

Tabla de Contenidos



- [JUnit y pruebas unitarias](#)
- [JUnit @Before vs @BeforeClass](#)
- [Otros artículos relacionados](#)

Vamos a ver la diferencia entre @Before vs @BeforeClass dos de las anotaciones más utilizadas en JUnit a la hora de organizar nuestro código y sus inicializaciones. Para ello vamos a partir de dos pruebas unitarias sencillas usando la clase Factura que contiene una lista de líneas que son valores de tipos doble.

```
package com.arquitecturajava.test;

import java.util.ArrayList;
import java.util.List;

public class Factura {

    private List<Double> lineas= new ArrayList<Double>();
    public List<Double> getLineas() {
        return lineas;
    }

    public void setLineas(List<Double> lineas) {
        this.lineas = lineas;
    }

    public void addLinea(double linea) {
        lineas.add(linea);
    }

    public double sumarLineas () {
```

```

        double total=0;
        for (double linea: lineas) {
            total+=linea;
        }
        return total;
    }
    public double mayor () {

        double mayor=0;
        for (double linea: lineas) {
            if (mayor<linea) {
                mayor=linea;
            }
        }
        return mayor;
    }
}

```

JUnit y pruebas unitarias

Una vez construida la clase podemos definir un par de pruebas unitarias sencillas que comprueben los métodos `sumarLineas()` y `mayor()`. Algo similar a lo siguiente podría valer:

```

package com.arquitecturajava.test;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class FacturaTest {

    @Test

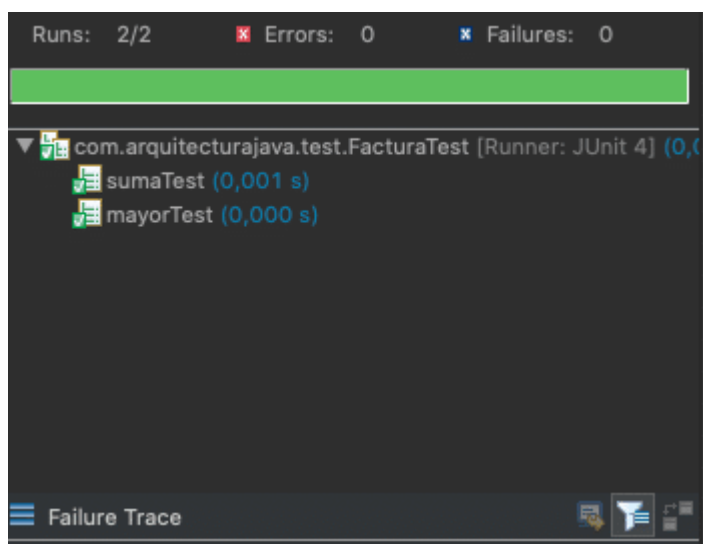
```

```

    public void sumaTest() {
        Factura f= new Factura();
        f.addLinea(100);
        f.addLinea(200);
        assertEquals(300,f.sumarLineas(),0);
    }
    @Test
    public void mayorTest() {
        Factura f= new Factura();
        f.addLinea(100);
        f.addLinea(200);
        assertEquals(200,f.mayor(),0);
    }
}

```

Acabamos de pasar dos pruebas unitarias con JUnit :



¿Son estas pruebas unitarias correctas? . La realidad es que no tienen problemas pero sí que es cierto que comparten mucho código. Es decir tenemos mucho código repetido. Podemos apoyarnos en @Before como anotación de JUnit para crear un método de

inicialización que se encargue de iniciar la Factura con sus lineas .

```
package com.arquitecturajava.test;

import static org.junit.Assert.assertEquals;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class FacturaTest2 {

    Factura f;
    @Before
    public void inicializar() {

        f = new Factura();
        f.addLinea(100);
        f.addLinea(200);
    }

    @Test
    public void sumaTest() {
        assertEquals(300, f.sumarLineas(), 0);
    }

    @Test
    public void restaTest() {
        assertEquals(200, f.mayor(), 0);
    }

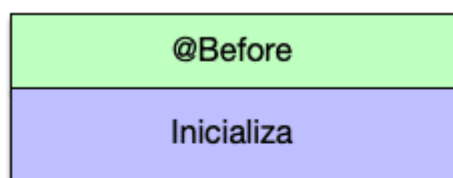
    @After
    public void finaliza() {
```

```

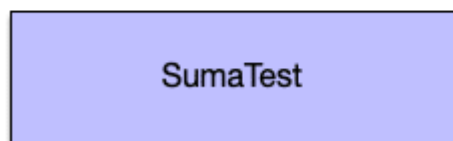
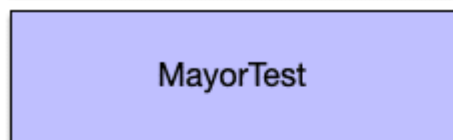
        f=null;
    }
}

```

En este caso hemos usado no solo @Before sino también @After . Estas anotaciones ejecutarán su funcionalidad cada vez que comience y termine cada uno de los Test . De esta forma nos encontramos con un código mas simplificado y reutilizable.



Se ejecuta al iniciar cualquier test
De la clase



Se ejecuta al finalizar cualquier test
De la clase

Junit @Before vs @BeforeClass

Sin embargo existen situaciones en las cuales a nosotros nos puede interesar iniciar una clase o una variable una sola vez para todos los test ya que todos la comparten y solo necesitan instanciarla la primera vez. Imaginemonos que tenemos un método adicional que calcula la suma de facturas con el IVA.

```

public double sumarLineasConIVA(double porcentaje) {

```

```
        return sumarLineas()+sumarLineas()/100*porcentaje;
    }
}
```

Se trata de un método muy sencillo pero igual el IVA a aplicar a las facturas se tiene que calcular primero con un servicio aparte.

```
package com.arquitecturajava.test;
```

```
public class ServicioFacturas {

    public static double obtenerIVA() {
        return 21;
    }
}
```

Podemos usar la anotación `@BeforeClass` para invocar a este servicio y poner a nuestra disposición el porcentaje de IVA.

```
public class FacturaTest3 {

    Factura f;
    double IVA;
    @BeforeClass
    public static void inicializadorComun() {
        IVA=ServicioFacturas.obtenerIVA();
    }

    .....
}
```

Una vez construido este método podemos crear un nuevo test que nos valide el método `sumarLineasConIVA`.

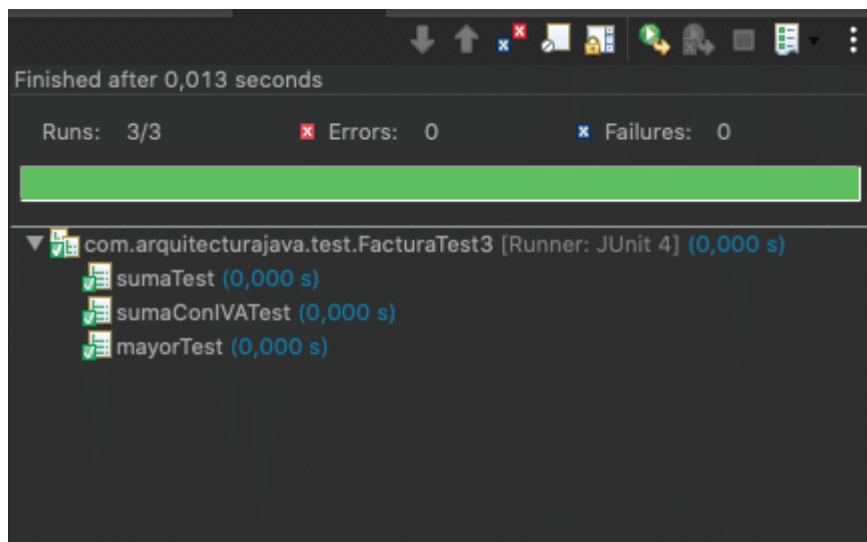
```
@Test
public void sumaConIVATest() {
```

```

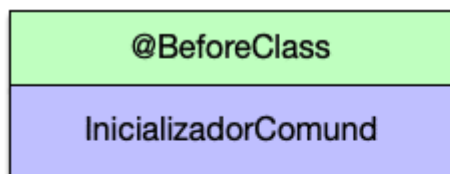
        assertEquals(363, f.sumarLineasConIVA(IVA), 0);
    }

```

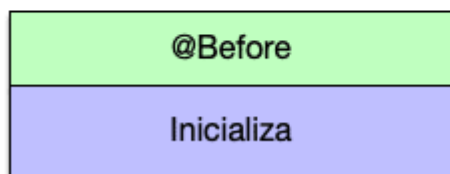
Todos los test se ejecutan correctamente y el IVA es calculado:



Acabamos de ver la diferencia entre @Before vs @BeforeClass en JUnit .



Se ejecuta una sola vez por clase



Se ejecuta al iniciar cualquier test
De la clase

Cada día es más importante construir pruebas unitarias para nuestro código ya que aportará seguridad y flexibilidad a lo que construimos.

Otros artículos relacionados

- [Spring Testing y el manejo de JUnit](#)

- [Curso TDD Java , Diseño y Patrones](#)
- [Mockito Stub , test y aislamiento](#)