



# Práctica Junit

Entornos de desarrollo

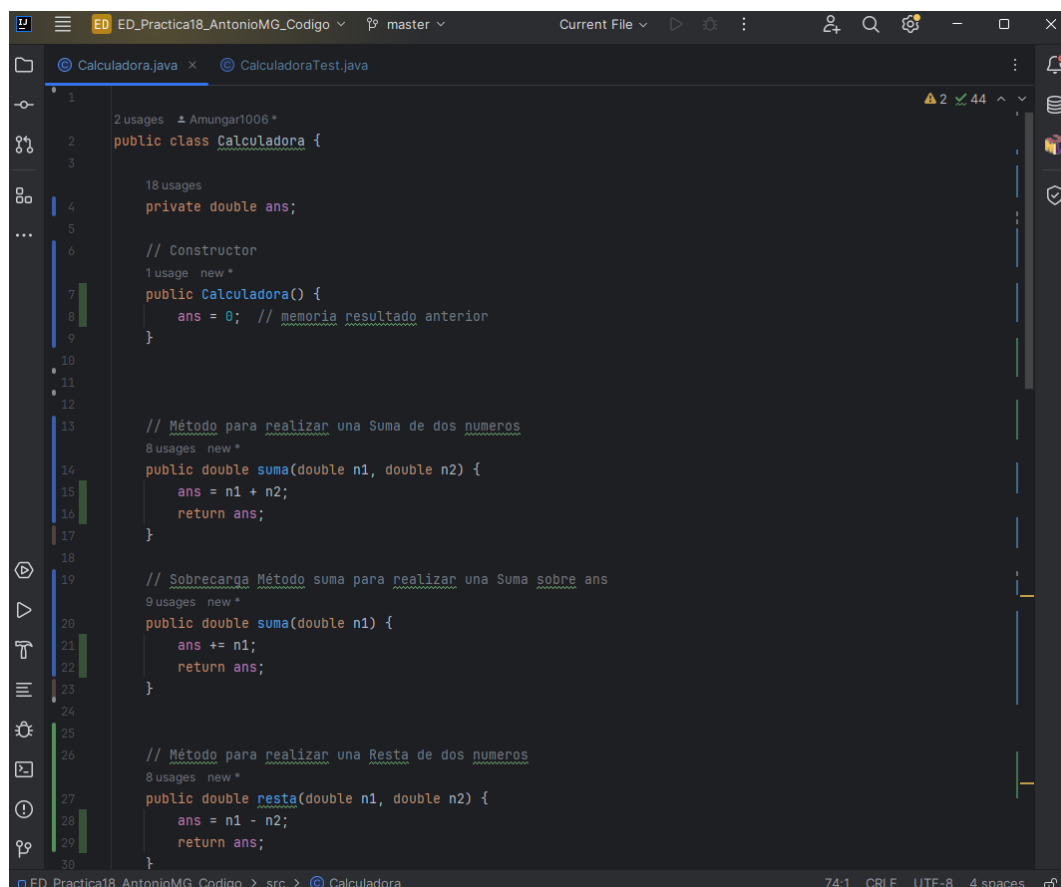
# Ejercicios

Se entregará en GitHub un pdf con todos los pantallazos y la explicación de lo sucedido, aparte del código del proyecto.

## 1. Clase calculadora (2 ptos)

Se solicita programar la clase Calculadora vista en teoría de forma que, además de incluir el método suma (visto en transparencias), incluya un método para restar 2 números, otro para multiplicar 2 números y otro para dividir entre 2 números enteros.

Se crea la clase calculadora, con un atributo llamado ans, que guardará el último valor de la calculadora, para poder operar directamente con él. Se implementan los métodos suma, resta, divide y multiplica. Se sobrecargan todos para que, en vez de recibir dos parámetros, reciban uno, y operen directamente con el valor guardado en ans. También se crea un método para el borrado de ans (ans = 0).

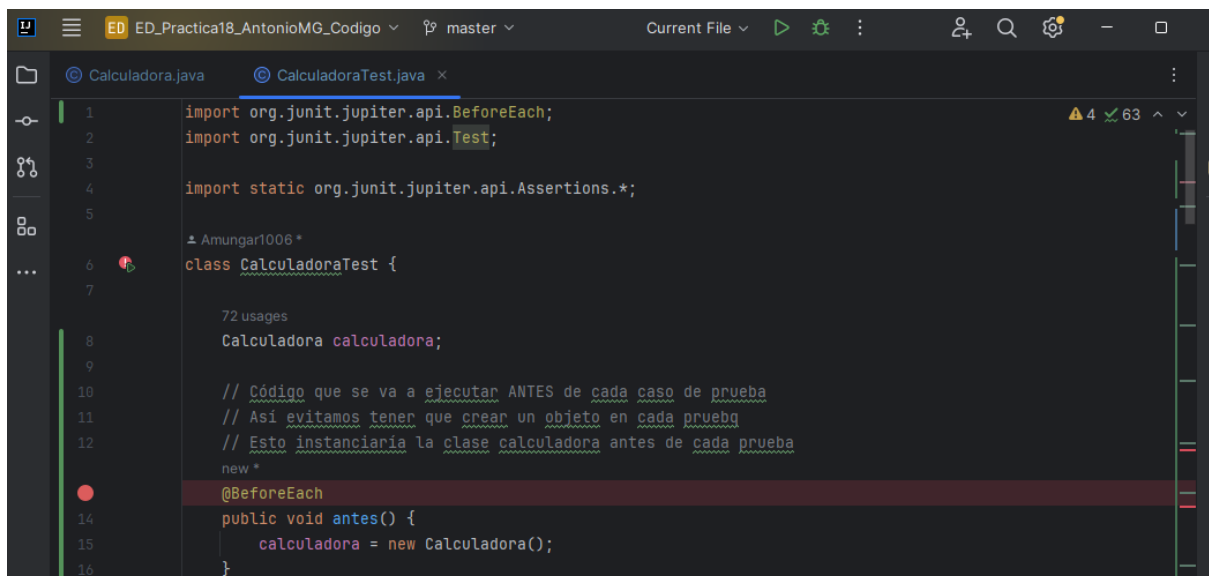


```
1 2 usages Amungar1006 *
2 public class Calculadora {
3
4     18 usages
5     private double ans;
6
7     // Constructor
8     1 usage new *
9     public Calculadora() {
10         ans = 0; // memoria resultado anterior
11     }
12
13     // Método para realizar una Suma de dos numeros
14     8 usages new *
15     public double suma(double n1, double n2) {
16         ans = n1 + n2;
17         return ans;
18     }
19
20     // Sobrecarga Método suma para realizar una Suma sobre ans
21     9 usages new *
22     public double suma(double n1) {
23         ans += n1;
24         return ans;
25     }
26
27     // Método para realizar una Resta de dos numeros
28     8 usages new *
29     public double resta(double n1, double n2) {
30         ans = n1 - n2;
31         return ans;
32     }
33 }
```

## 2. Clase test (5 ptos)

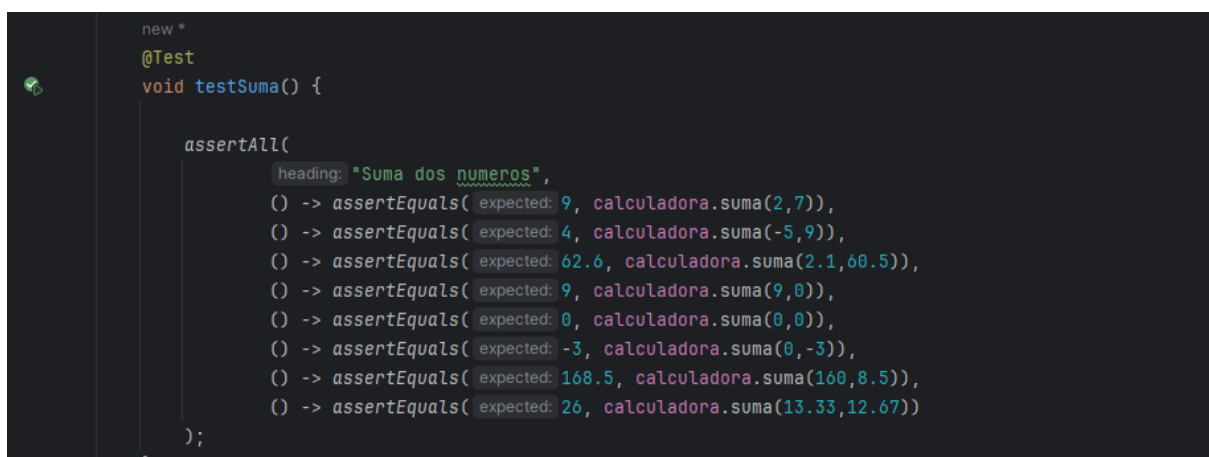
Deberás realizar los test con JUnit a todos los métodos indicados anteriormente.

Lo primero que hago es crear un método llamado `antes()` que se ejecutará antes de cada prueba gracias a indicar la notación `@BeforeEach`, que instanciará un objeto `Calculadora`, ya que toda la clase está pensada para operar a través de un objeto `Calculadora` con su memoria (llamada `ans`).



```
1 import org.junit.jupiter.api.BeforeEach;
2 import org.junit.jupiter.api.Test;
3
4 import static org.junit.jupiter.api.Assertions.*;
5
6 class CalculadoraTest {
7
8     // Código que se va a ejecutar ANTES de cada caso de prueba
9     // Así evitamos tener que crear un objeto en cada prueba
10    // Esto instanciaría la clase calculadora antes de cada prueba
11    new *
12    @BeforeEach
13    public void antes() {
14        calculadora = new Calculadora();
15    }
16 }
```

Después se crea un test para cada método, y también un test para la sobrecarga de cada método indicado como “Test nombreMétodo Ans”. Lo hacemos con un aserto compuesto (`assertAll`), que espera que todos sus sub-assertos pasen el test.



```
new *
@Test
void testSuma() {
    assertAll(
        heading: "Suma dos numeros",
        () -> assertEquals( expected: 9, calculadora.suma(2,7)),
        () -> assertEquals( expected: 4, calculadora.suma(-5,9)),
        () -> assertEquals( expected: 62.6, calculadora.suma(2.1,60.5)),
        () -> assertEquals( expected: 9, calculadora.suma(9,0)),
        () -> assertEquals( expected: 0, calculadora.suma(0,0)),
        () -> assertEquals( expected: -3, calculadora.suma(0,-3)),
        () -> assertEquals( expected: 168.5, calculadora.suma(160,8.5)),
        () -> assertEquals( expected: 26, calculadora.suma(13.33,12.67))
    );
}
```

```
50
51
52 new *
53 @Test
54 void testResta() {
55     assertAll(
56         heading: "Resta de dos numeros",
57         () -> assertEquals( expected: 2, calculadora.resta(6, 4)),
58         () -> assertEquals( expected: 9, calculadora.resta(13,4)),
59         () -> assertEquals( expected: 6, calculadora.resta(10.5, 4.5)),
60         () -> assertEquals( expected: 21.4, calculadora.resta(14, -7.4)),
61         () -> assertEquals( expected: 0, calculadora.resta(26.33, 26.33)),
62         () -> assertEquals( expected: -16, calculadora.resta(-7, 9)),
63         () -> assertEquals( expected: 160, calculadora.resta(-10, -170)),
64         () -> assertEquals( expected: 32.89, calculadora.resta(78.56, 45.67))
65     );
66 }
67
68 new *
69 @Test
70 void testRestaAns() {
71     assertAll(
72         heading: "Resta de memoria (ans) - numero",
73         () -> assertEquals( expected: -4, calculadora.resta( n1: 4)),
74         () -> assertEquals( expected: -9, calculadora.resta( n1: 5)),
75         () -> assertEquals( expected: 4.5, calculadora.resta( n1: -13.5)),
76         () -> assertEquals( expected: 0, calculadora.resta( n1: 4.5)),
77         () -> assertEquals( expected: -5, calculadora.resta( n1: 5)),
78         () -> assertEquals( expected: -8.33, calculadora.resta( n1: 3.33)),
79         () -> assertEquals( expected: -7, calculadora.resta( n1: -1.33)),
80         () -> assertEquals( expected: 33, calculadora.resta( n1: -40))
81     );
82 }
```

```
83
84 @Test
85 void testMultiplica() {
86     assertAll(
87         heading: "Multiplicación de dos numeros",
88         () -> assertEquals( expected: 12, calculadora.multiplica(4, 3)),
89         () -> assertEquals( expected: -38.08, calculadora.multiplica(5.6, -6.8)),
90         () -> assertEquals( expected: 8, calculadora.multiplica(8, 1)),
91         () -> assertEquals( expected: 0, calculadora.multiplica(-5, 0)),
92         () -> assertEquals( expected: 22.5, calculadora.multiplica(5, 4.5)),
93         () -> assertEquals( expected: 19.98, calculadora.multiplica(3.33, 6))
94     );
95 }
96
97 new *
98 @Test
99 void testMultiplicaAns() {
100     // sumamos uno a ans, sino el test siempre daria cero
101     calculadora.suma( n1: 1);
102     assertAll(
103         heading: "Multiplicación de memoria (ans) por numero",
104         () -> assertEquals( expected: 4, calculadora.multiplica( n1: 4)),
105         () -> assertEquals( expected: 12, calculadora.multiplica( n1: 3)),
106         () -> assertEquals( expected: -36, calculadora.multiplica( n1: -3)),
107         () -> assertEquals( expected: 18, calculadora.multiplica( n1: -0.5)),
108         () -> assertEquals( expected: 4.5, calculadora.multiplica( n1: 0.25)),
109         () -> assertEquals( expected: 0, calculadora.multiplica( n1: 0)),
110         () -> assertEquals( expected: 0, calculadora.multiplica( n1: 8)),
111         () -> assertEquals( expected: 0, calculadora.multiplica( n1: -2.5))
112     );
113 }
```

```
ED ED_Practica18_AntonioMQ_Codigo master Current File
Calculadora.java CalculadoraTest.java

117 void testDivide() {
118     assertAll(
119         heading: "Division de dos numeros",
120         () -> assertEquals( expected: 2, calculadora.divide(4,2)),
121         () -> assertEquals( expected: 12, calculadora.divide(6, 0.5)),
122         () -> assertEquals( expected: -32, calculadora.divide(8, -0.25)),
123         () -> assertEquals( expected: 6.72, calculadora.divide(26.88, 4)),
124         () -> assertEquals( expected: 2.5, calculadora.divide(5, 2)),
125         () -> assertEquals( expected: 0, calculadora.divide(0, 5))
126     );
127 }
128
129 new *
130 @Test
131 void testDivideAns() {
132     // sumamos uno a ans, sino el test siempre daría cero
133     calculadora.suma( n1: 1);
134     assertAll(
135         heading: "Division de memoria (ans) entre numero",
136         () -> assertEquals( expected: -4, calculadora.divide( n1: -0.25)),
137         () -> assertEquals( expected: 2, calculadora.divide( n1: -2)),
138         () -> assertEquals( expected: 20, calculadora.divide( n1: 0.1)),
139         () -> assertEquals( expected: 2.5, calculadora.divide( n1: 8)),
140         () -> assertEquals( expected: -1.25, calculadora.divide( n1: -2)),
141         () -> assertEquals( expected: 5, calculadora.divide( n1: -0.25)),
142         // Deberia capturar el error y mostrar la excepcion y por tanto no modificar ans
143         () -> assertEquals( expected: 0, calculadora.divide( n1: 0)),
144         // Entonces la division deberia ser el ans anterior 5, entre 8
145         // Gracias al hacer assertAll, capturará el error de la division entre cero y se mostrará, pero también se m
146         () -> assertEquals( expected: 0.1625, calculadora.divide( n1: 8))
147     );
148 }
```

También realizo un test para comprobar que el borrado de ans se realiza correctamente y así cubrir toda la cobertura del código en las pruebas.

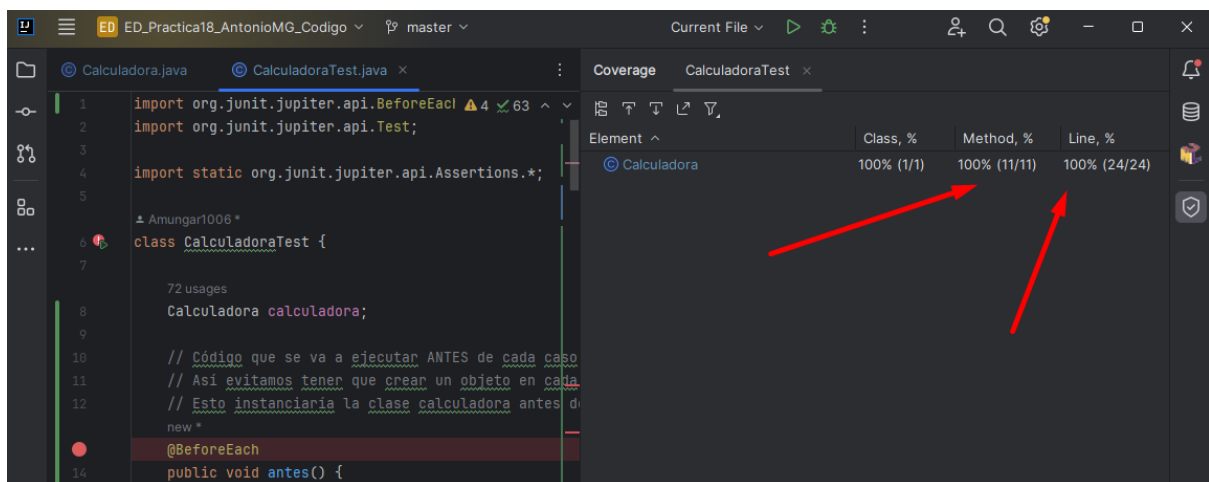
Además se implementa prueba para comprobar que la división entre cero lanza una excepción.

```
149
150 new *
151 @Test
152 void testDivideEntreCero() {
153     assertThrows(ArithmeticException.class, () -> {
154         calculadora.divide(8, 0);
155     });
156 }
157
158 new *
159 @Test
160 void testBorrar() {
161     calculadora.suma( n1: 2);
162     assertEquals( expected: 2, calculadora.getAns());
163     calculadora.borrar();
164     assertEquals( expected: 0, calculadora.getAns());
165
166     calculadora.resta( n1: 8);
167     assertEquals( expected: -8, calculadora.getAns());
168     calculadora.borrar();
169     assertEquals( expected: 0, calculadora.getAns());
170 }
```

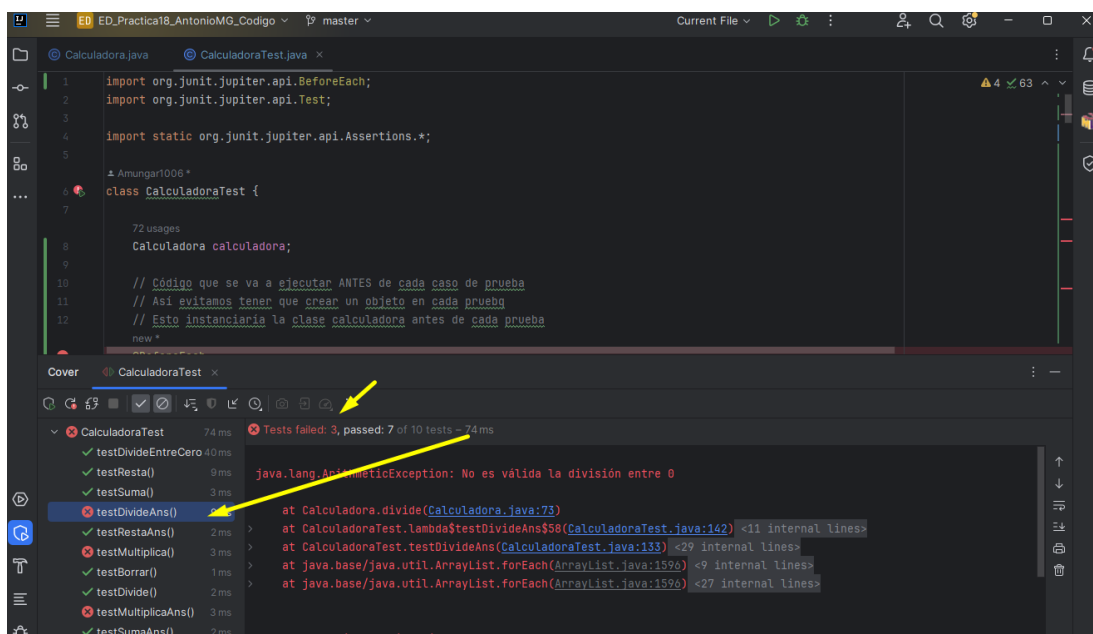
### 3. Pruebas (3 ptos)

Comprueba que los test pasan satisfactoriamente con los resultados esperados y fuerza en todos los casos con valores que den como resultado test fallido. Prueba también con dividir entre 0. (Mínimo 5 pruebas con cada clase)

Lo primero que vamos a hacer es ejecutarlo con cobertura, y así comprobar que no nos hemos dejado ninguna parte del código sin testear. Como vemos los test cubren todo el código.



Luego ejecutamos los test y podemos ver que hay algunos que fallan.

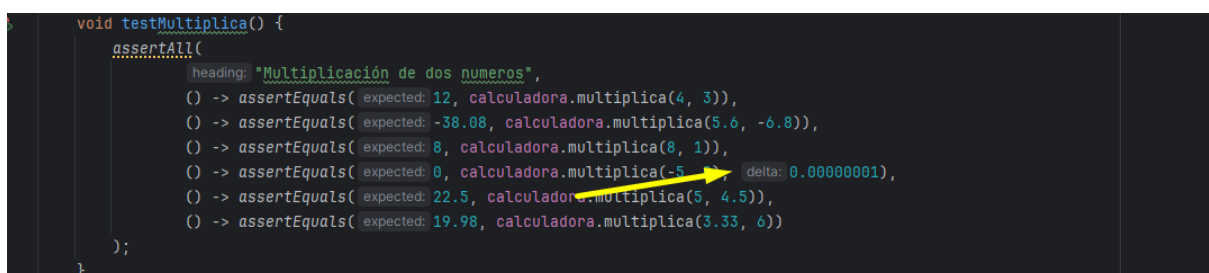


Vamos a intentar solucionar los errores que nos arrojan los test. Como vemos el error que nos muestra es que en el test de la línea 90 espera un 0.0 y sin embargo recibe un -0.0. Gracias a este test se ha podido encontrar este error.

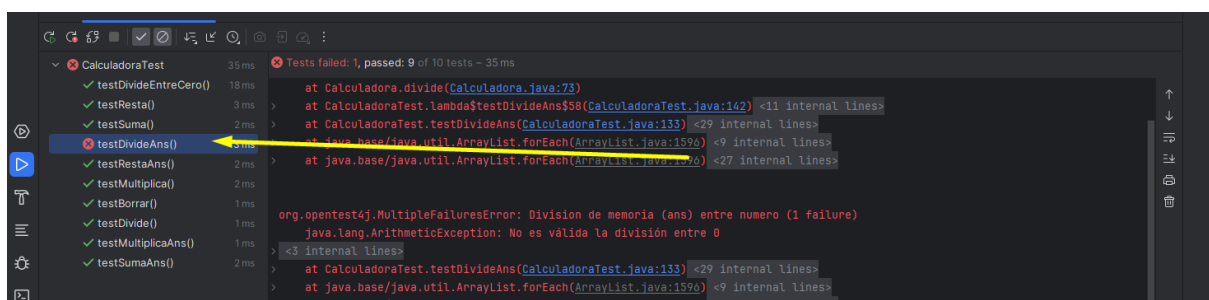


El error es debido a la forma en que se manejan los números en punto flotante en Java. Cuando se comparan ceros, con resultado negativo o positivo, aunque matemáticamente deberían ser lo mismo, java a veces a compararlo, da un resultado falso. Gracias a esta prueba podremos corregir este error y mejorar nuestro código.

Luego de un rato de investigar, verificamos que  $0.0 == -0.0$  arroja True, por lo que el problema está en como Equals maneja estos casos. La solución encontrada es añadir una tolerancia pequeña a los valores que esperen cero.



Se han solucionado casi todos los errores pero vemos que aún sigue quedando alguno:



El error que nos muestra, es que esperamos un cero, y no una excepción. Para corregir esto podemos cambiar esto:

```
public double divide(double n1) {  
    if (n1 != 0) {  
        ans /= n1;  
        return ans;  
    } else {  
        throw new ArithmeticException("No es válida la división entre 0");  
    }  
}
```

```
() -> assertEquals( expected: -1.25, calculadora.divide( n1: -2)),  
() -> assertEquals( expected: 5, calculadora.divide( n1: -0.25)),  
// Debería capturar el error y mostrar la excepción y por tanto no modificar ans  
() -> assertThrows(ArithmeticException.class, () -> {calculadora.divide(8, 0);}),  
// Entonces la división debería ser el ans anterior 5, entre 8  
// Gracias al hacer assertAll, capturará el error de la división entre cero y se mostrará, pero también se muest  
() -> assertEquals( expected: 0.625, calculadora.divide( n1: 8))  
};
```

Ya tendríamos todos los test correcto y comprobado el funcionamiento:

The screenshot shows an IDE with two files: `Calculadora.java` and `CalculadoraTest.java`. The `CalculadoraTest.java` file contains several JUnit tests. The `Run` tab at the bottom shows the execution results of the tests. The tests are all passed, and the process finished with exit code 0.

Test Results:

Test Name	Duration	Status
testDivideEntreCero()	17 ms	Passed
testResta()	1 ms	Passed
testSuma()	2 ms	Passed
testDivideAns()	3 ms	Passed
testRestaAns()	3 ms	Passed
testMultiplica()	1 ms	Passed
testBorrar()	1 ms	Passed
testDivide()	1 ms	Passed
testMultiplicaAns()	1 ms	Passed
testSumaAns()	1 ms	Passed

Summary: Tests passed: 10 of 10 tests - 32 ms. Process finished with exit code 0.



Además realizamos varios test que deberían mostrar error, ya que no obtienen el resultado obtenido. Es importante también probar los casos erróneos y verificar que darían error.

```
new *
@Test
void testSumaERROR() {
    assertEquals( expected: 2, calculadora.suma(2, 0.5));
}

new *
@Test
void testRestaERROR() {
    assertEquals( expected: 1, calculadora.resta(2, -3));
}

new *
@Test
void testDivideERROR() {
    assertEquals( expected: 1.5, calculadora.multiplifica(2, 0.5));
}

new *
@Test
void testMultiplicaERROR() {
    assertEquals( expected: 2, calculadora.divide(2, 0.5));
}
```

