UT3 - DISEÑO Y REALIZACIÓN DE PRUEBAS – PROCEDIMIENTOS DE PRUEBAS Y TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA – PARTE III

RESULTADOS DE APRENDIZAJE ASOCIADOS
3.- Verifica el funcionamiento de programas diseñando y realizando pruebas.
CRITERIOS DE EVALUACIÓN
a) Se han identificado los diferentes tipos de pruebas.
b) Se han definido casos de prueba
c) Se han identificado las herramientas de depuración y prueba de aplicaciones ofrecidas por el entorno de desarrollo.
d) Se han utilizado herramientas de depuración para definir puntos de ruptura y seguimiento.
e) Se han utilizado las herramientas de depuración para examinar y modificar el comportamiento de un programa en tiempo de ejecución.
f) Se han efectuado pruebas unitarias de clases y funciones.
g) Se han implementado pruebas automáticas.
h) Se han documentado las incidencias detectadas.

UT3 - DISEÑO Y REALIZACIÓN DE PRUEBAS – PROCEDIMIENTOS DE PRUEBAS Y TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA – PARTE III

Índice de contenido

1.- Contenidos

En esta sección, abordaremos la esencialidad de las **herramientas de depuración** y **las pruebas unitarias** en la ingeniería de software, subrayando su papel fundamental en la elevación de la calidad del producto final. Exploraremos cómo la identificación precisa de diferentes tipos de pruebas, junto con la implementación de casos de prueba bien definidos, establece las bases para un análisis exhaustivo y detallado. Profundizaremos en el uso estratégico de herramientas de depuración integradas en el entorno de desarrollo, resaltando cómo facilitan la identificación y corrección de errores mediante puntos de ruptura y seguimiento en tiempo real. Además, se destacará la importancia de las **pruebas unitarias** en la verificación de la funcionalidad individual de clases y funciones, y cómo su **automatización** contribuye significativamente a la eficiencia del proceso de desarrollo.

Mientras que, en epígrafes anteriores del tema, abordábamos las pruebas, planificación de pruebas, etc. desde un plano teórico, **en esta parte del tema lo haremos desde un plano más práctico**.

2.- Herramientas de depuración

El inicio del proceso de depuración se da con la realización de un caso de prueba específico.

Durante este proceso, se analizan los resultados obtenidos y se compara si estos coinciden o difieren de los resultados esperados.

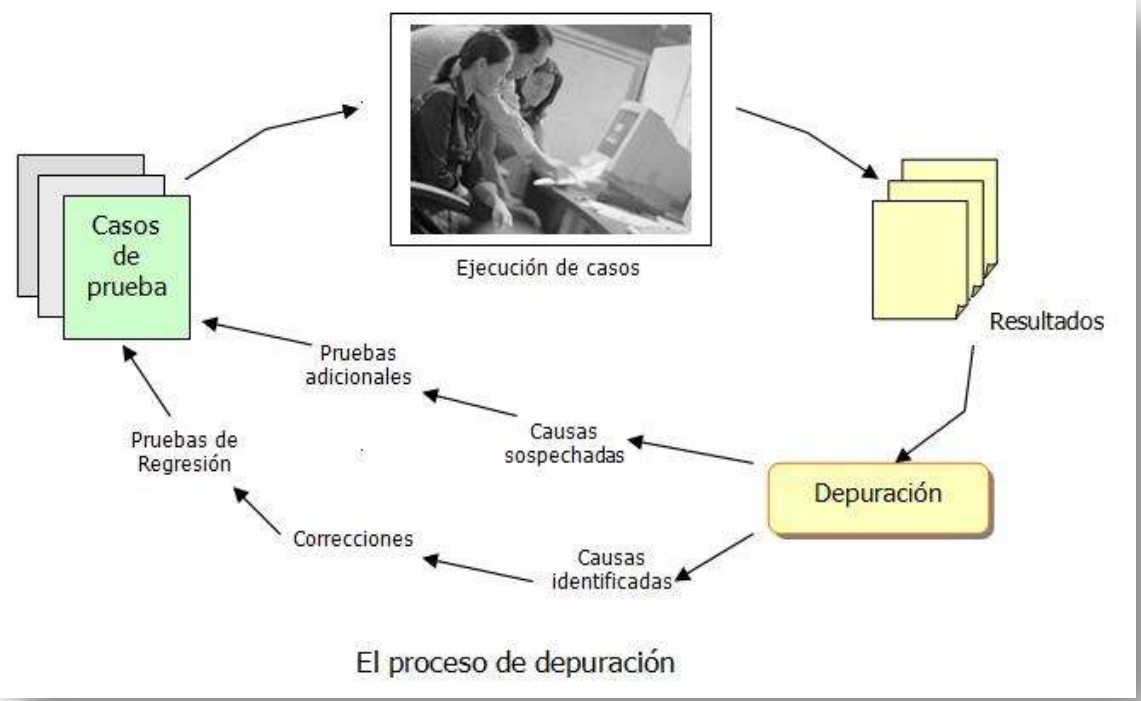
El proceso de depuración puede concluir de dos maneras distintas:

- Se identifica y corrige la fuente del fallo, eliminando el error.
- Si no se localiza la fuente del error, el responsable de la depuración debe formular hipótesis sobre la posible causa, desarrollar nuevos casos de prueba que permitan verificar estas suposiciones y llevar a cabo pruebas adicionales para hallar y solventar los errores (incluyendo **pruebas de regresión** y la repetición selectiva de ciertas pruebas para descubrir problemas surgidos durante la modificación).

Interesante artículo (proceso de depuración de errores):

https://www.velneo.com/blog/14-pasos-para-depurar-tu-software#:~:text=,Image%209%3A%20pasos%20depuraci%C3%B3n%20software

A continuación, se presenta un esquema del proceso de depuración:



El proceso de depuración

Recapitulemos…

Fijaos que la depuración es una etapa posterior a la ejecución de los casos de prueba. Por tanto:

1. **Se diseñan los casos de prueba (en papel, por ejemplo)**
2. **Se crean las pruebas automatizadas en código (estas pruebas podrían ser automatizadas o no).**
3. **Se ejecutan dichas pruebas.**

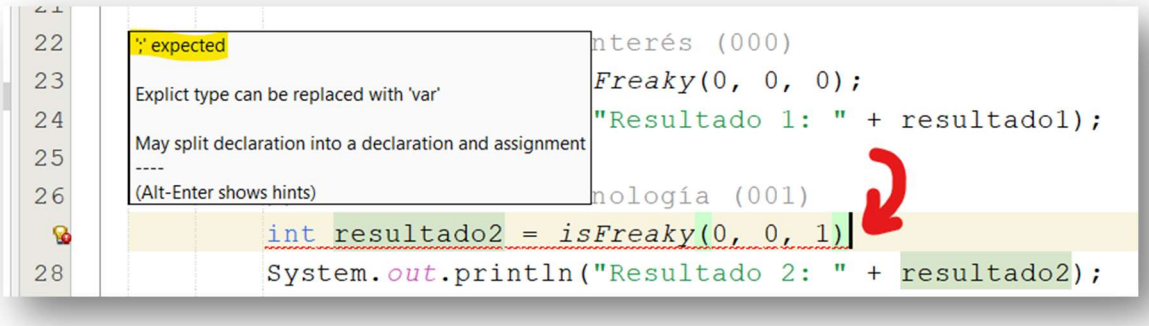
4. **Se observan los resultados**
5. **Aquí pueden ocurrir dos cosas**
    a. **El código pasó la prueba**
    b. **El código NO pasó la prueba**
        i. **Comenzamos la etapa de depuración de dicho código que no pasó las pruebas.**

2.1.- Tipos de errores que podemos cometer

Al desarrollar programas cometemos dos tipos de errores:

- **Errores de compilación** → La corrección de errores es en general sencilla, especialmente cuando se utiliza un Entorno de Desarrollo Integrado (IDE). Este tipo de herramientas no solo señala la ubicación exacta del error, sino que también ofrece sugerencias y posibles soluciones para resolverlo.

Ejemplo:



- **Errores lógicos** → Estos errores resultan más complejos de identificar, dado que el programa puede compilar correctamente sin presentar errores de sintaxis, pero al ejecutarse, puede generar resultados inesperados o incorrectos.

A estos errores de tipo lógico se les suele llamar **bugs**

Los entornos de desarrollo incorporan una herramienta conocida como **depurador** (o **debugger**) para ayudarnos a resolver este tipo de errores.

El depurador nos permite:

- **Analizar el código del programa mientras se ejecuta.**
- **Establecer puntos de interrupción o de ruptura.**
- **Suspender la ejecución del programa**
- **Ejecutar código paso a paso.**
- **Examinar el contenido de las variables**

2.2.- Cómo utilizar el depurador

El uso del depurador es bastante intuitivo, especialmente si se familiariza con uno en un Entorno de Desarrollo Integrado (IDE). Esta familiaridad **facilita su uso en otros IDEs**, ya que **el proceso de depuración suele ser consistente y los controles y botones tienden a ser similares en diferentes entornos**. Por lo tanto, aprender a manejar un depurador en un IDE específico proporciona una base sólida para trabajar con otros.

El proceso para lanzar un depurador en un Entorno de Desarrollo Integrado (IDE) suele seguir estos pasos básicos:

1. **Abrir el Proyecto:** Iniciar el IDE y abrir el proyecto que contiene el código a depurar.
2. **Configurar Puntos de Interrupción:** Establecer puntos de interrupción en las líneas de código donde se desea detener la ejecución para inspeccionar el comportamiento del programa.
3. **Iniciar la Depuración:** Seleccionar la opción para iniciar el modo de depuración en el IDE. Esto puede ser un botón o una opción en el menú.
4. **Ejecución Paso a Paso:** Una vez que el programa se detiene en un punto de interrupción, utilizar las opciones de ejecución paso a paso para avanzar a través del código, observando el flujo de ejecución y el estado de las variables.
5. **Inspeccionar Variables:** Verificar los valores de las variables en los puntos de interrupción para identificar posibles discrepancias o errores.

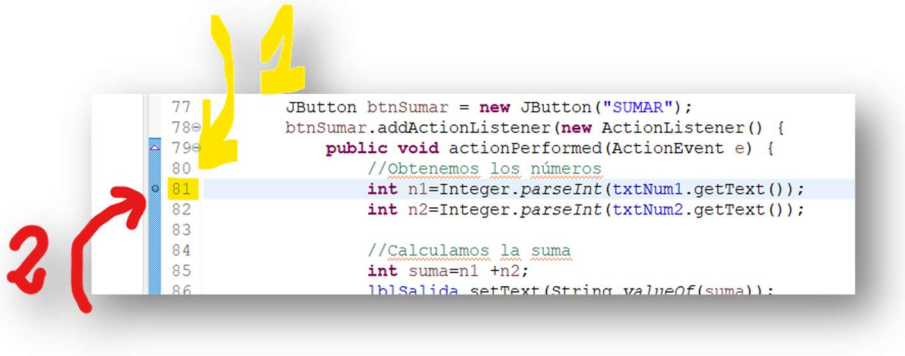
6. **Continuar o Detener la Ejecución:** Después de inspeccionar y modificar si es necesario, se puede continuar la ejecución hasta el próximo punto de interrupción o detener la depuración.
7. **Modificar y Repetir:** Si se identifican problemas, modificar el código y repetir el proceso de depuración para asegurarse de que los cambios hayan resuelto los errores.

Este proceso puede variar ligeramente dependiendo del IDE específico y del lenguaje de programación utilizado.

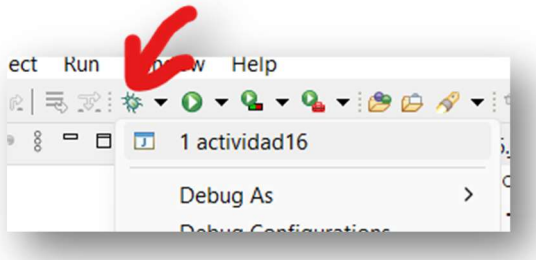
2.2.1.- Cómo lanzar el depurador en Eclipse

Para lanzar el depurador en Eclipse haremos lo siguiente:

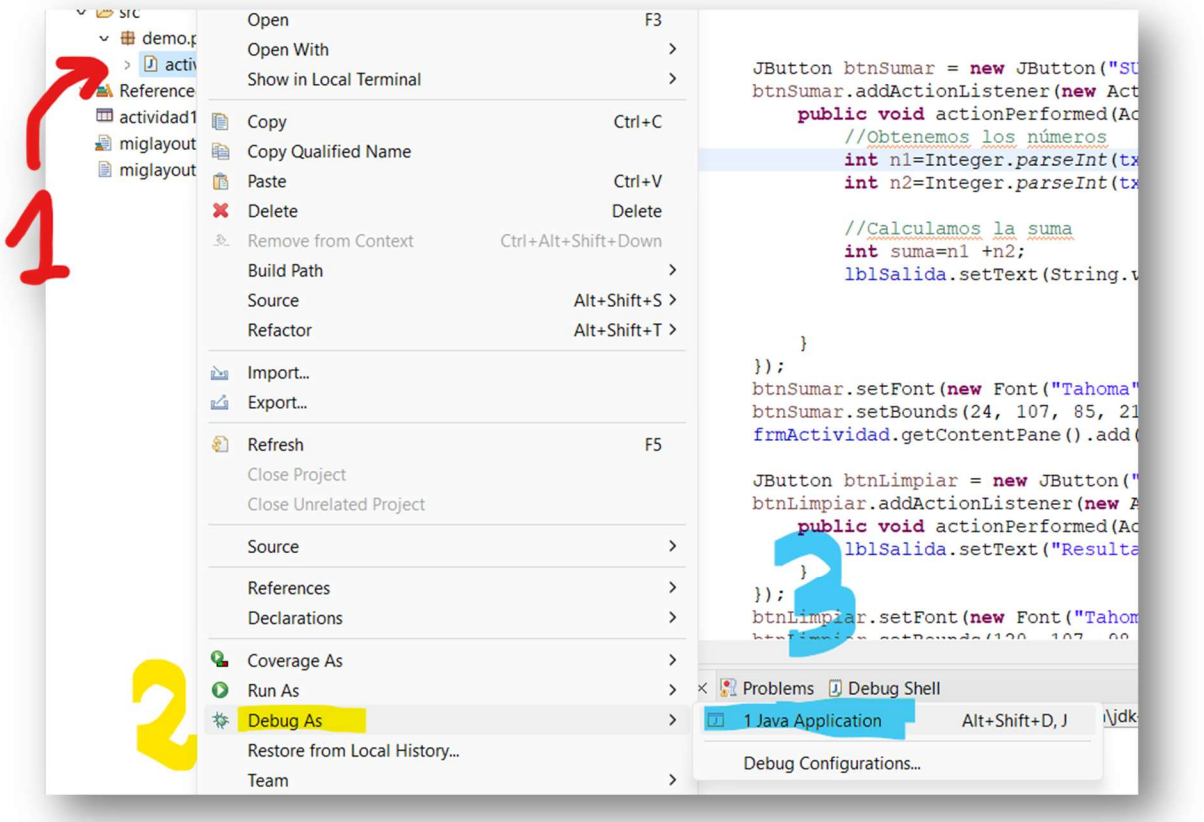
1. Abrimos el proyecto que queramos.
2. Configuramos el punto de interrupción
   a. Hacemos doble clic en el número de línea donde queramos establecer el **punto de ruptura** (o **breakpoint**)
   b. Aparecerá un puntito en dicha línea. El programa parará su ejecución justo en esa línea.



3. Ahora ejecutaremos el programa en modo depuración. Podemos hacerlo de varias formas:
   a. Seleccionando el menú **RUN >> Debug**



b. Mediante el menú contextual que se muestra al hacer clic con el botón derecho del ratón en la clase que se va a ejecutar. En este caso seleccionamos **Debug As >> Java Application.**



Para diferentes IDEs el proceso será similar o el mismo.

Probemos a lanzar el depurador con otros IDEs…

- Netbeans
- VisualStudioCode

NOTA: La complicación entre unos IDEs y otros radica en cómo añadir el **breakpoint** o **punto de ruptura**.

Al iniciarse el modo de depuración, el IDE nos preguntará si queremos cambiar de vista al modo depuración.

En esta vista se pueden ver varias zonas:

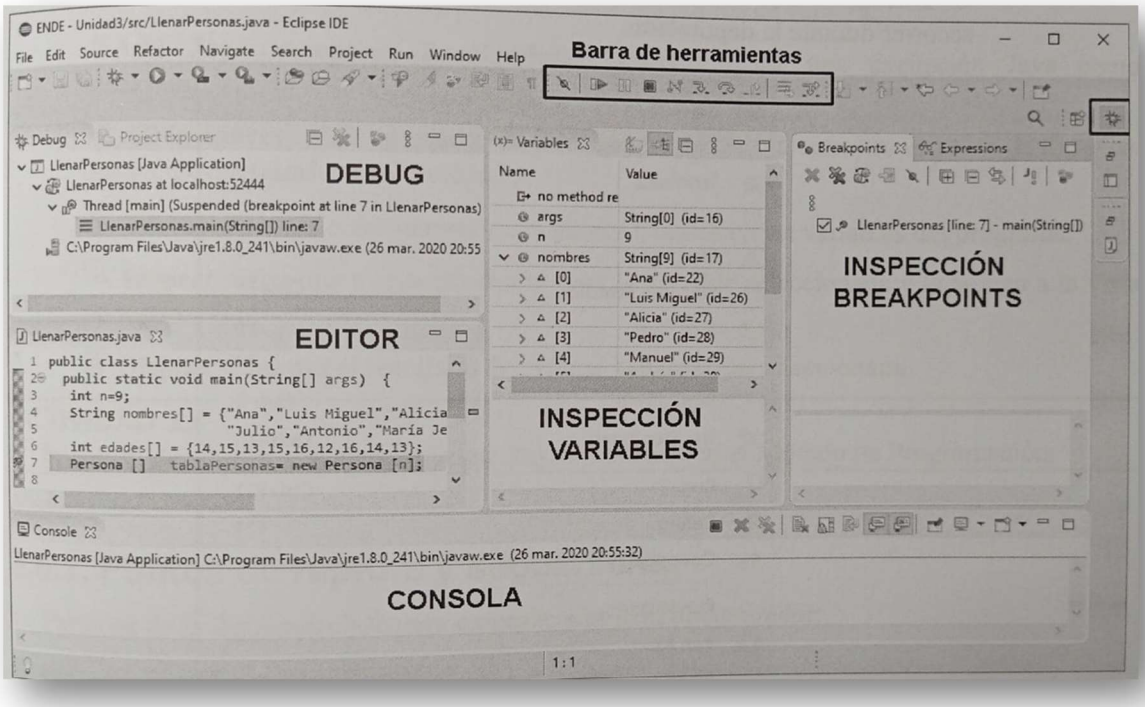


FIGURA 1. FOTO EXTRAÍDA DEL LIBRO DE RAMOS MARTÍN A. Y RAMOS MARTÍN M.J. GRUPO EDITORIAL GARCETA.

La imagen muestra la interfaz de Eclipse IDE en la perspectiva de depuración. Las zonas se pueden describir de la siguiente manera:

- **Barra de Herramientas:** Contiene iconos para las acciones más comunes en la depuración, como ejecutar paso a paso, reanudar y detener la ejecución, entre otros.
- **Zona de Debug:** Muestra la jerarquía de procesos y *threads* (hilos) en ejecución, y los puntos de interrupción activos.
- **Editor:** Es el área donde se visualiza y edita el código fuente, y donde se establecen los puntos de interrupción.
- **Inspección de Variables**: Ventana que lista las variables en el contexto actual y sus valores.
- **Inspección de Breakpoints:** Lista los puntos de interrupción establecidos.
- **Consola:** Muestra la salida del programa y mensajes del sistema durante la depuración.

Investiguemos…

¿Qué es un *Thread* en informática y por qué son tan importantes?

A continuación, especificaremos los controles de la **barra de herramientas** con más detalle, ya que es muy importante conocerlos a la perfección.

Algunas opciones son las siguientes:

Resume (o tecla *F8*). Esta acción reactiva un proceso que estaba en pausa, permitiendo que el programa retome su curso. Es útil cuando se prefiere evitar el análisis detallado de cada instrucción y se desea que el depurador se detenga en el próximo *breakpoint* establecido.

Suspend. Suspende el hilo seleccionado, es decir, suspende la ejecución del programa que se está ejecutando.

Terminate (o *Ctrl+F2*). Finaliza el proceso de depuración.

Step Into (o tecla *F5*). Se ejecuta paso a paso cada instrucción. Si el depurador encuentra una llamada a un método o a una función, al pulsar en *Sept Into* se irá a la primera instrucción de dicho método.

Step Over (o tecla *F6*). Se ejecuta paso a paso cada instrucción, pero si el depurador encuentra un método, al pulsar *Step Over* se irá a la siguiente instrucción, sin entrar en el código del método.

Step Return (o tecla F7). Si nos encontramos dentro de un método, el depurador sale del método y vuelve a la instrucción que llamó a dicho método.

Ejercicio de clase…

Encuentra el error en el código y arréglalo para que todo funcione correctamente.

Obligatorio utilizar:

- **Eclipse**
- **Depurador (*Debugger*) de Eclipse**

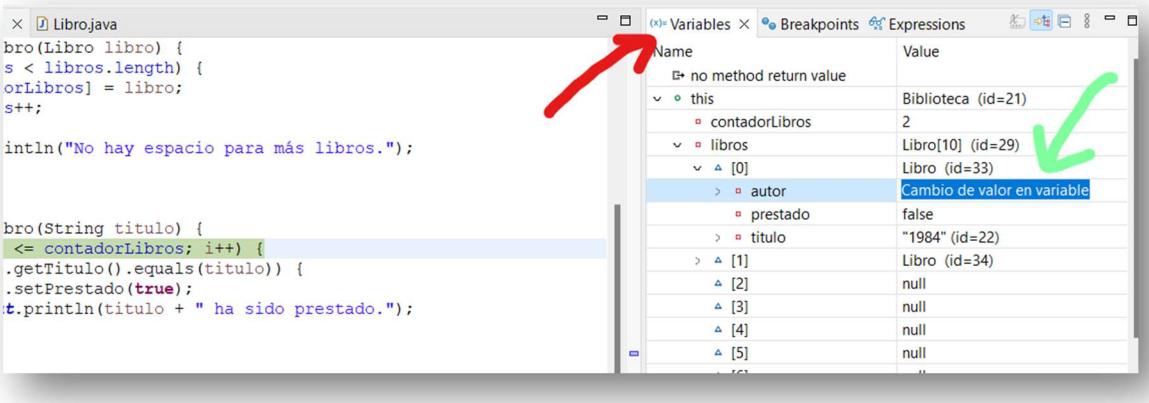
NOTA: Usa el ejemplo de la biblioteca que está subido en el aula.

2.2.1.1.- Examen y modificación de variables

Hemos podido ver que desde la vista de ***INSPECCIÓN DE VARIABLES*** o desde la pestaña de ***Variables*** podemos inspeccionar las variables definidas en el punto en el que el programa está detenido en ese momento.

Desde esta vista también se puede modificar el valor de una variable tan solo haciendo clic sobre ella y escribiendo el valor deseado.

Ejemplo:



3.- Pruebas unitarias con Junit

Hasta el momento, nuestras pruebas se han basado en ejecuciones manuales siguiendo las directrices especificadas para un código dado. En esta sección, nos enfocaremos en el uso de una herramienta específica diseñada para ejecutar pruebas automatizadas, asegurando así que nuestro software funcione como se espera.

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.

3.1.- Creación de una clase de prueba

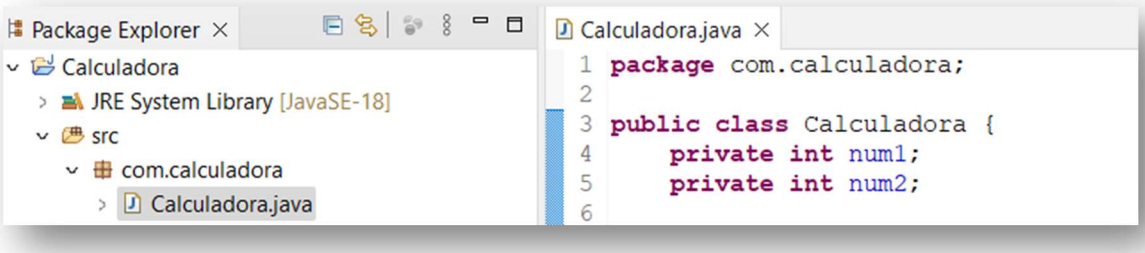
Para crear una clase de prueba con ***JUnit*** seguiremos estos pasos:

1) Creamos un nuevo proyecto en Eclipse
2) Creamos la clase a probar, que en este caso se llama *Calculadora*.

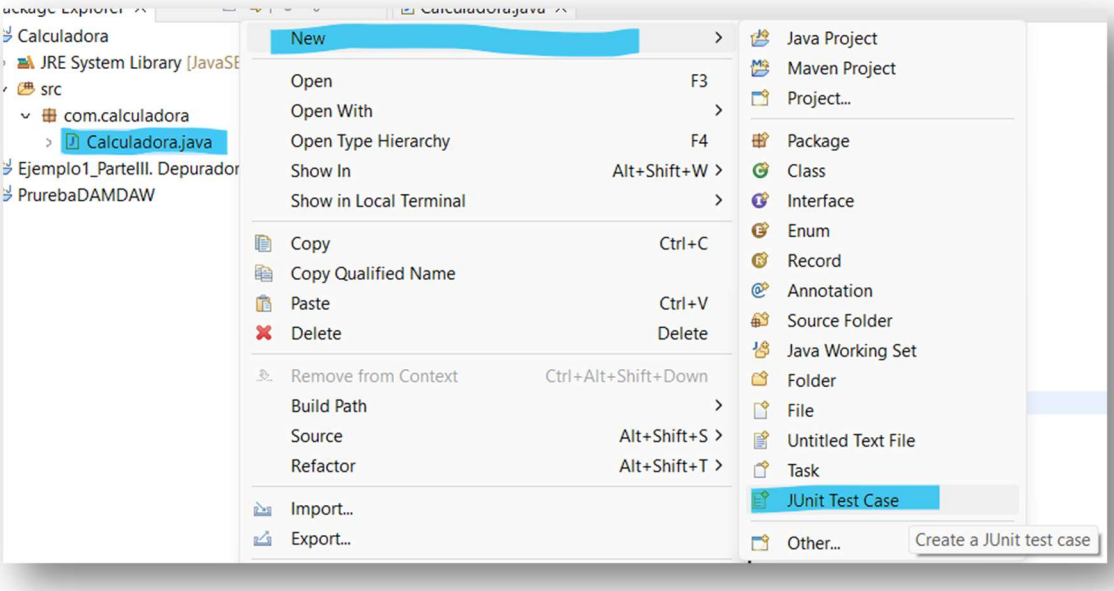
```
public class Calculadora {
    private int num1;
    private int num2;

    /**
     * Constructor de la clase Calculadora.
     *
     * @param num1 El primer número.
     * @param num2 El segundo número.
     */
    public Calculadora(int num1, int num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    /**
     * Suma los números.
     *
     * @return La suma de num1 y num2.
     */
```

```
    public int suma() {
        return num1 + num2;
    }

    /**
     * Resta los números.
     *
     * @return La resta de num1 y num2.
     */
    public int resta() {
        return num1 - num2;
    }

    /**
     * Multiplica los números.
     *
     * @return El producto de num1 y num2.
     */
    public int multiplicar() {
        return num1 * num2;
    }

    /**
     * Divide los números.
     *
     * @return El cociente de num1 y num2.
     * @throws ArithmeticException si num2 es 0.
     */
    public int dividir() {
        if (num2 == 0) {
            throw new
ArithmeticException("División por cero");
        }
        return num1 / num2;
    }

    // Métodos getters y setters para num1 y num2

    public int getNum1() {
        return num1;
    }
```

```
    public void setNum1(int num1) {
        this.num1 = num1;
    }

    public int getNum2() {
        return num2;
    }

    public void setNum2(int num2) {
        this.num2 = num2;
    }
}
```

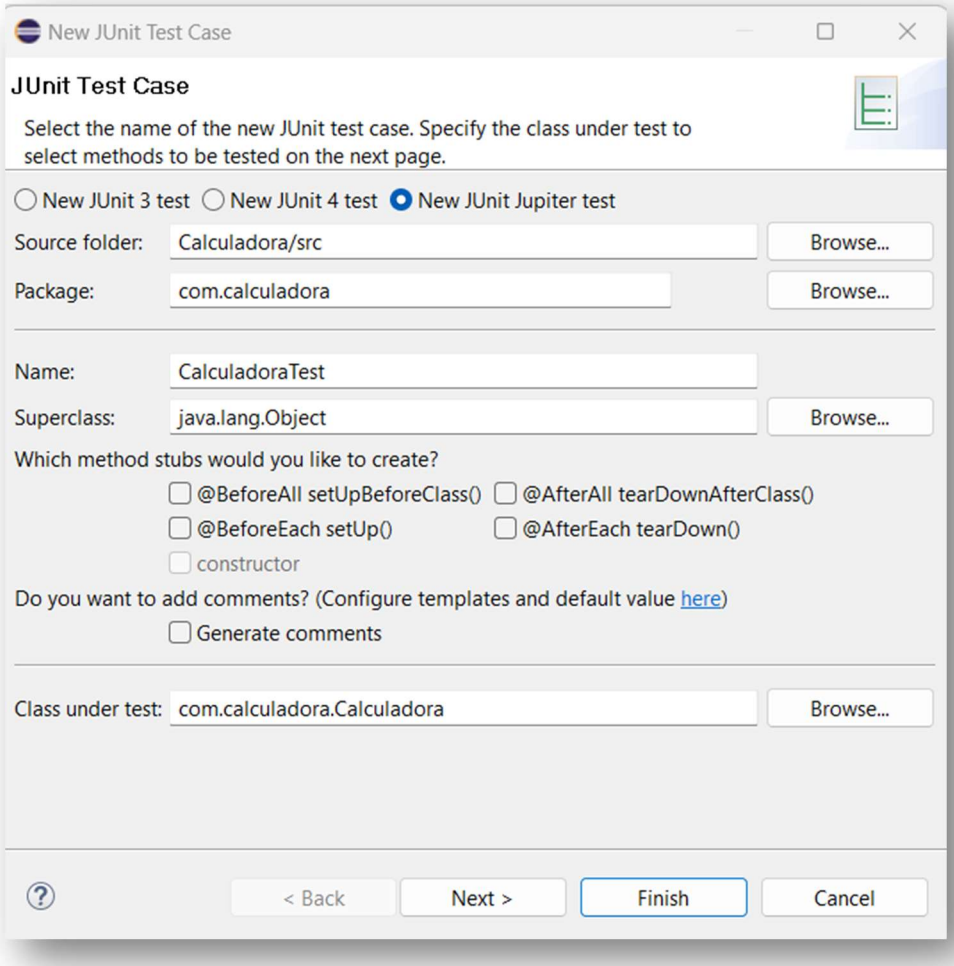


3) Se crea la clase de prueba. Con la clase **Calculadora** seleccionada, pulsamos el botón derecho del ratón y seleccionamos ***New >> JUnit Test Case***

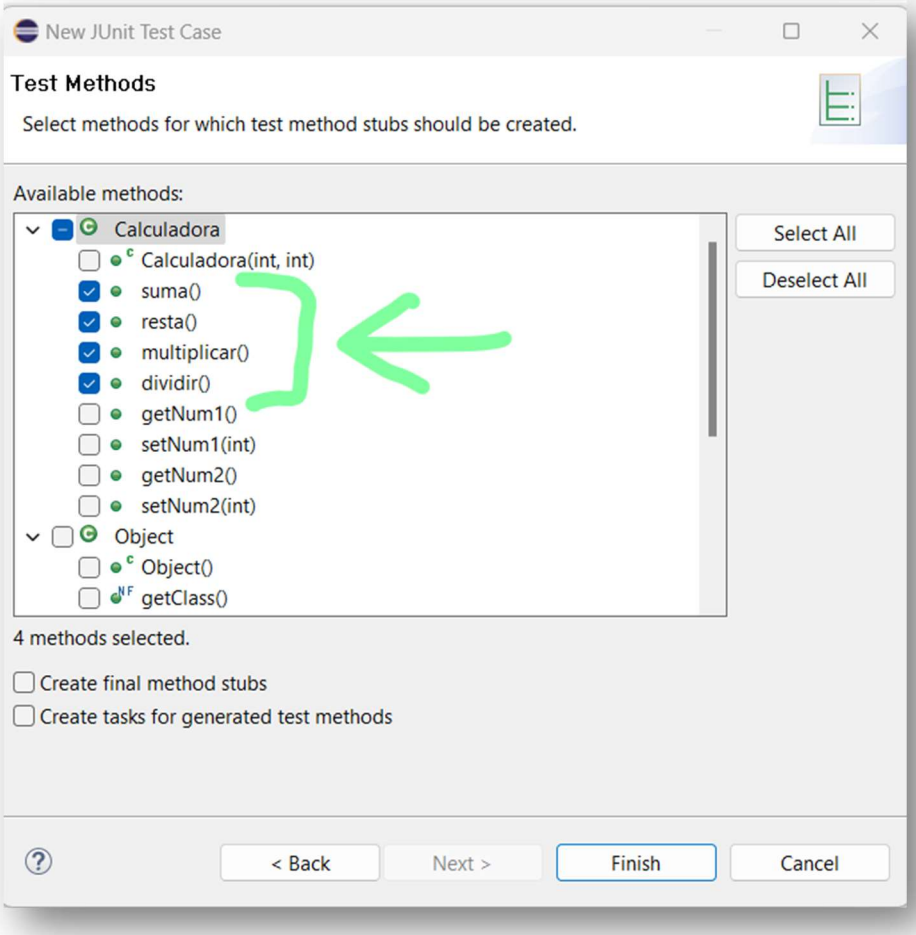


También se puede hacer desde el menú ***File >> New >> JUnit Test Case***

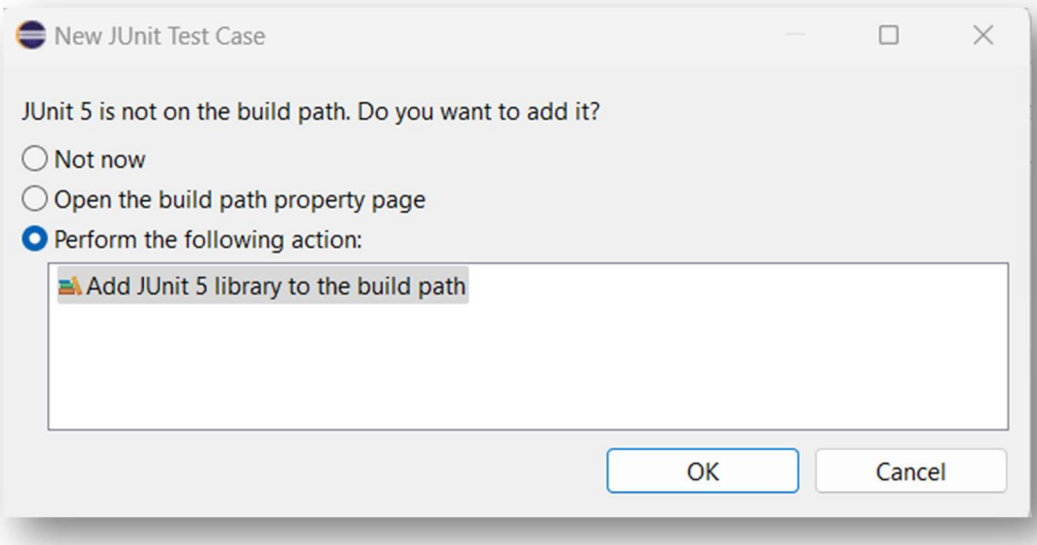
4) Se nos abrirá la siguiente ventana de diálogo. Aquí deberemos marcar ***New JUnitJupiter test.*** El resto de las opciones las dejamos por defecto.
   Se generará una clase test por defecto con el nombre ***CalculadoraTest.***



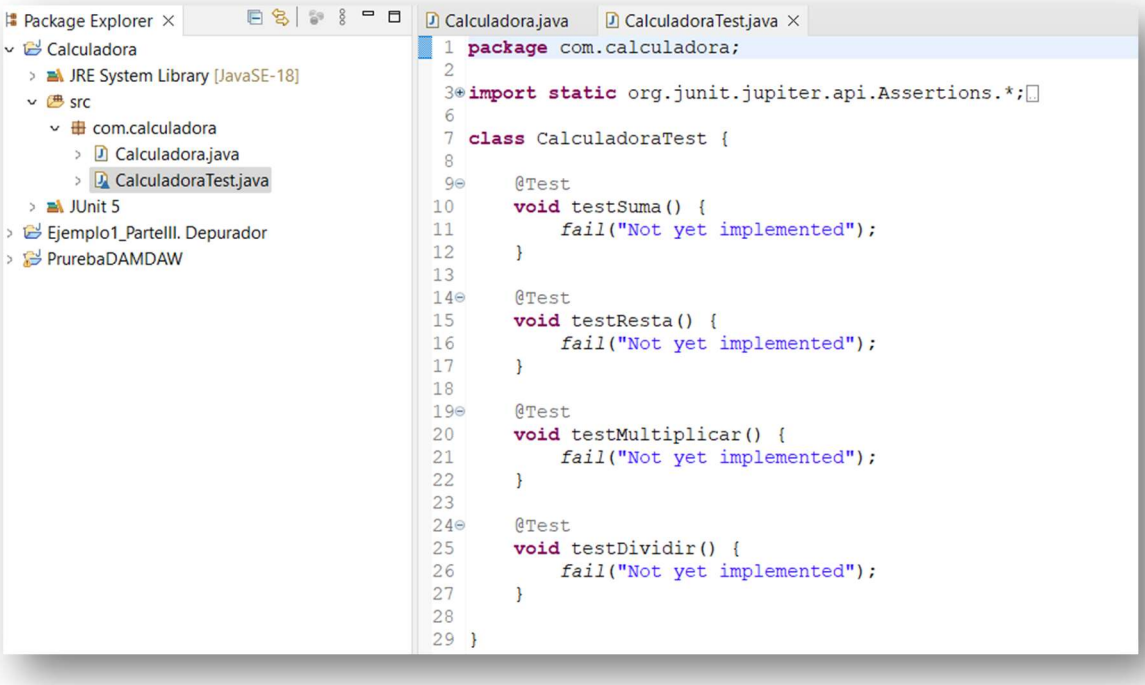
5) Pulsamos en el botón "*Next*". Aquí deberemos seleccionar los métodos que deseamos probar. Hacemos clic en el botón "*Finish*".



6) La librería de *JUnit* no está incluida en nuestro proyecto, por lo que nos saldrá una ventana para incluirla. Pulsamos en el botón de *"OK*".



En este punto, ya tenemos creada la clase test. Y se muestra de la siguiente forma:



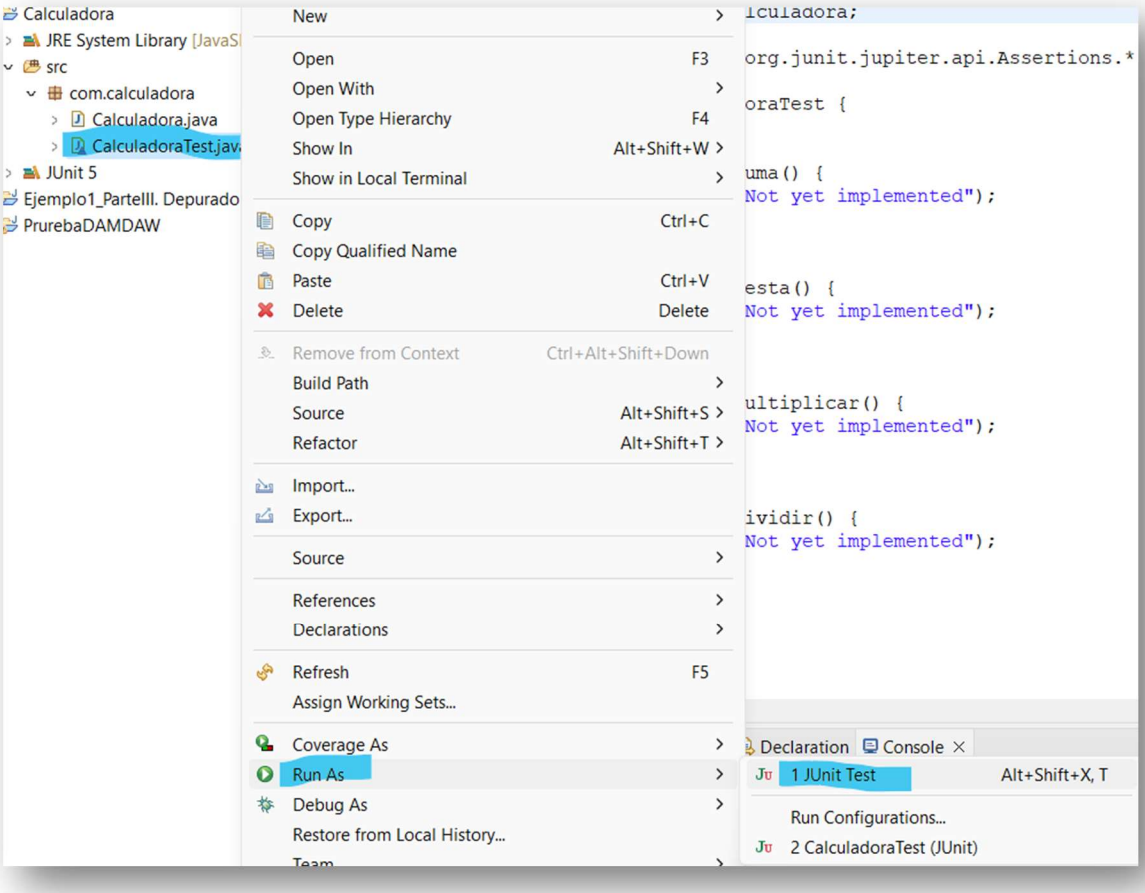
Varios detalles:

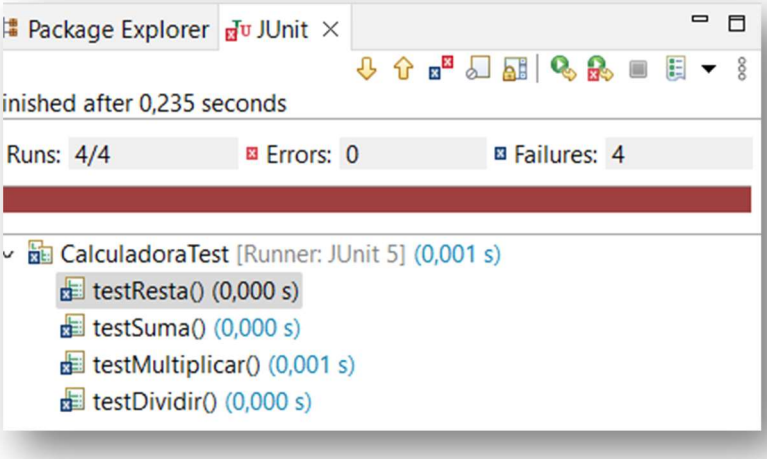
- Sobre cada uno de los métodos aparece la anotación **@Test** que indica al compilador que es un método de prueba.

- 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.

Vamos a verlo:

- Hacemos clic con el botón derecho en la clase CalculadoraTest y **RunAs >> JUnit Test**





3.2.- 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*.

ASERCIONES EN JUNIT

Método	Descripción
assertTrue(boolean condition, String message)	Asegura que la condición booleana sea verdadera. Muestra el mensaje si la aserción falla.
assertFalse(boolean condition, String message)	Asegura que la condición booleana sea falsa. Muestra el mensaje si la aserción falla.
assertEquals(Object expected, Object actual, Double delta, String message) (Se puede usar con cualquier tipo de dato, Integer, Short, Object,etc;*delta* ser usa en los tipos flotar y double)	Asegura que dos objetos sean iguales. Muestra el mensaje si la aserción falla. Para arrays, compara el contenido. El ***delta***, 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 ambos números deben de ser iguales. Un valor de 0.01 consideraría estos dos números iguales: 1259.9916 y 1259.9917. Un valor de "0" los consideraría distintos.
assertSame(Object expected, Object actual, String message)	Asegura que dos objetos sean exactamente el mismo objeto (comparación de identidad). Muestra el mensaje si la aserción falla.
assertNotSame(Object unexpected, Object actual, String message)	Asegura que dos objetos no sean el mismo objeto (comparación de identidad).

Método	Descripción
	Muestra el mensaje si la aserción falla.
assertNull(Object actual, String message)	Asegura que un objeto sea nulo. Muestra el mensaje si la aserción falla.
assertNotNull(Object actual, String message)	Asegura que un objeto no sea nulo. Muestra el mensaje si la aserción falla.
fail() **fail(String message)**	Hace que la prueba falle. Si se incluye un String, la prueba falla lanzando el *mensaje.*

NOTA: Hay muchos más métodos.

TIPOS DE ANOTACIONES (JUnit 5)

Anotación	Descripción
@Test	Indica que el método anotado es un método de prueba.
@BeforeEach	Se ejecuta antes de cada método de prueba en la clase actual, utilizado para la preparación del entorno de prueba.
@AfterEach	Se ejecuta después de cada método de prueba en la clase actual, comúnmente utilizado para la limpieza del entorno de prueba.
@BeforeAll	Se ejecuta una sola vez antes de todos los métodos de prueba en la clase actual. Útil para configuraciones costosas. Puede ser no estático si la clase de prueba se anota con @TestInstance(Lifecycle.PER_CLASS).
@AfterAll	Se ejecuta una sola vez después de todos los métodos de prueba en la clase actual. Utilizado para actividades de cierre. Puede ser

Anotación	Descripción
	no estático si la clase de prueba se anota con @TestInstance(Lifecycle.PER_CLASS).
@Disabled	Indica que un método de prueba o clase de prueba no debe ejecutarse.

NOTA: Hay muchas más anotaciones.

La tablas anteriores fueron extraídas de la documentación oficial de JUnit 5.

https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

3.3.- Creación de una prueba

Siguiendo con el ejemplo anterior (el de la Calculadora), 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
Multiplicar	20, 10	200
Dividir	20, 10	2

Todo caso de prueba se compone básicamente de 3 partes:

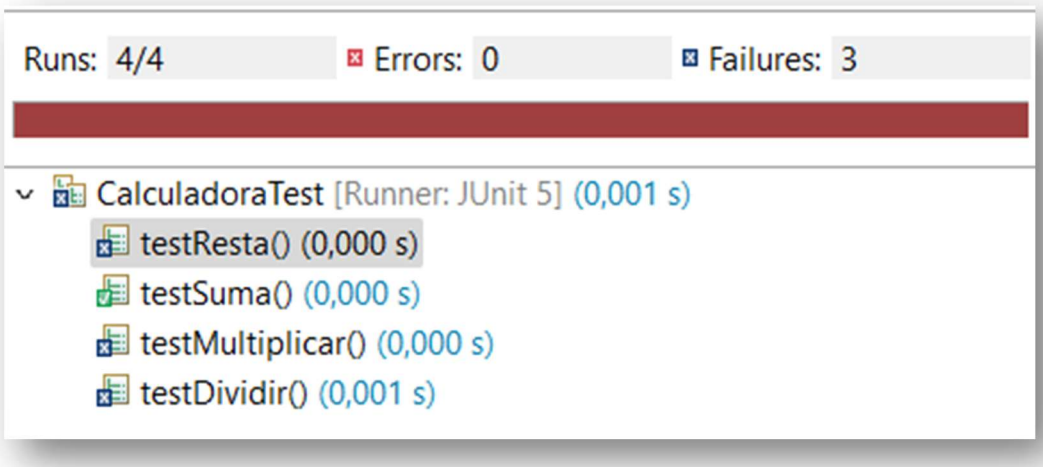
1) Se indica en una variable cuál es el valor esperado por el método que vamos a probar
2) Se ejecuta el método que queremos probar con los datos de entrada adecuados y se guarda el resultado en otra variable.
3) Se comprueba la relación entre el valor esperado y el resultado de la llamada al método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

Ejemplo:

Vamos a crear el código de prueba para el método ***testSuma()*** que probará el método ***suma()*** de la clase ***Calculadora.***

```
@Test
 void testSuma() {
      double valorEsperado=30;
      Calculadora calc=new Calculadora(20,10);
      double resultado=calc.suma();
      assertEquals(valorEsperado, resultado,0);
 }
```

Para comprobar si todo esto funciona, ejecutaremos la clase de prueba, como vimos antes:



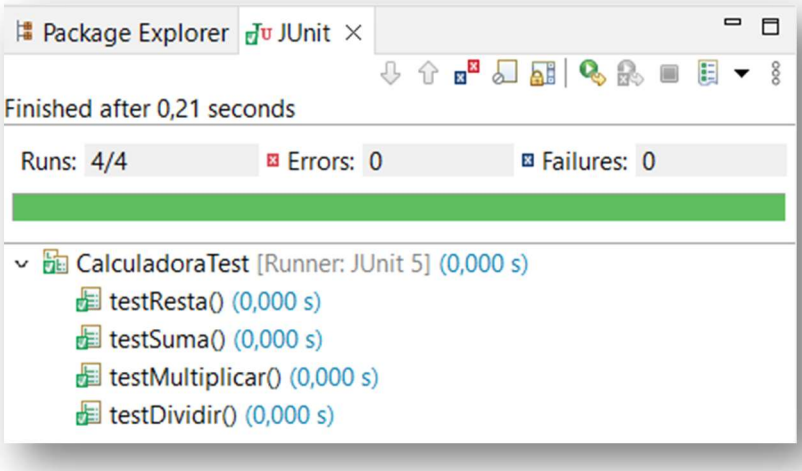
En la imagen anterior, podemos observar como el testSuma tiene un tic verde, por lo que podemos decir que ha pasado la prueba de manera satisfactoria, mientras que los demás tienen un aspa con fondo azul.

El resultado de la ejecución de la prueba muestra: **Runs: 4/4 - Errors: 0 - Failures: 3**

Esto quiere decir que se han realizado 4 pruebas, ninguna de ellas ha provocado ningún error y 3 de ellas no han superado su test o prueba correspondiente.

Completemos el ejemplo en clase.

Ahora tendréis que hacer vosotros las pruebas que faltan. Nuestro objetivo es que pase el test y veamos lo siguiente:

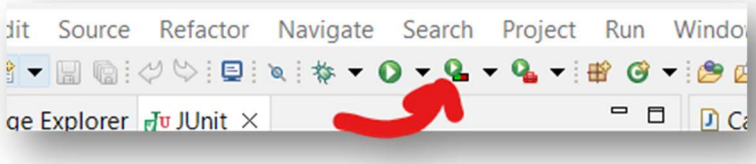


3.4.- Medición de la cobertura de código

Una vez que hemos ejecutado los casos de prueba y hemos visto que funciona correctamente, nos hacemos la siguiente pregunta:

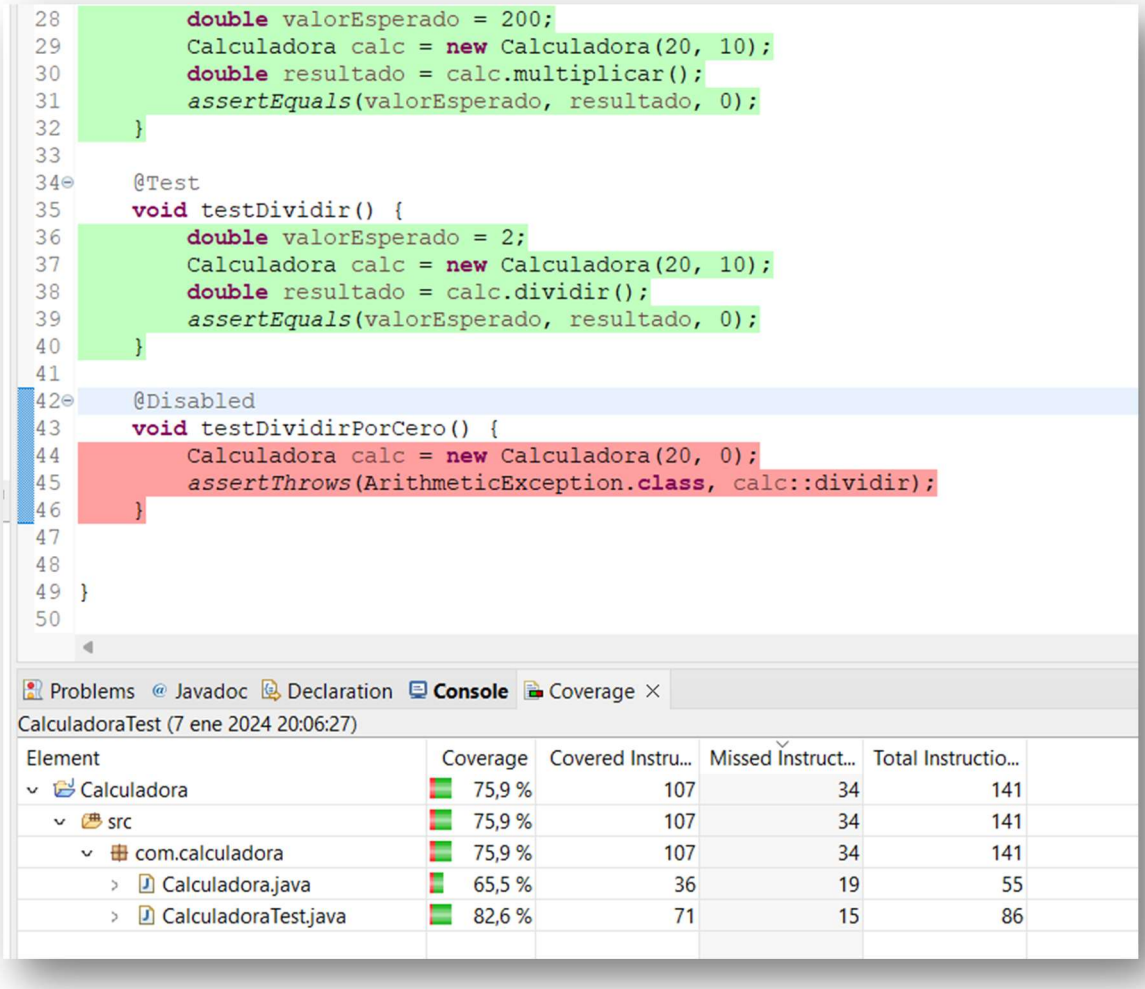
¿Tenemos la seguridad de que nuestros casos de prueba cubren todas las ramas de ejecución del programa?

Desde el entorno de programación *Eclipse* podemos visualizar la **cobertura del código** de los tests unitarios. Se puede acceder a esta herramienta desde la opción de ***Menú >> Coverage*** o desde el menú de herramientas pulsando en el botón de ***Coverage.*** 



La **cobertura de código** es el método de análisis que determina qué partes del software han sido ejecutadas por los casos de prueba y cuáles no.

Ejemplo de ejecución de ***Coverage***:



```
28          double valorEsperado = 200;
29          Calculadora calc = new Calculadora(20, 10);
30          double resultado = calc.multiplicar();
31          assertEquals(valorEsperado, resultado, 0);
32      }
33
34      @Test
35      void testDividir() {
36          double valorEsperado = 2;
37          Calculadora calc = new Calculadora(20, 10);
38          double resultado = calc.dividir();
39          assertEquals(valorEsperado, resultado, 0);
40      }
41
42      @Disabled
43      void testDividirPorCero() {
44          Calculadora calc = new Calculadora(20, 0);
45          assertThrows(ArithmeticException.class, calc::dividir);
46      }
47
48
49  }
50
```

Problems | Javadoc | Declaration | **Console** | Coverage

CalculadoraTest (7 ene 2024 20:06:27)

Element	Coverage	Covered Instru...	Missed Instruct...	Total Instructio...
Calculadora	75,9 %	107	34	141
src	75,9 %	107	34	141
com.calculadora	75,9 %	107	34	141
Calculadora.java	65,5 %	36	19	55
CalculadoraTest.java	82,6 %	71	15	86

Ahora inténtalo tú.

1) Una vez que tengas todos los tests listos y funcionando. Ejecuta esta herramienta.

2) Intenta, por otra parte, que se ejecuten todas las líneas de código de este ejemplo. Intenta conseguir un *coverage* cercano al 100%.

4.- Referencias bibliográficas

- Moreno Pérez, J.C. *Entornos de desarrollo.* Editorial Síntesis.
- Ramos Martín, A. & Ramos Martín, M.J. *Entornos de desarrollo*. Grupo editorial Garceta.