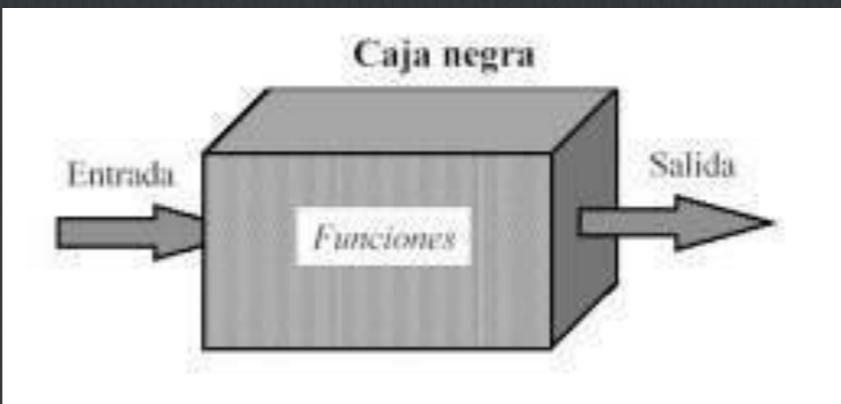


# **PRUEBAS UNITARIAS**

**UT 3 : DISEÑO Y REALIZACIÓN DE PRUEBAS**

# DEFINICIÓN (I)

---



**En contraposición a las pruebas de caja blanca, las Pruebas de Caja Negra, constituyen una técnica de pruebas de software en para comprobar y verificar la funcionalidad de una aplicación sin tener en cuenta la implementación o estructura interna de código, así como los escenarios de ejecución (donde se va a ejecutar).**

# DEFINICIÓN (II)

---

**En estas pruebas, no hace falta conocer la estructura interna del programa ni su funcionamiento. Su busca la obtención de casos de prueba que demuestren que las salidas que devuelve la aplicación son las esperadas en función de las entradas que se proporcionen.**

**A este tipo de pruebas también se les llama prueba de comportamiento. Con ellas intentamos encontrar errores de las siguientes categorías:**

- Funcionalidades incorrectas o ausentes.**
- Errores de interfaz.**
- Errores en estructuras de datos o en accesos a bases de datos externas.**
- Errores de rendimiento.**
- Errores de inicialización y finalización.**

# TEST UNITARIOS

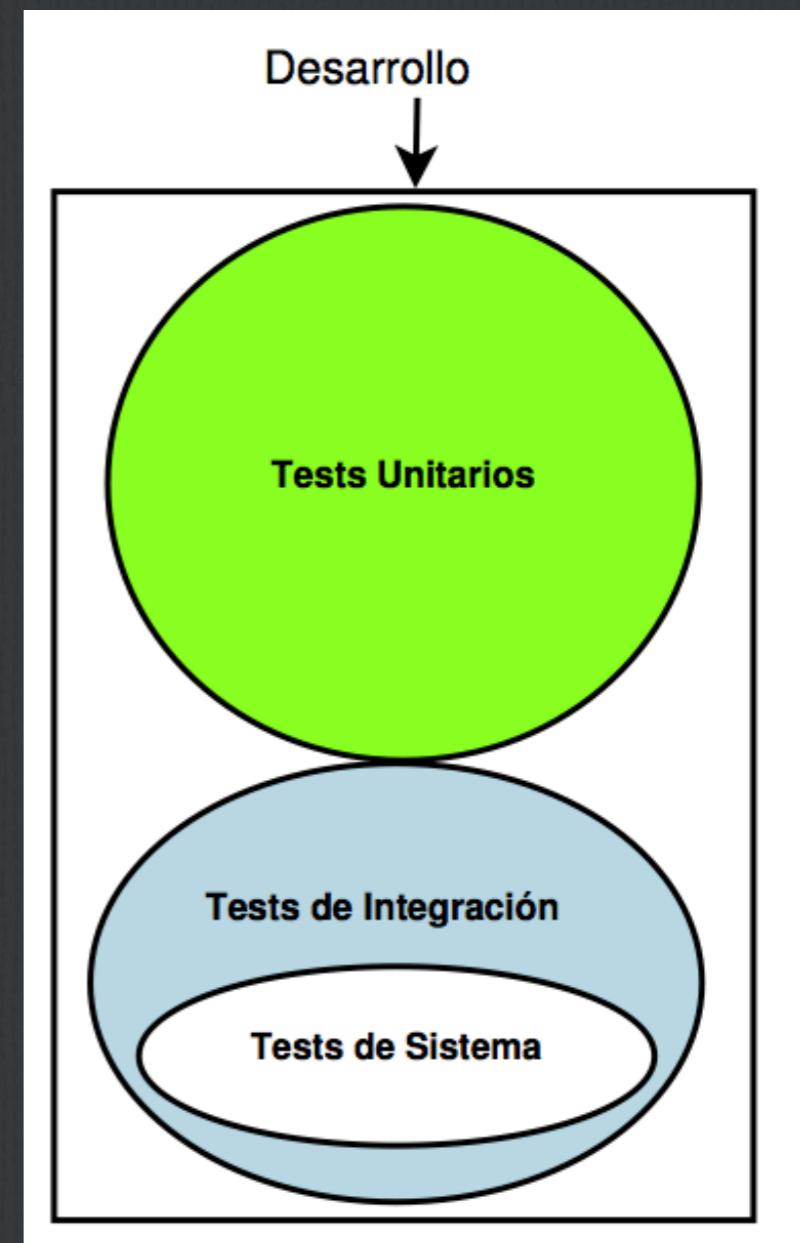
---

Son una forma de manera el correcto funcionamiento de una unidad de código.

Nos ayuda a asegurarnos que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, se puede comprobar que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve.

Para ello, se debe escribir casos de prueba para cada función o método no trivial (que tenga cierta complejidad) en el módulo.

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.



# QUÉ VAMOS A VER

---

En nuestro caso vamos a ver dos maneras de realizar test o pruebas:

- Prueba de clase: Se genera una clase de prueba para toda la clase de manera global. Este tipo de pruebas es recomendable en el caso de clases sencillas en los que suele devolver un tipo de dato concreto (integer, char, storing ,...).
- Prueba unitaria parametrizada: En este caso, probamos un método con diferentes conjuntos de parámetros de entrada para comprobar que los procesa de manera correcta y devuelve en cada caso el valor y el tipo correspondiente.

Son dos maneras de abordar las pruebas, que podemos realizar de manera conjunta o independiente según las necesidades de nuestra aplicación.

# CONCEPTOS PREVIOS DE JAVA: ANOTACIONES

---

Antes de empezar, debemos conocer dos conceptos que vamos a utilizar durante el desarrollo de nuestras pruebas: las anotaciones y los módulos.

Las anotaciones son, según la wikipedia, una forma de añadir metadatos al código fuente Java, y se pueden añadir a los elementos de programa tales como clases, métodos, metadatos, campos, parámetros, variables locales, y paquetes.

Para una información más detallada, podéis leer este útil enlace:

<https://jarroba.com/annotations-anotaciones-en-java/>

# CONCEPTOS PREVIOS DE JAVA: MÓDULOS

---

Por otro lado, los módulos de java se basan en la programación modular, y tratan de ser una aproximación a este paradigma de programación. El concepto fue introducido en JAVA 9 y básicamente es una manera de agrupar clases comunes para ofrecer una agrupación más flexible y versátil que la que se ofrecía desde un enfoque diferente los packages, que son mas monolíticos.

Para entender todo esto os recomiendo visitar los siguientes enlaces:

<https://www.arquitecturajava.com/java-9-modules/>

<https://es.stackoverflow.com/questions/282078/qu%C3%A9-es-el-module-info-de-java-y-para-qu%C3%A9-se-utiliza>

# HERRAMIENTA



# JUnit 5

---

**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.**

**Hay problemas con su integración en Netbeans, pero hay soluciones no oficiales por internet.**

**La versión actual es la 5 (JUPITER) y tiene algunos cambios respecto a las versiones anteriores, por lo que es recomendable que todo el mundo use esta versión, si no, habrá problemas a la hora de corregir los ejercicios y/o exámenes.**

**En el siguiente enlace podéis encontrar más información sobre las novedades de JUNIT 5:**

**<https://www.adictosaltrabajo.com/2016/11/24/primeros-pasos-con-junit-5/>**

# EJEMPLO I : PRUEBAS DE CLASE (I)

---

```
package pruebas;

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;
    }
}
```

**Para comenzar nuestro ejemplo, en primer lugar debemos crear nuestra clase sobre la que vamos a realizar los test.**

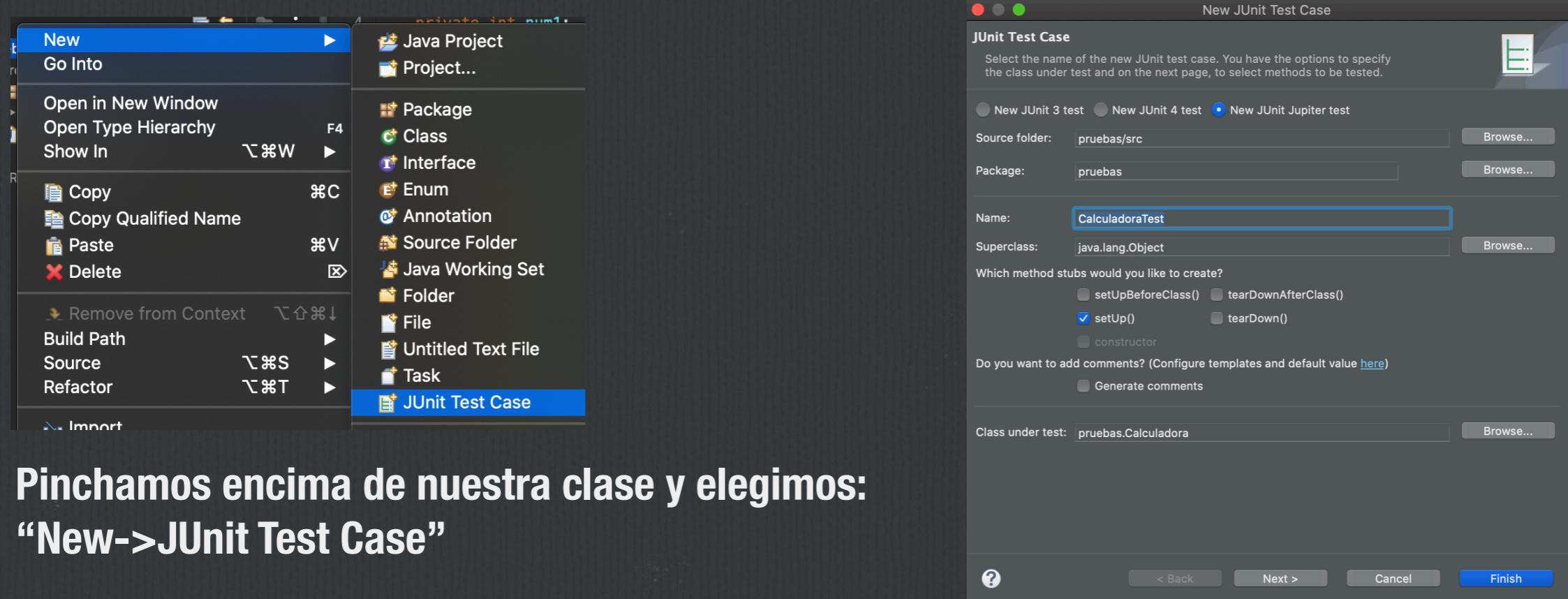
**Para este ejemplo básico, hemos creado esta clase calculadora que incluye cuatro métodos y el constructor, que recibe dos números como entrada.**

**Recordemos que para crear un objeto de esta clase debemos hacerlo usando el constructor:**

```
Calculadora micalc= new Calculadora(10, 5);
```

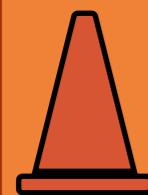
**Ahora vamos a crear una clase de prueba para verificar nuestra calculadora.**

# EJEMPLO I : PRUEBAS DE CLASE (II)



Pinchamos encima de nuestra clase y elegimos:  
“New->JUnit Test Case”

y nos aparecerá la ventana donde automáticamente se pone el nombre.  
Recomendamos marcar el método setUp() para que lo genere. Pinchamos en “Next”.



Si pinchamos encima de otro sitio que no sea el archivo de clase, no nos dejará pinchar en “Next”.

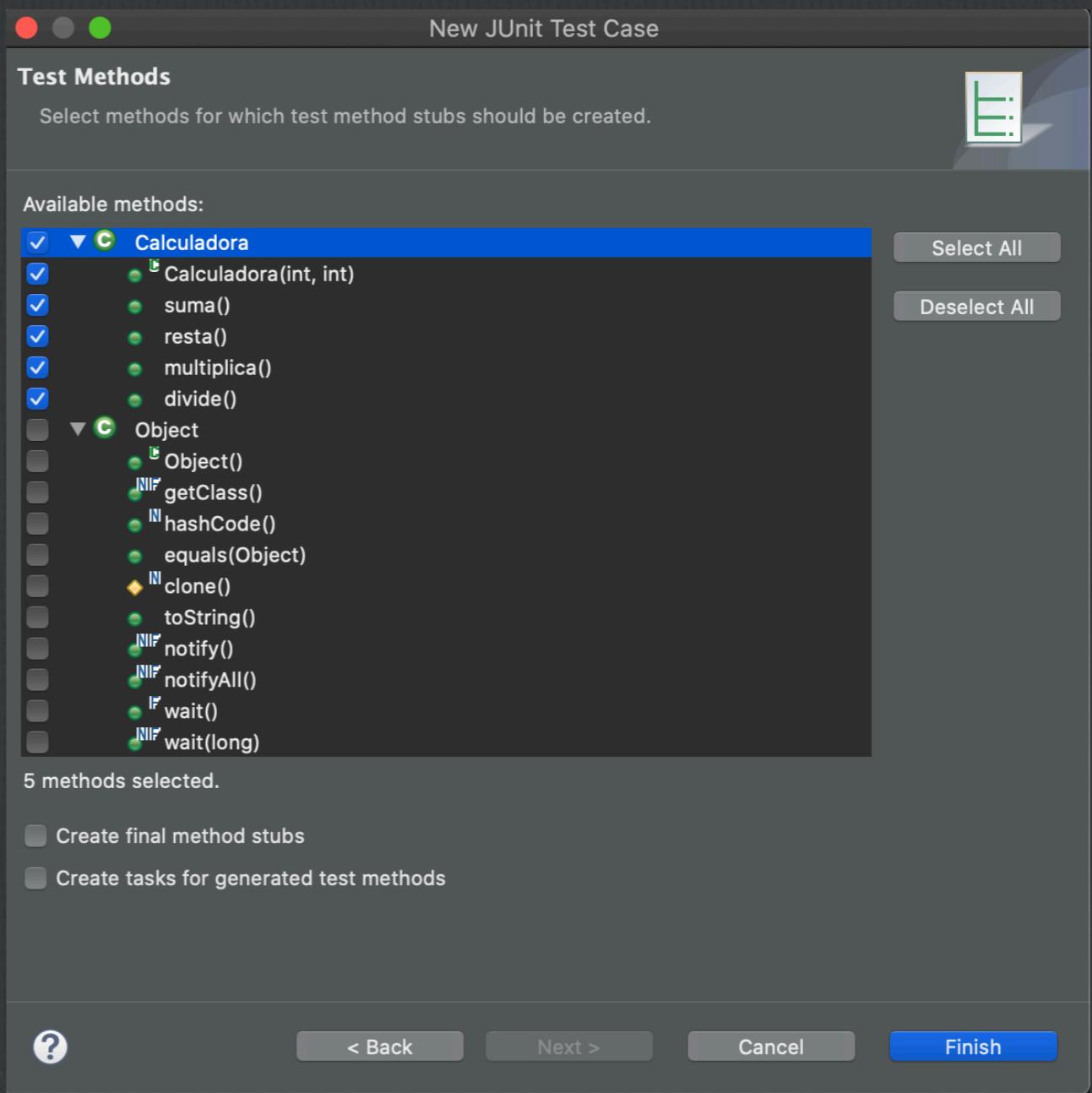
# EJEMPLO I : PRUEBAS DE CLASE (III)

En esta pestaña debemos elegir sobre qué métodos vamos a crear el test.

Marcamos todos los de nuestra clase.

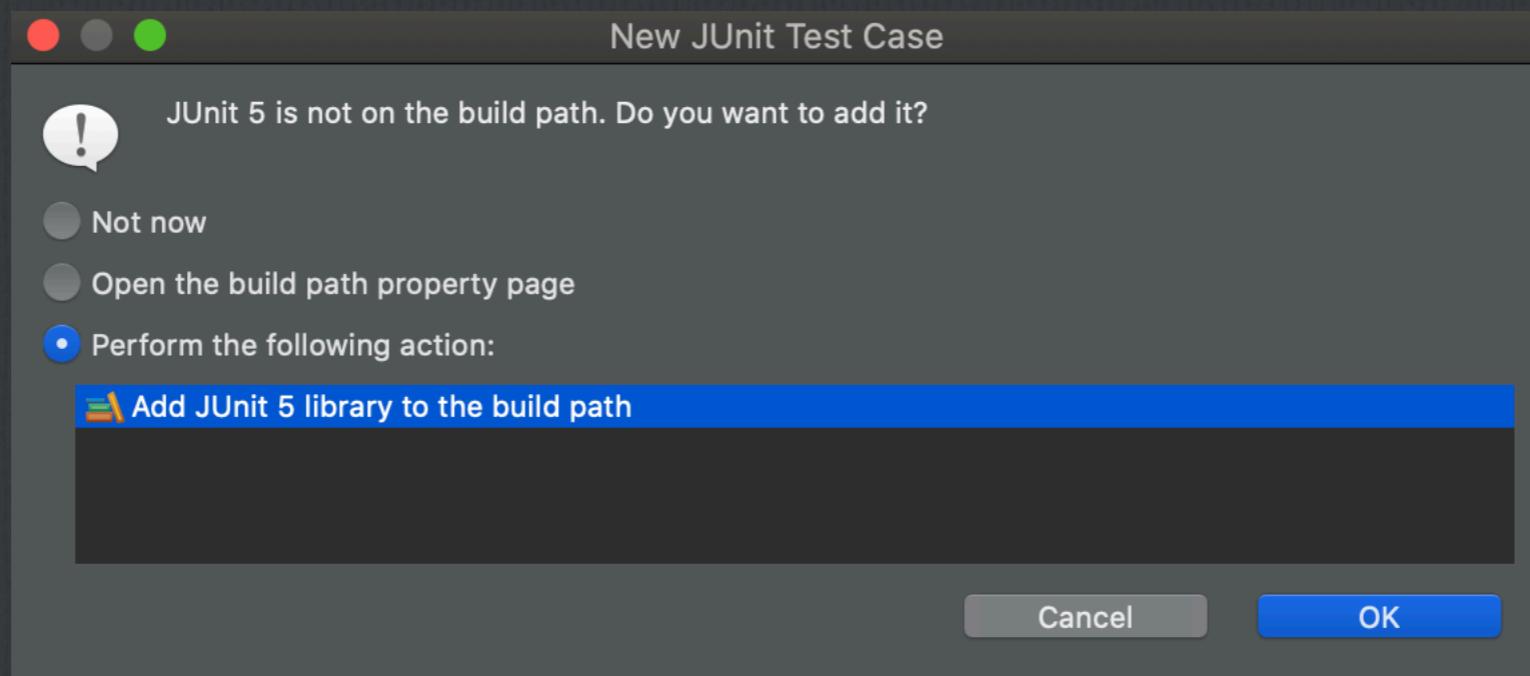
El resto no los vamos a ver en este curso.

Pinchamos en “Finish”.



# EJEMPLO I : PRUEBAS DE CLASE (IV)

El sistema nos pregunta si queremos añadir JUnit 5 a nuestro “build path”, que es el lugar donde importamos nuestras librerías.

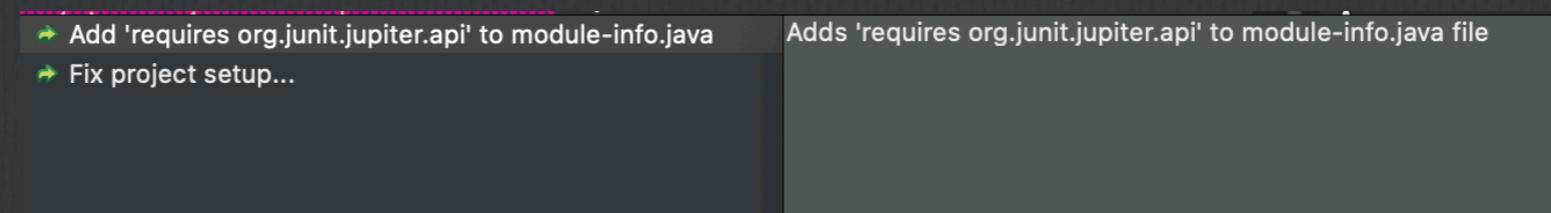


Pinchamos en “OK” y el sistema nos genera un archivo dentro de nuestro package.

# EJEMPLO I : PRUEBAS DE CLASE (V)

La clase generada nos da un error al intentar ejecutarla inicialmente:  
“The type org.junit.jupiter.api.Assertions is not accessible”

Si tratamos de solucionarlo con la primera de las opciones propuestas:

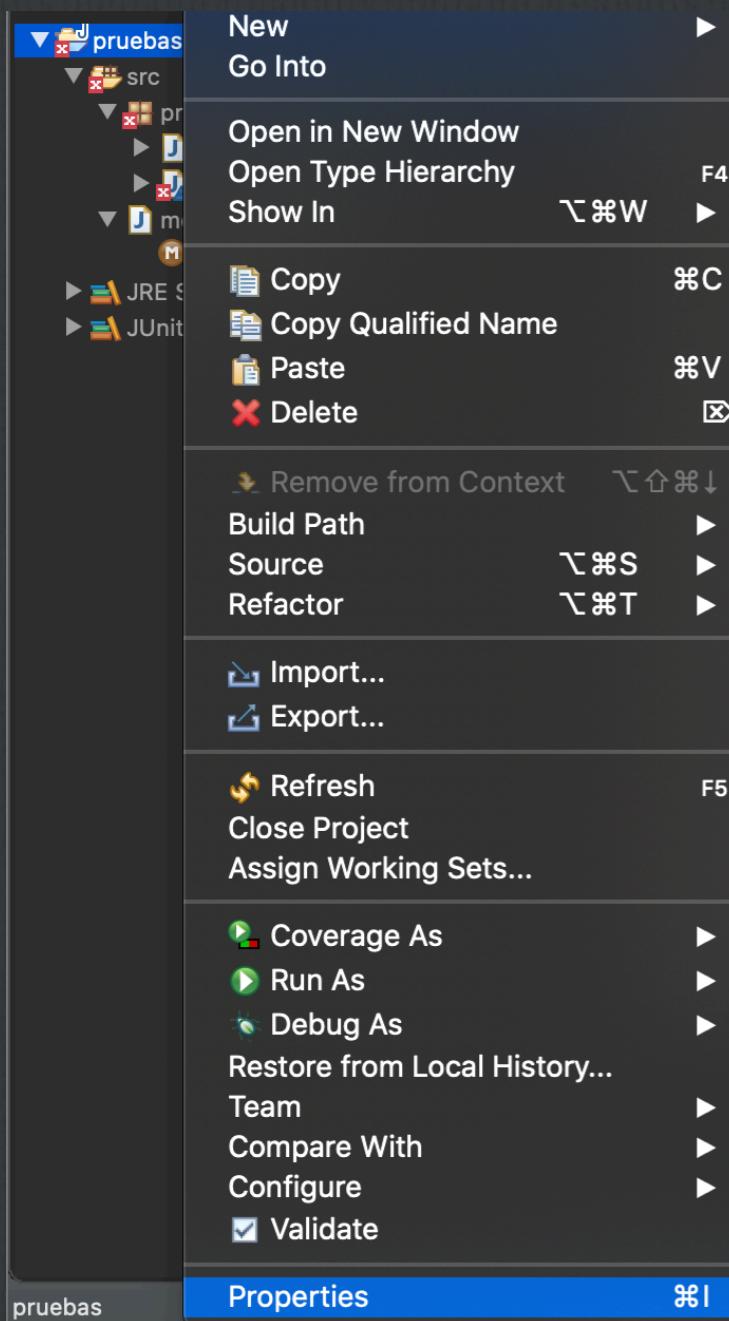


Nos generará una linea en el archivo “module-info.java”

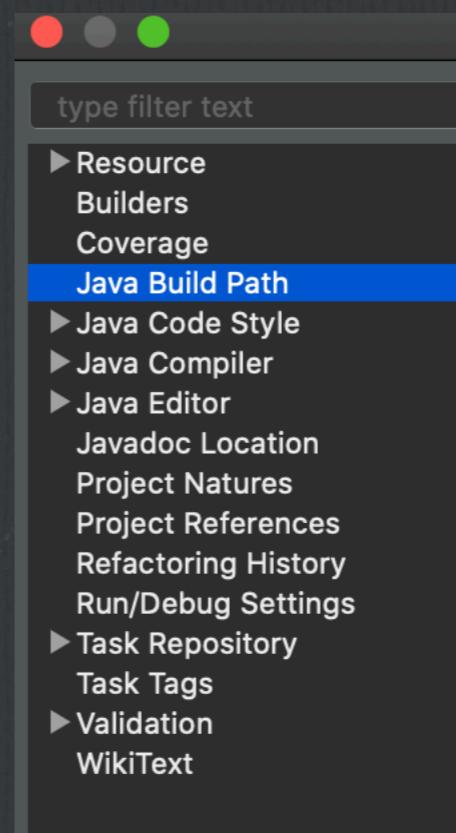
```
module pruebas {  
    requires org.junit.jupiter.api;  
}
```

Pero, aun así, nos sigue dando error, pues eclipse no ubica bien la librería JUNIT5, cosa que debemos arreglar....

# EJEMPLO I : PRUEBAS DE CLASE (VI)

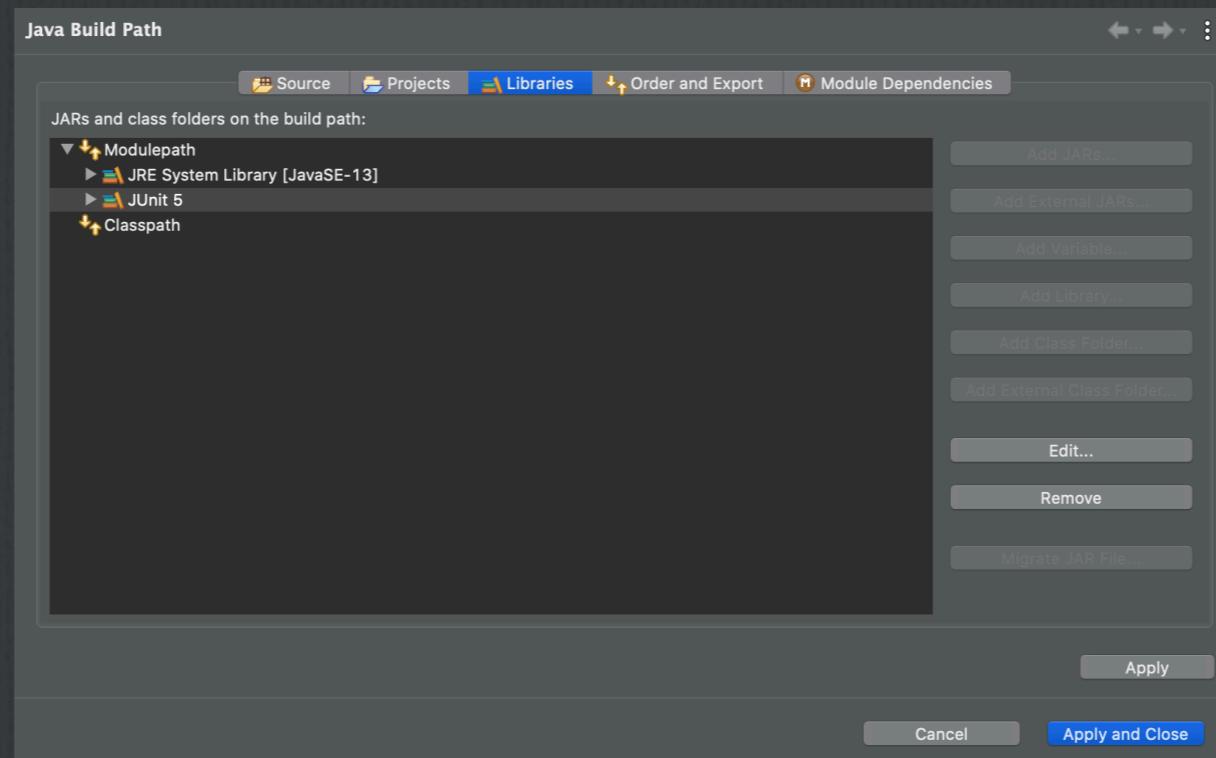


... ubicando la librería “JUnit 5” dentro de “ModulePath”. Para ello, abrimos las propiedades de nuestro proyecto (clic con el botón derecho) y en la parte izquierda elegimos, la opción “Java Build Path”



# EJEMPLO I : PRUEBAS DE CLASE (VII)

Aquí debemos mover la librería “JUnit 5” del “ClassPath” al “ModulePath”, quedando las librerías de la siguiente manera:



Pinchamos en “Apply and Close” y de esta manera deberían desaparecer los errores.

# EJEMPLO I : PRUEBAS DE CLASE (VIII)

---

Una vez solucionado esto, podemos observar nuestra clase de prueba creada automáticamente, y podemos ver algunas 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()`.
- Encima de 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.

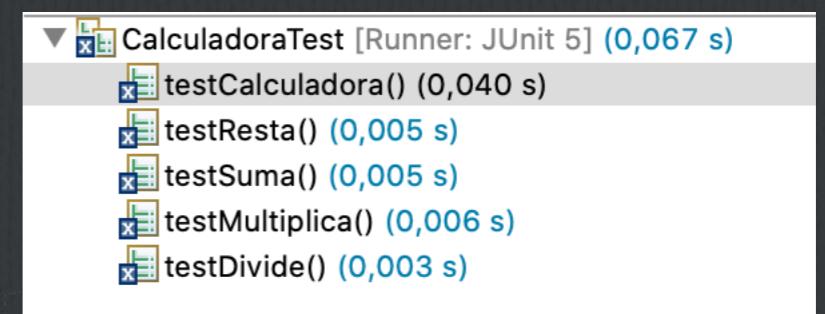
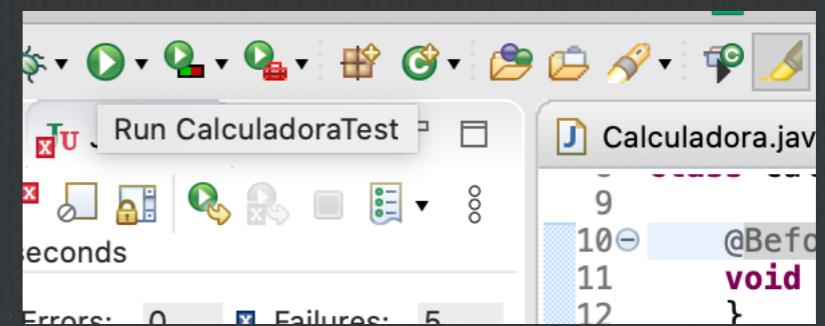
# EJEMPLO I : PRUEBAS DE CLASE (IX)

Si pulsamos ahora el botón Run, ejecutaremos el test, nos generará una ventana con los resultados. También se puede ejecutar la clase de prueba pulsando sobre la clase con el botón derecho del ratón y seleccionando Run As-> JUnit Test

Al lado de cada prueba aparece un icono con una marca indicando éxito (verificación verde), fallo (aspas azules) o error (aspas rojas).

En nuestro caso, se nos indica que se han realizado 4 pruebas, ninguna de ellas ha provocado error y todas ellas han provocado fallo.

Más adelante veremos la diferencia entre fallo y error.



# EJEMPLO I : PRUEBAS DE CLASE (IX)

Para crear nuestro test, debemos modificar cada método siguiendo los siguientes pasos:

- A) Creamos una instancia de la clase con los valores que nos interese.
- B) Invocamos al método que queremos testar.
- C) Comprobamos que el valor obtenido coincide con el valor deseado. Para ello hacemos uso de los métodos assert. (ver siguiente diapositiva)

```
Calculadora.java module-info.java CalculadoraTest.java
```

```
7
8 class CalculadoraTest {
9
10    @BeforeEach
11    void setUp() throws Exception {
12    }
13
14    @Test
15    void testCalculadora() {
16        fail("Not yet implemented");
17    }
18
19    @Test
20    void testSuma() {
21        Calculadora calcu = new Calculadora(20, 10);
22        int resultado = calcu.suma();
23        assertEquals(30, resultado);
24    }
25
26    @Test
27    void testResta() {
28        Calculadora calcu = new Calculadora(20, 10);
29        int resultado = calcu.resta();
30        assertEquals(10, resultado);
31    }
32
33    @Test
34    void testMultiplica() {
35        fail("Not yet implemented");
36    }
37
```

# EJEMPLO I : PRUEBAS DE CLASE (X). MÉTODOS ASSERT

---

**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.

**assertTrue(String mensaje, boolean expresión) :** Comprueba que la expresión se evalúe a *true*. Si no es *true* y se incluye el String, al producirse error se lanzará el *mensaje*.

**assertFalse(String mensaje, boolean expresión) :** Comprueba que la expresión se evalúe a *false*. Si no es *false* y se incluye el String, al producirse error se lanzará el *mensaje*.

**assertNull(String mensaje, Object objeto) :** Comprueba que el objeto sea *null*. Si no es *null* y se incluye el String, al producirse error se lanzará el *mensaje*.

**assertNotNull(String mensaje, Object objeto) :** Comprueba que el objeto no sea *null*. Si es *null* y se incluye el String, al producirse error se lanzará el *mensaje*.

**assertSame(String mensaje, Object objetoEsperado, Object objetoReal):** 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*.

**assertNotSame(String mensaje, Object objetoEsperado, Object objetoReal):** 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*.

**fail(String mensaje):** Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el *mensaje*.

# EJEMPLO I : PRUEBAS DE CLASE (XI). FALLO vs ERROR.

---

Podemos ver la diferencia entre un fallo y un error si, por ejemplo, cambiamos el código de dos métodos:

- En el método `multiplica()` hacemos que el valor esperado no coincida con el resultado, lo que nos devuelve un fallo. Podemos incluir un String en el método `assertEquals()` para que si se produce el fallo se lance el mensaje.
- En el método `divide()` asignamos el valor 0 al segundo parámetro, que será el denominador de la división. Al dividir por 0 se produce una excepción y saltará el error.

```
@Test
void testMultiplica() {
    Calculadora calcu = new Calculadora(20, 10);
    int resultado = calcu.multiplica();
    assertEquals(100, resultado , "Fallo en la multiplicación: " );
}

@Test
void testDivide() {
    Calculadora calcu = new Calculadora(20, 0);
    int resultado = calcu.divide();
    assertEquals(100, resultado , "Fallo en la división: " );
}
```

# EJEMPLO I : PRUEBAS DE CLASE (XII)

Al ejecutar otra vez, encontramos que `testMultiplica()` da fallo:

The screenshot shows a JUnit test runner interface. On the left, a tree view lists tests: CalculadoraTest [Runner: JUnit 5] (0,065 s) with children testCalculadora() (0,045 s), testResta() (0,003 s), testSuma() (0,003 s), testMultiplica() (0,007 s) which is selected and highlighted in blue, and testDivide() (0,003 s). On the right, a "Failure Trace" panel displays the error details: AssertionFailedError: Fallo en la multiplicación: ==> expected: <100> but was: <200>. The trace points to line 33 of CalculadoraTest.java, method testMultiplica(). The stack trace shows calls from CalculadoraTest.testMultiplica() through java.util.ArrayList.forEach() twice.

Y `testDivide()` da error:

The screenshot shows a JUnit test runner interface. On the left, a tree view lists tests: CalculadoraTest [Runner: JUnit 5] (0,065 s) with children testCalculadora() (0,045 s), testResta() (0,003 s), testSuma() (0,003 s), testMultiplica() (0,007 s), and testDivide() (0,003 s) which is selected and highlighted in blue. On the right, a "Failure Trace" panel displays the error details: java.lang.ArithmetricException: / by zero. The trace points to line 28 of Calculadora.java, method divide(). The stack trace shows calls from Calculadora.divide() through CalculadoraTest.testDivide() and java.util.ArrayList.forEach() twice.

Si pulsamos en los test que han producido fallos o errores, podemos ver la traza de ejecución, aquí en la parte derecha.

# EJEMPLO I : PRUEBAS DE CLASE (XIII). ANOTACIONES

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

**@BeforeEach:** si anotamos un método con esta etiqueta, el código será ejecutado antes de cualquier método de prueba. Podemos usarlo, por ejemplo, en una aplicación de acceso a base de datos para preparar la base de datos. En nuestro caso hemos instanciado la clase, y previamente hemos definido el objeto calcu como atributo, para poder usarlo:

```
private Calculadora calcu;  
  
@BeforeEach  
public void inicio() {  
    calcu = new Calculadora(20, 10);  
}
```

Ahora los dos test hechos ya podemos eliminar la creación de la instancia, quedando así:

```
@Test  
void testMultiplica() {  
    int resultado = calcu.multiplica();  
    assertEquals(200, resultado, "Fallo en la multiplicación: ");  
}
```

De la misma manera, existe **@AfterEach**, cuyo código será ejecutado después de la ejecución de cada uno de los métodos de prueba. Se puede utilizar para limpiar datos. Puede haber varios métodos en la clase de prueba con estas dos anotaciones.

# EJEMPLO I : PRUEBAS DE CLASE (XIV). ANOTACIONES

---

Existe otras anotaciones que permiten ejecutar código y afectan a la clase en sí. Veamos:

**@BeforeAll:** 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. Tanto los atributos modificados como la clase se deben definir como *static*.

```
private static Calculadora calcu;  
  
@BeforeAll  
public static void inicio() {  
    calcu = new Calculadora(20, 10);  
}
```

De la misma manera, existe **@Afterall**, cuyo códigoEste método será invocado un sólo vez cuando finalicen todas las pruebas. Se puede utilizar para limpiar los atributos de la clase.

Solamente puede haber un método en la clase de prueba con estas dos anotaciones.

# EJEMPLO I : PRUEBAS DE CLASE (XV). PRUEBAS PARAMETRIZADAS

---

En determinados casos, para probar un método específico, debemos hacer pruebas con varios valores de entrada y comprobar que devuelve los resultados esperados en valor y tipo. JUNIT 5 permite crear pruebas parametrizadas usando anotaciones.

Para ello, debemos usar la anotación `@ParameterizedTest` para indicar que es un test con parámetros y luego una anotación para indicar de dónde vamos a obtener los valores que vamos a pasar al método.

Existen muchas formas de generar valores, y dependerá del método a probar su elección.

Podéis ver un interesante tutorial sobre test parametrizados en el siguiente enlace:

<https://blog.codefx.org/libraries/junit-5-parameterized-tests/>

# EJEMPLO I : PRUEBAS DE CLASE (XVI). PRUEBAS PARAMETRIZADAS

---

En nuestro ejemplo vamos a probar el método división mejorado, que comprueba evitar las divisiones por cero.

El método mejorado quedaría así:

```
public int divide() {  
    if (num2 == 0)  
        return 0;  
    int resul = num1 / num2;  
    return resul;  
}
```

Como se ve, hemos hecho que devuelva 0 si el divisor es 0. Esto no es lo más adecuado, pues debería lanzar una excepción, pero de momento nos vale así.

Ahora vamos a crear un archivo JUNIT vacío y vamos a crear un test para este método. Como fuente de datos vamos a elegir `@CsvSource`, puesto que es muy sencilla de entender y nos vale para nuestro caso, que únicamente maneja datos de tipo *int*. Nuestro método tendrá pues tres parámetros de entrada: los dos valores de los números y el resultado esperado. Podemos usar, o no, la anotación `@BeforeAll`. El ejemplo quedaría como sigue ...

# EJEMPLO I : PRUEBAS DE CLASE (XVII). PRUEBAS PARAMETRIZADAS

Vemos ahora que el `@CsvSource` incluye tres elementos separados por coma y encerrados entre comillas. Éstos son los parámetros que le vamos a pasar al método:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class Parametrizadas {

    @ParameterizedTest
    @DisplayName("calcular división")
    @CsvSource({
        "100,2,50",
        "40,5,8",
        "20, 0 , 0"})
    void pruebaDivide(int a, int b, int esperado) throws Exception {
        Calculadora calculator = new Calculadora(a , b);
        int result = calculator.divide();
        assertEquals(esperado, result);
    }
}
```

Y el resultado es:



# EJEMPLO I : PRUEBAS DE CLASE (XVIII). PRUEBAS PARAMETRIZADAS

Si queremos manejar parámetros de diferentes tipo, podemos usar un método como argumento.

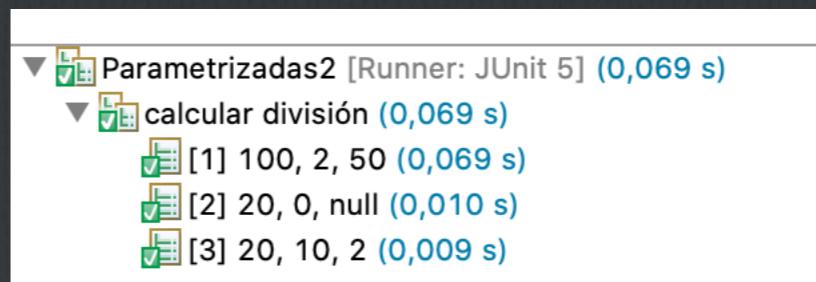
Para ello, necesitamos echar mano del Tipo Stream y del tipo Arguments que nos ofrece JUNIT para enviar un objeto con diferentes valores y tipos, el método .of de la clase Stream nos permite ir añadiendo uno por uno los diversos valores, que pueden ser de distinto tipo.

```
import static org.junit.jupiter.api.Assertions.*;
import java.util.stream.Stream;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

class Parametrizadas2 {

    @ParameterizedTest
    @DisplayName("calcular división")
    @MethodSource("parametrosMetodoDivide")
    void pruebaDivide(int a, int b, Integer esperado) throws Exception {
        Calculadora calcu = new Calculadora(a, b);
        Integer result = calcu.divide();
        assertEquals(esperado, result);
    }

    private static Stream<Arguments> parametrosMetodoDivide() {
        return Stream.of(
            Arguments.of(100, 2, 50),
            Arguments.of(20, 0, null),
            Arguments.of(20, 10, 2));
    }
}
```



# REFERENCIA

---

- [https://es.wikipedia.org/wiki/Pruebas\\_de\\_software](https://es.wikipedia.org/wiki/Pruebas_de_software)
- <https://www.meetup.com/es-ES/MadridJUG/events/246018603>
- <https://www.adictosaltrabajo.com/2016/11/24/primeros-pasos-con-junit-5/>
- <https://www.arquitecturajava.com/java-9-modules/>
- <http://java-white-box.blogspot.com/p/junit.html>
- <https://www.baeldung.com/parameterized-tests-junit-5>
- <https://www.geeksforgeeks.org/stream-in-java/>