

## 3.7. PRUEBAS UNITARIAS CON JUNIT

Hasta ahora hemos estado haciendo pruebas de forma manual a partir de una especificación de un código. En este apartado 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.

### 3.7.1. Creación de una clase de prueba

Para empezar a usar JUnit creamos un nuevo proyecto en Eclipse y creamos la clase a probar, en este caso se llama *Calculadora*:

```
public class Calculadora {  
    private int num1;  
    private int num2;  
  
    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 pulsamos el botón derecho del ratón y seleccionamos *New >> JUnit Test Case*, véase Figura 3.24.

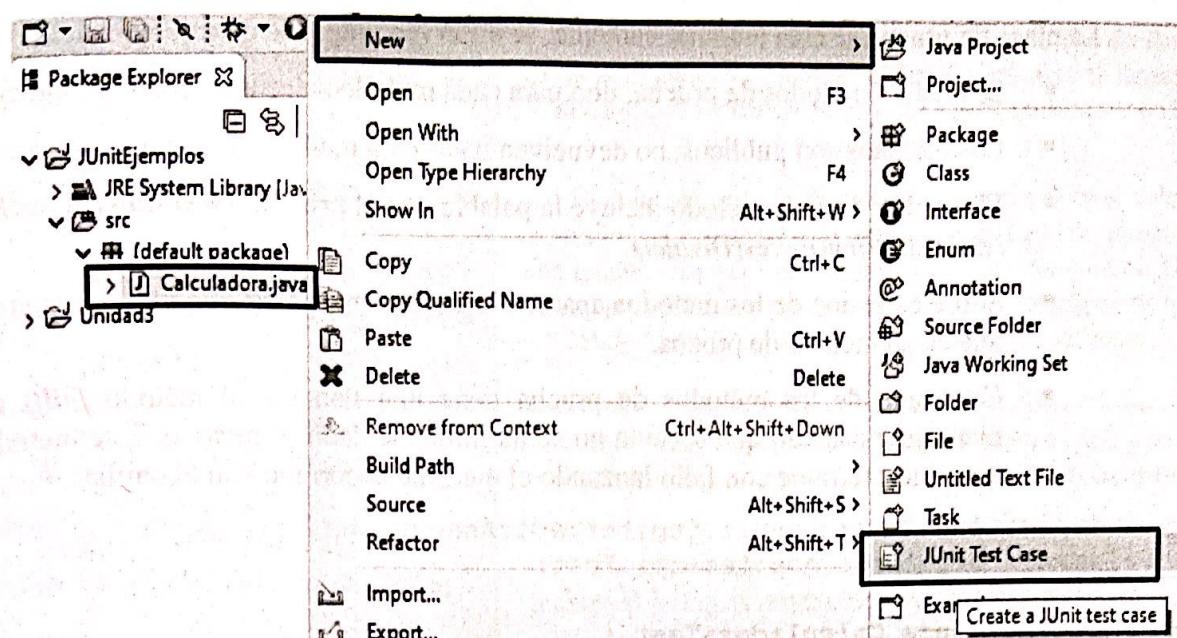


Figura 3.24. Opción New&gt;&gt;JUnit Test Case.

También se puede hacer desde el menú **File >> New >> JUnit Test Case**. En cualquier caso se abre una ventana de diálogo. Desde aquí debemos marcar **New JUnit Jupiter test**, el resto de opciones dejamos los valores por defecto, como nombre de clase se generará el nombre **CalculadoraTest**. Pulsamos el botón **Next**. A continuación, hemos de seleccionar los métodos que queremos probar, marcamos los 4 métodos y pulsamos **Finish**. Se abre una ventanita indicándonos que la librería **JUnit 5** no está incluida en nuestro proyecto, pulsamos el botón **OK** para que se incluya, véase Figura 3.25.

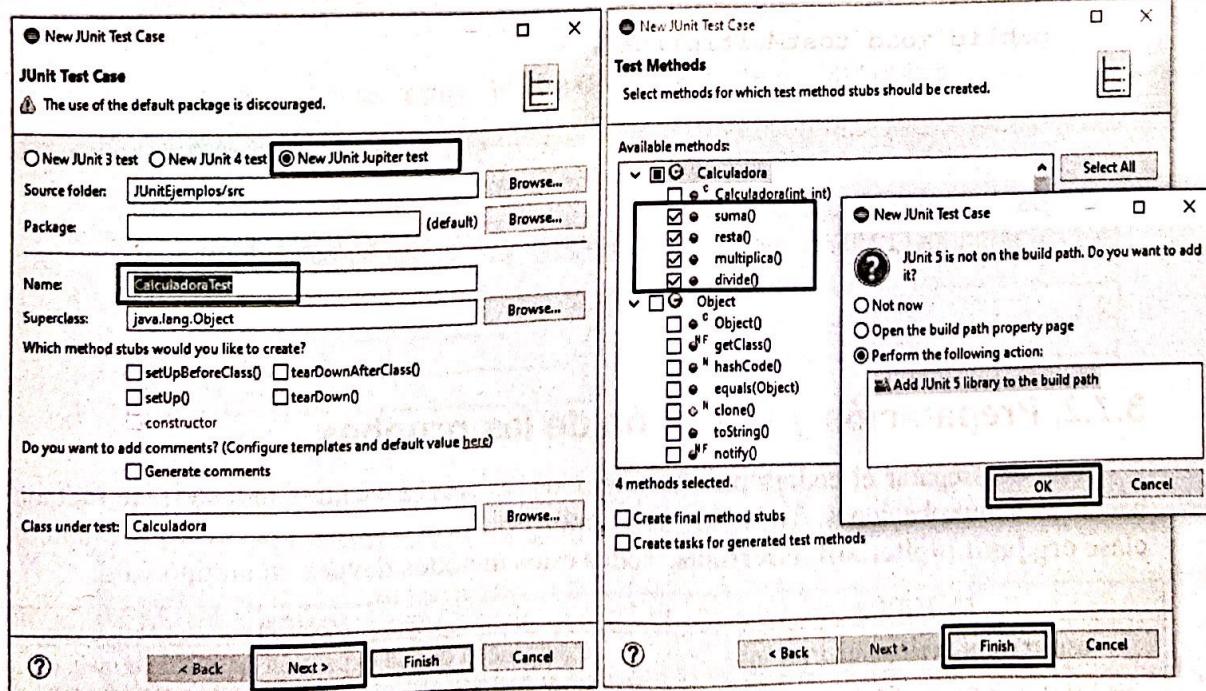


Figura 3.25. Creación de la clase de prueba.

Para que la clase de prueba se incluya en un paquete diferente al de la clase a probar se escribe el nombre del paquete en el campo **Package** (en el ejemplo la clase de prueba y la clase a probar están dentro del mismo paquete: **default package**).

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

- Se crean 4 métodos de prueba, 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.
- 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 encerrado entre comillas.

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.Test;
```

```
public class CalculadoraTest {  
  
    @Test  
    public void testSuma() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    public void testResta() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    public void testMultiplica() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    public void testDivide() {  
        fail("Not yet implemented");  
    }  
}
```

### 3.7.2. Preparación y ejecución de las pruebas

Antes de preparar el código para los métodos de prueba veamos una serie de métodos para hacer las comprobaciones. Todas las aserciones de *JUnit Jupiter* son métodos estáticos en la clase `org.junit.jupiter.api.Assertions`. Todos estos métodos devuelven un tipo `void`:

MÉTODOS	MISIÓN
<code>assertTrue(boolean expresión)</code> <code>assertTrue(boolean expression, String mensaje)</code>	Comprueba que la expresión se evalúe a <i>true</i> . Si no es <i>true</i> y se incluye el String, al producirse error se lanzará el mensaje.
<code>assertFalse(boolean expresión)</code> <code>assertFalse(boolean expression, String mensaje)</code>	Comprueba que la expresión se evalúe a <i>false</i> . Si no es <i>false</i> y se incluye el String, al producirse error se lanzará el mensaje.

<code>assertEquals(double valorEsperado, double valorReal, double delta)</code>  <code>assertEquals(double valorEsperado, double valorReal, double delta, String mensaje)</code>  (Se puede usar con cualquier tipo de dato, Integer, Short, Object, etc; <i>delta</i> se usa en tipos float y double)	Comprueba que el <i>valorEsperado</i> sea igual al <i>valorReal</i> . Si no son iguales y se incluye el String, entonces se lanzará el <i>mensaje</i> . <i>ValorEsperado</i> y <i>valorReal</i> pueden ser de diferentes tipos.  El <i>delta</i> , describe la diferencia admisible entre el valor esperado y el valor real para considerar que ambos números son iguales. Un valor de 0 indica que deben ser iguales. Un valor de 0.01 consideraría estos dos números iguales: 1259.9916 y 1259.9917. Un valor de 0 los consideraría distintos
<code>assertNull(Object objeto), assertNull(Object objeto, String mensaje)</code>	Comprueba que el <i>objeto</i> sea null. Si no es null y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
<code>assertNotNull(Object objeto), assertNotNull(Object objeto, String mensaje)</code>	Comprueba que el <i>objeto</i> no sea null. Si es null y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
<code>assertSame(Object objetoEsperado, Object objetoReal)</code>  <code>assertSame(Object objetoEsperado, Object objetoReal, String mensaje)</code>	Comprueba que <i>objetoEsperado</i> y <i>objetoReal</i> sean el mismo objeto. Si no son el mismo y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
<code>assertNotSame(Object objetoEsperado, Object objetoReal)</code>  <code>assertNotSame(Object objetoEsperado, Object objetoReal, String mensaje)</code>	Comprueba que <i>objetoEsperado</i> no sea el mismo objeto que <i>objetoReal</i> . Si son el mismo y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
<code>fail()</code>  <code>fail(String mensaje):</code>	Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el <i>mensaje</i> .

Más información sobre estos métodos la podemos encontrar en la siguiente web:  
<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>.

Vamos a definir los siguientes casos de prueba para probar los métodos:

MÉTODO A PROBAR	ENTRADA	SALIDA ESPERADA
Suma	20, 10	30
Resta	20, 10	10
Multiplica	20, 10	200
Divide	20, 10	2

Todo caso de prueba se compone básicamente de 3 partes:

1. Se indica en una variable cuál es el valor esperado por el método que vamos a probar.
2. Se ejecuta el método que queremos probar con los datos de entrada adecuados y se guarda el resultado en otra variable.
3. Se comprueba la relación entre el valor esperado y el resultado de la llamada al método (valor real), a través de una aserción. En este caso la aserción es `assertassertEquals()`.

Creamos el código de prueba para el método `testSuma()` que probará el método `suma()` de la clase `Calculadora`:

```

public void testSuma() {
    double valorEsperado = 30;
    Calculadora calcu = new Calculadora(20, 10);
    double resultado = calcu.suma();
    assertEquals(valorEsperado, resultado, 0);
}

```

Para ejecutar la clase de prueba pulsamos en el botón **Run** , o bien pulsamos sobre ella y con el botón derecho del ratón seleccionamos la opción **Run As >> JUnit Test**. Se abre la pestaña de JUnit donde se muestran los resultados de ejecución de las pruebas, véase Figura 3.26.

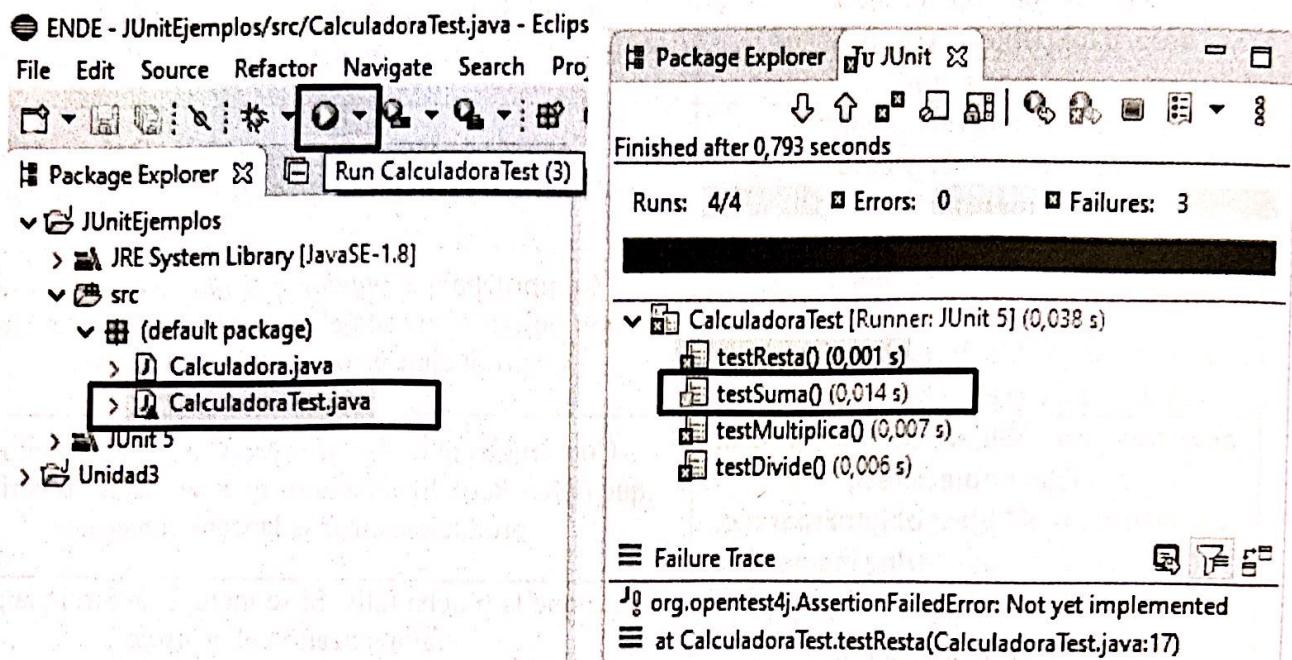


Figura 3.26. JUnit con el resultado de las pruebas.

Al lado de cada prueba aparece un ícono con una marca: una marca de verificación verde indica **prueba exitosa** , un aspa azul  indica **fallo**; y un aspa roja indica **error** . Más adelante veremos la diferencia entre fallo y error. En el panel inferior se muestra información acerca del fallo y el número de línea del caso de test donde se ha producido el fallo.

El resultado de la ejecución de la prueba muestra: **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, un error es una excepción durante la ejecución del código. En esta prueba sólo se ha realizado satisfactoriamente la prueba con el método **testSuma()** que muestra un ícono con una marca de verificación al lado , el resto de pruebas han fallado, muestran el ícono con aspa azul  (recordemos que todos los métodos inicialmente incluyen el método **fail()** que hace fallar la prueba).

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 y ejecutamos:

```

@Test
public void testResta() {
    double valorEsperado = 10;
    Calculadora calcu = new Calculadora(20, 10);
    double resultado = calcu.resta();
}

```

```

        assertEquals(valorEsperado, resultado, 0);
    }

    @Test
    public void testMultiplica() {
        double valorEsperado = 200;
        Calculadora calcu = new Calculadora(20, 10);
        double resultado = calcu.multiplica();
        assertEquals(valorEsperado, resultado, 0);
    }

    @Test
    public void testDivide() {
        double valorEsperado = 2;
        Calculadora calcu = new Calculadora(20, 10);
        double resultado = calcu.divide();
        assertEquals(valorEsperado, resultado, 0);
    }
}

```

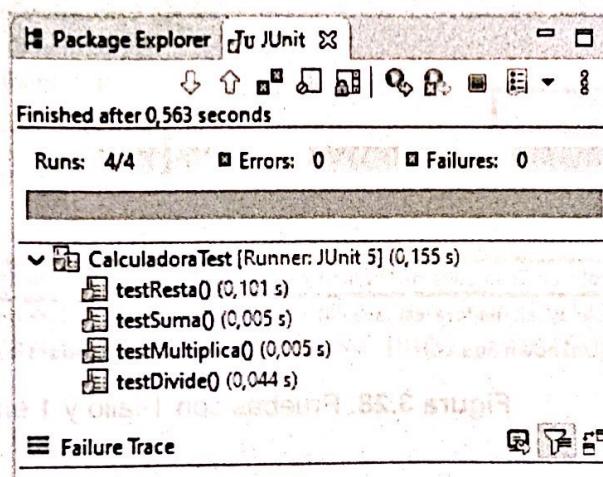


Figura 3.27. JUnit con el resultado satisfactorio de las pruebas.

Ahora al ejecutar la clase de prueba el resultado que muestra es: **Runs: 4/4 Errors: 0 Failures: 0**; nos indica que se han realizado 4 pruebas, ninguna ha provocado error y ninguna ha provocado fallo, véase Figura 3.27.

Para ver la diferencia entre un fallo y un error cambiamos el código de dos de los métodos de prueba. Para hacer que el método ***multiplica()*** produzca un **fallo** hacemos que el valor esperado no coincida con el resultado; se incluye un String en el método ***assertEquals()*** para que si se produce el fallo se muestre el valor indicado en el String. Para que el método ***divide()*** produzca un **error**, al crear el objeto **Calculadora** asignamos el valor 0 al segundo parámetro (será el denominador de la división, al dividir por 0 se produce una excepción). El código de los métodos es el siguiente:

```

    @Test
    public void testMultiplica() {
        double valorEsperado = 200;
        Calculadora calcu = new Calculadora(20, 50);
        double resultado = calcu.multiplica();
        assertEquals(valorEsperado, resultado, 0,
                    "Fallo en la multiplicación: ");
    }

    @Test
    public void testDivide() {
        double valorEsperado = 2;
        Calculadora calcu = new Calculadora(20, 0);
        double resultado = calcu.divide();
        assertEquals(valorEsperado, resultado, 0);
    }
}

```

```

@Test
public void testDivide() {
    double valorEsperado = 2;
    Calculadora calcu = new Calculadora(20, 0);
    double resultado = calcu.divide();
    assertEquals(valorEsperado, resultado, 0);
}

```

Al pulsar en los test que han producido fallos o errores se muestra la traza de ejecución. En la Figura 3.28 se muestra la vista JUnit con el resultado de la ejecución, el botón *Filter Stack Trace* muestra la traza completa de ejecución.

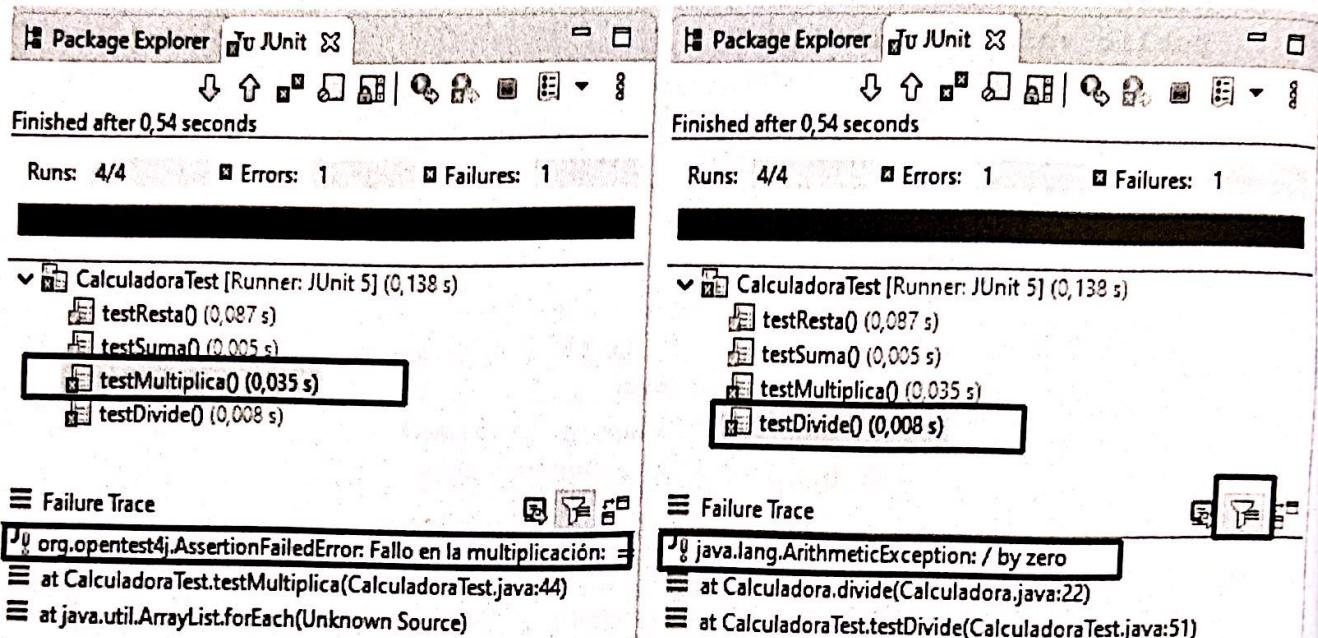


Figura 3.28. Pruebas con 1 fallo y 1 error.

El mensaje mostrado en el panel inferior al seleccionar *testMultiplica()* es el siguiente: *org.opentest4j.AssertionFailedError: Fallo en la multiplicación: ==> expected: <200.0> but was: <1000.0>* que nos informa que ha fallado porque esperaba 200.0 y ha recibido como respuesta 1000.0. Es un fallo porque el caso de prueba está mal definido.

El mensaje mostrado en el panel inferior al seleccionar *testDivide()* es el siguiente: *java.lang.ArithmetricException: / by zero*; que nos informa que se ha producido una excepción al realizar una división por 0, y por tanto se ha producido un error.

El siguiente test comprueba que la llamada al método *divide()* devuelve la excepción *ArithmetricException* al dividir 20 entre 0; por tanto sale por la sentencia *catch*. Si no se lanza la excepción se lanza 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 testExcepcion() {
    try {
        Calculadora calcu = new Calculadora(20, 0);
        double resultado = calcu.divide();
        fail("FALLO, Deberia haber lanzado la excepción");
    } catch (ArithmetricException e) {
        // PRUEBA satisfactoria
    }
}

```