para la división equivale a la operación $resul = nume1 / nume2$, es decir $2 = 20 / 10$ (sería un caso de prueba correcto).

En el siguiente ejemplo para probar el método *divide()* usamos tres casos de prueba, dos de ellos correctos y uno con resultado erróneo. La clase *CalculadoraTest4* quedaría así:

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

@RunWith(Parameterized.class)
public class CalculadoraTest4 {

    private int nume1;
    private int nume2;
    private int resul;

    public CalculadoraTest4(int nume1, int nume2, int resul) {
        this.nume1 = nume1;
        this.nume2 = nume2;
        this.resul = resul;
    }

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

    @Test
    public void testDivide() {
        Calculadora calculo = new Calculadora (nume1, nume2);
        int resultado = calculo.divide();
        assertEquals(resul, resultado);
    }
}
```

La ejecución produce la siguiente salida. Al lado del método de prueba se muestra entre corchetes la prueba de que se trata. En este caso no se prueban todos los métodos a la vez ya que la lista de valores para las pruebas se ha preparado para la operación de dividir.

