1. Pruebas unitarias con JUNIT

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 Netbeans, 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.

1.1. Enfoque recomendado para el diseño de casos

Las diferentes técnicas de caja blanca y caja negra que se han visto para desarrollar casos de prueba representan diferentes enfoques. El enfoque recomendado consiste en el uso conjunto de estas técnicas para lograr un nivel de seguridad en las pruebas "razonable". Por ejemplo:

- Crear casos de prueba de caja negra para las entradas y salidas usando clases de equivalencia tanto válido (CEV) como no válido (CEI).
- Desarrollar casos de prueba de caja blanca basados en la prueba del camino básico para complementar los casos de prueba de caja negra. En algunos casos, estas pruebas coincidirán con las clases de equivalencia del punto anterior.

La mejor manera de representar los casos de prueba será creando una tabla con las siguientes columnas:

- Número de caso de prueba (simplemente un número secuencial, 1,2,3... para luego identificarlo en Junit).
- Tipo de prueba (de las anteriores, aunque un caso puede abarcar varios tipos).
- Valores de entrada que correspondan a la prueba deseada.
- Valores de salida esperados.

1.2. Creación de un proyecto Maven para Junit

Maven es una potente herramienta de gestión de proyectos que se utiliza para gestión de dependencias (librerías), como herramienta de compilación e incluso como herramienta de documentación. Es de código abierto, gratuita y no depende del editor o IDE que uses. Un proyecto creado en NetBeans con Maven puede ser fácilmente abierto y modificado en Eclipse sin mayores cambios.

Con Maven se puede:

- Gestionar las dependencias del proyecto, para descargar e instalar módulos, paquetes y herramientas que sean necesarios para el mismo.
- Compilar el código fuente de la aplicación de manera automática.
- Empaquetar el código en archivos .jar o .zip.
- Instalar los paquetes en un repositorio (local, remoto o en el central de la empresa)
- Generar documentación a partir del código fuente.
- Gestionar las distintas fases del ciclo de vida de las build: validación, generación de código fuente, procesamiento, generación de recursos, compilación, ejecución de test ...

Además, la mayor parte de los entornos de desarrollo y editores de Java disponen de plugins específicos o soporte directo de Maven para facilitarnos el trabajo con esta, puesto que se ha convertido en una herramienta casi universal.

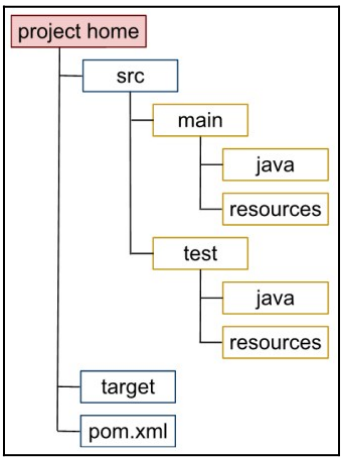
¿Qué es el archivo pom.xml?

La unidad básica de trabajo en **Maven** es el llamado Modelo de Objetos de Proyecto conocido simplemente como **POM** (de sus siglas en inglés: Project Object Model).

Se trata de un archivo XML llamado pom.xml que se encuentra por defecto en la raíz de los proyectos y que contiene toda la información del proyecto: su configuración y las librerías o bibliotecas que usarás, entre otras cosas.

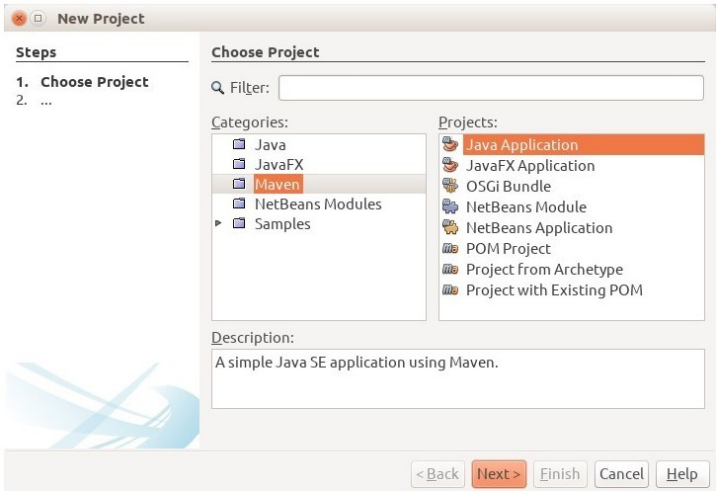
Otra característica importante de **Maven**, es que existe un repositorio central desde el cual se distribuyen una gran cantidad de librerías que podrías usar libremente. Cuando crees tu proyecto, el **IDE** que usas descargará una copia del repositorio en tu equipo y sólo agregará en tu proyecto final aquellas dependencias que necesites.

Esta sería la estructura habitual de un proyecto Java que utiliza **Maven**:

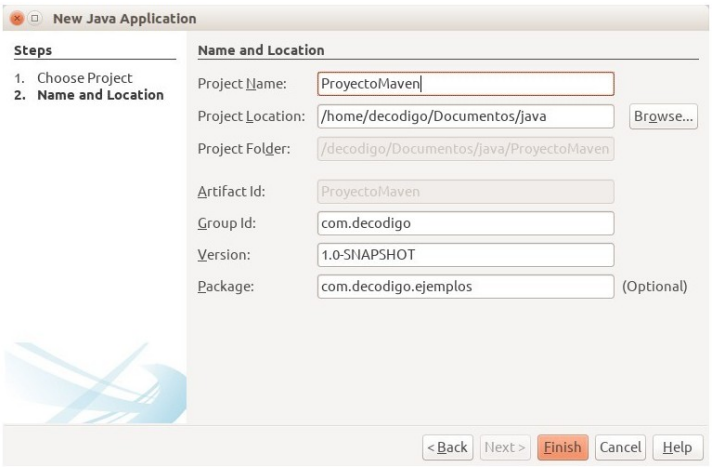


Pero veamos como crear un proyecto:

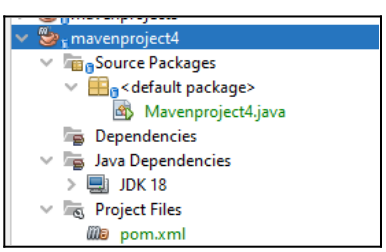
- Para crear un proyecto Maven debes dirigirte a **File>New Project** y seleccionas la categoría **Maven** y **Java Application**.



- Asignarás un nombre al proyecto, la ruta donde se creará, el grupo que desees, la versión (que puedes dejar como está) y finalmente el paquete default para tus clases que se creará cuando se construya el proyecto.



- Una vez creado el proyecto tienes una estructura básica de carpetas y un archivo **pom.xml** donde podrás continuar con la configuración.




```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/X
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>mavenproject4</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>18</maven.compiler.source>
        <maven.compiler.target>18</maven.compiler.target>
        <exec.mainClass>Mavenproject4</exec.mainClass>
    </properties>
</project>
```

- Para que resolver un bug que existe en Netbeans con la librería Junit y poder nombrar los métodos de testing sin la necesidad de que empiecen por la palabra **test** debemos agregar al **pom** del proyecto Maven la siguiente sentencia:

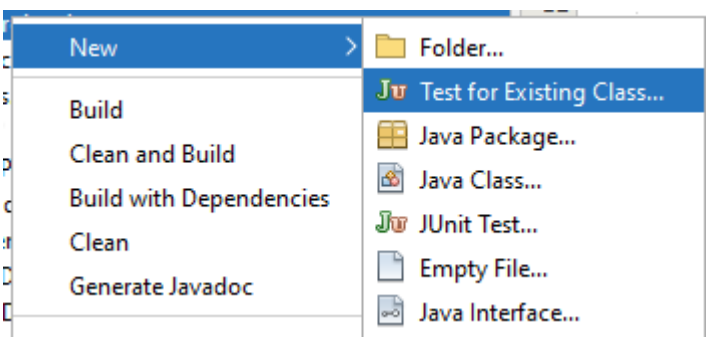
```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
    </plugins>
</build>
```

1.3. Creación de una clase de prueba

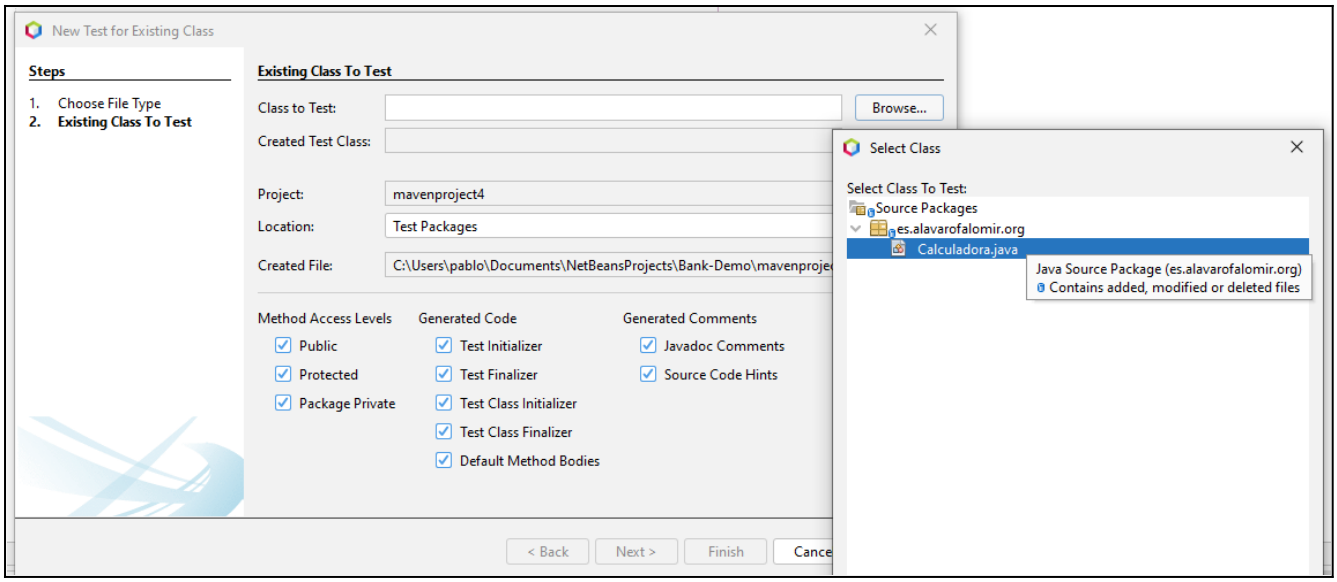
Para empezar a usar **Junit** creamos un nuevo proyecto en **Maven** en **Netbeans** y creamos la clase a probar, en este caso se llama Calculadora:

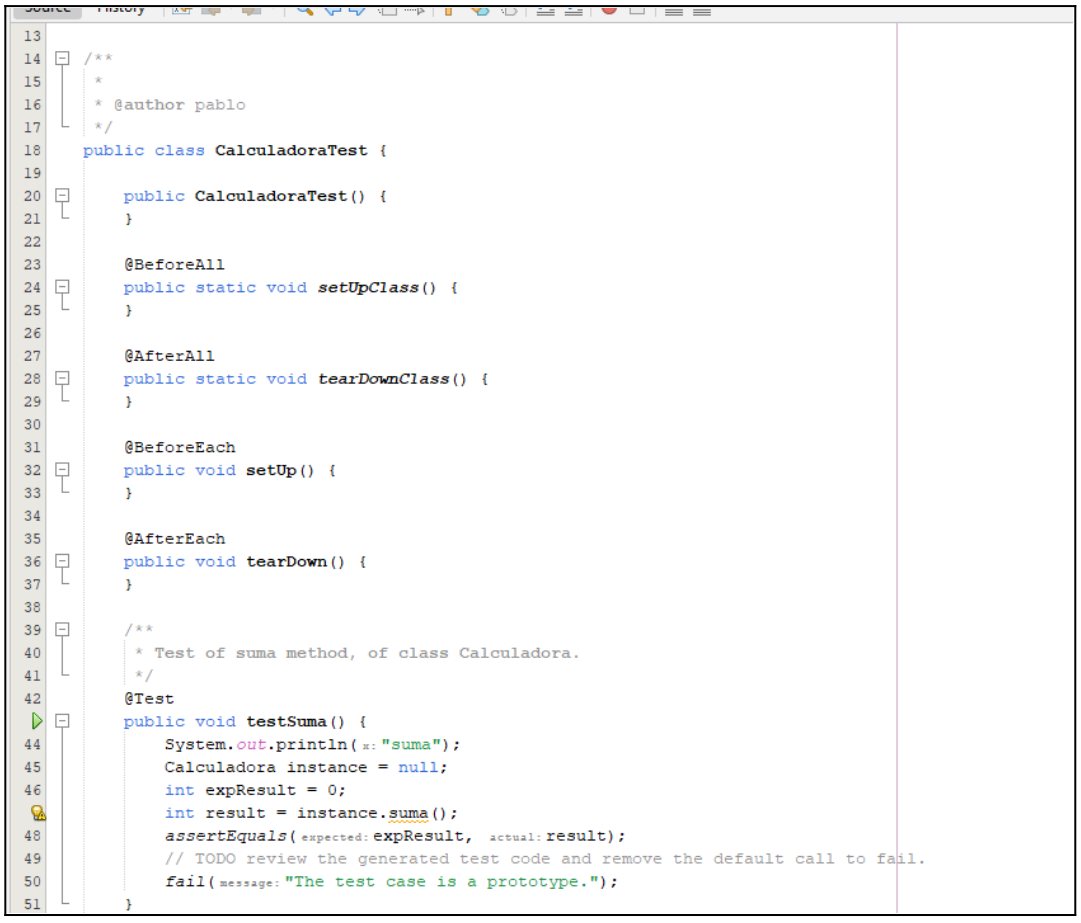
```
package calculadora;

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, para ello pulsamos el botón derecho del ratón sobre el proyecto y seleccionamos **New – Test for Existing Class...**



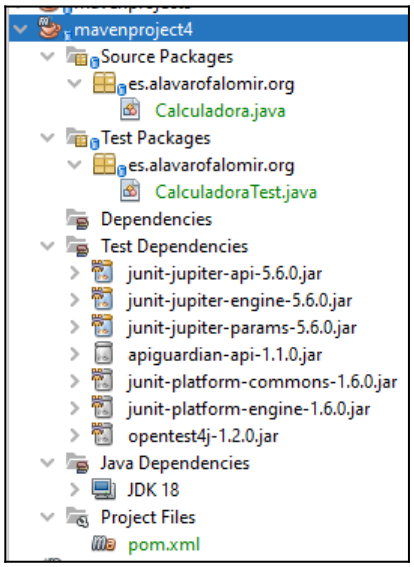
A continuación seleccionamos la clase que queremos probar y pulsamos “Finish” y Netbeans generará automáticamente la clase de prueba junto con los métodos a testear.





```java
/**
 *
 * @author pablo
 */
public class CalculadoraTest {

    public CalculadoraTest() {
    }

    @BeforeAll
    public static void setUpClass() {
    }

    @AfterAll
    public static void tearDownClass() {
    }

    @BeforeEach
    public void setUp() {
    }

    @AfterEach
    public void tearDown() {
    }

    /**
     * Test of suma method, of class Calculadora.
     */
    @Test
    public void testSuma() {
        System.out.println( x: "suma");
        Calculadora instance = null;
        int expResult = 0;
        int result = instance.suma();
        assertEquals( expected: expResult,  actual: result);
        // TODO review the generated test code and remove the default call to fail.
        fail( message: "The test case is a prototype.");
    }
```

Además de que la clase de prueba se cree automáticamente, fíjate que **Netbeans** y **Maven** han creado la estructura de directorios para realizar las pruebas y han alojado ahí la clase de prueba recién creada. También se ha modificado el fichero **pom** de **Maven** agregando automáticamente las librerías de **Junit** de modo transparente.




```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-params</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

1.4. 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. JUnit utiliza la clase Assertions para lanzar los test, que básicamente está compuesta por una serie de métodos, que una vez llamados ejecutan los métodos a probar y analizan su comportamiento comparándolos con los resultados que se espera de ellos.

Así, hay métodos que nos permiten comprobar si dos valores son o no iguales, si el valor del parámetro pasado se puede resolver como true o false, si el tiempo consumido en ejecutar un método supera el previsto, etc.

Además, los métodos están sobrecargados, permitiendo en algunos casos indicar el mensaje que ha de devolver si la comprobación no resulta exitosa, o definir un margen que valide dos números como iguales si su diferencia es inferior a dicha tolerancia ....

Método	Descripción
assertTrue(boolean expresión) **assertTrue(boolean expresión, String mensaje)**	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(boolean expresión) **assertFalse(boolean expresión, String mensaje)**	Comprueba que la expresión se evalúe a false. Si no es false y se incluye el String, al producirse error se lanzará el mensaje.
assertEquals(double valorEsperado, double valorReal, double delta) **assertEquals(double valorEsperado, double valorReal, double delta, String mensaje)** **Se puede usar con cualquier tipo de dato: Integer, short, object,…Delta se usa en tipos float y double.**	Comprueba que el valorEsperado sea igual al valorReal. Si no son iguales y se incluye el String, se lanzará el mensaje. ValorEsperado y valorReal pueden ser de diferentes tipos. 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 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.
assertNull(Object objeto), **assertNull(Object objeto, String mensaje)**	Comprueba que el objeto sea null. Si no es null y se incluye el String, se lanzará el mensaje.
assertNotNull(Object objeto), **assertNotNull(Object objeto, String mensaje)**	Comprueba que el objeto no sea null. Si es null y se incluye el String, se lanzará el mensaje.
assertSame(Object objetoEsperado, Object objetoReal),	Comprueba que objetoEsperado y objetoReal sean el mismo objeto. Si no son el mismo y se incluye el String, se lanzará el mensaje.

assertSame(Object objetoEsperado, Object objetoReal, String mensaje)	
assertNotSame(Object objetoEsperado, Object objetoReal) **assertNotSame(Object objetoEsperado, Object objetoReal, String mensaje)**	Comprueba que objetoEsperado y objetoReal no sean el mismo objeto. Si son el mismo y se incluye el String, se lanzará el mensaje.
fail(), **fail(String mensaje)**	Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el mensaje.

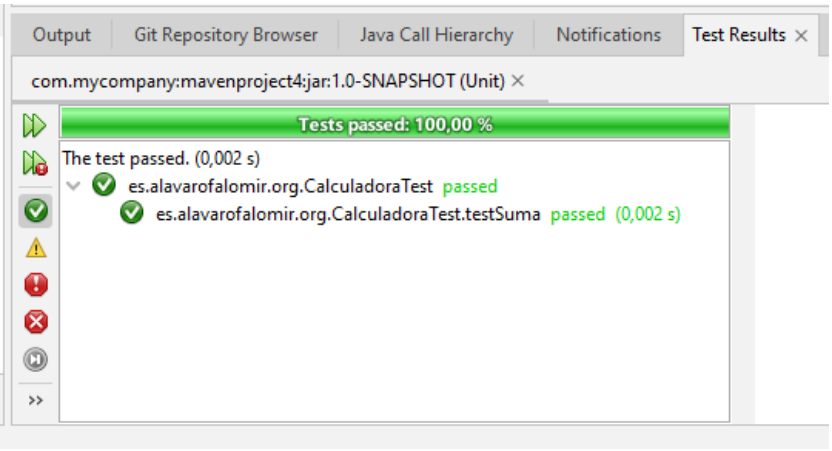
Vamos a definir los siguientes casos de prueba para probar los métodos:

Método	Entrada	Salida esperada
Suma	20, 10	30
Resta	20, 10	10
Multiplica	20, 10	200
Divide	20, 10	2

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

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

Vamos a probarlo. Sobre la clase de prueba, botón derecho del ratón – **Test File**.



Al lado de cada prueba aparece un icono con una marca: una marca de verificación verde indica prueba exitosa, un aspa azul indica fallo; y un aspa roja indica error.

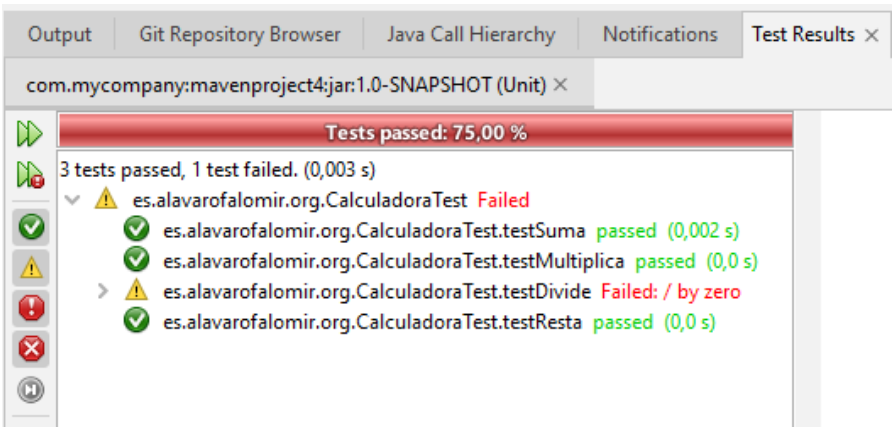
En el contexto Junit, un fallo es una comprobación que no se cumple, mientras que un error es una excepción durante la ejecución del código.

Completamos el resto de los métodos de prueba.

```
@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);
}
```

Para ver la diferencia entre un fallo y un error hacemos los siguientes cambios y ejecutamos:

```
@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);
}
```



1.5. Probar excepciones

Podemos probar casos de excepción de la siguiente manera:

```
public int divide0() {
	if(num2 == 0)
		throw new java.lang.ArithmeticException("División por 0");
	else
	 {
		int resul = num1 / num2;
		return resul;
	 }
}


@Test
public void testDivide0() {
	Calculadora calcu = new Calculadora(20,0);
	assertThrows(ArithmeticException.class,() -> calcu.divide0() );
}
```

Actividad 1

Modifica el método resta() de la clase Calculadora y añade los métodos resta2() y divide2() que se exponen a continuación. Crea después los test para probar los 3 métodos. Los métodos son:

```
public int resta() {
	int resul;
	if(resta2())
		resul = num1 - num2;
	else
		resul = num2 - num1;

	return resul;

	//		int resul = num1 - num2;
	//		return resul;
}

public boolean resta2() {
	if (num1 >= num2)
		return true;
	else
		return false;
}
public Integer divide2() {
	if(num2 == 0) return null;
	int resul = num1 / num2;
	return resul;
}
```

Utiliza los métodos asserTrue(), assertFalse(), assertNull(), assertNotNull() o assertEquals() según convenga.

Actividad 2

Escribe una clase de pruebas para probar el método calculo() de la clase Factorial. En el método se comprueba si el número es menor que 0, en este caso se lanza la excepción **IllegalArgumentException** con el mensaje "Número n no puede ser < 0".

Si el valor del factorial calculado es menor que 0 es que ha ocurrido un error de desbordamiento, en este caso se lanza la excepción ArithmeticException y se lanza el mensaje "Overflow, número n demasiado grande".

La clase a probar es la siguiente:

```
public class Factorial {
	public static int calculo(int n) {
		if (n < 0) {
			throw new IllegalArgumentException("Número"+ n + " no
puede ser < 0");
		}
		int fact = 1;
		for (int i=2;i<=n;i++)
			fact *= i;

		if (fact<0) {
			throw new ArithmeticException("Overflow, número "+ n + "
demasiado grande");
		}
		return fact;

	}
}
```

1.6. Tipos de anotaciones

En todos los métodos de prueba anteriores se repetía la línea Calculadora calcu=new Calculadora(20,10);. Esta sentencia de inicialización se puede escribir una sola vez dentro de la clase.

Las anotaciones proporcionan información sobre un programa y pueden ser utilizadas por el compilador, por herramientas de software que pueden procesarlos (por ejemplo, para generar código o archivos xml) o para ser procesados en tiempo de ejecución. Las anotaciones se pueden aplicar a declaraciones de clases, campos, métodos y otros elementos de un programa.

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

Etiqueta @Test

Indica que el siguiente método será llamado para ser probado.	`@Test`
	`void testDevuelveTrue() {`
	`  System.out.println("Llamando a testDevuelveTrue");`
	`}`

Etiqueta RepeatedTest

Indica que el siguiente método será llamado las veces que la etiqueta recibe como parámetro. Dos en este caso.	`@RepeatedTest(2)`
	`void testRepiteTest() {`
	`  System.out.println("Llamando a testRepiteTest");`
	`}`

Etiqueta @BeforeAll

Indica que el siguiente método es llamado una sola vez en toda la ejecución del programa. A continuación son llamadas las pruebas etiquetadas con @Test, @RepeatedTest, @ParameterizedTest o @TestFactory.	`@BeforeAll`
	`static void MetodoBeforeAll() {`
	`  System.out.println("Llamando a MetodoBeforeAll");`
	`}`

Etiqueta @AfterAll

Indica que el siguiente método es llamado una sola vez en toda la ejecución del programa. Antes habrán sido llamadas todas las pruebas etiquetadas con @Test, @RepeatedTest, @ParameterizedTest o @TestFactory.	`@AfterAll`
	`static void MetodoAfterAll() {`
	`  System.out.println("Llamando a MetodoAfterAll");`
	`}`

Etiqueta @BeforeEach

Indica que el siguiente método es llamado antes de ejecutar cada una de las pruebas etiquetadas con @Test, @RepeatedTest, @ParameterizedTest o @TestFactory.	`@BeforeEach`
	`void MetodoBeforeEach() {`
	`  System.out.println("Llamando a MetodoBeforeEach");`
	`}`

Etiqueta @AfterEach

Indica que el siguiente método es llamado después de ejecutar cada una de las pruebas etiquetadas con @Test, @RepeatedTest, @ParameterizedTest o @TestFactory.	`@AfterEach`
	`void MetodoAfterEach() {`
	`  System.out.println("Llamando a MetodoAfterEach");`
	`}`

Etiquetas @ParameterizedTest y @ValueSource

Permite pasar al método de prueba parámetros a utilizar en la ejecución. Se lanza una vez por parámetro pasado.	`@ParameterizedTest`
	`@ValueSource(strings = {"HOLA","ADIOS"})`
	`void testParameterizedTest(String sMiImput) {`
	`  System.out.println("Llamando a testParameterizedTest " +`
	`  sMiImput );`
	`}`

Etiquetas @ParameterizedTest y @CsvSource

@ValueSource sólo permite el paso de un parámetro al método de prueba. Si necesitamos pasar varios se utiliza la etiqueta @CsvSource. Se lanza una vez por cada tupla de parámetros pasada.	`@ParameterizedTest`
	`@CsvSource({"HOLA,1","ADIOS,2"})`
	`void testParameterizedIntTest(String a, int b) {`
	`  System.out.println("Llamando a testParameterizedIntTest : "`
	`  + a  + "--- " + b);`
	`}`

@Disabled

Desactiva la llamada a un método de prueba.	`@Test`
	`@Disabled`
	`void testDesactivado() {`
	`  System.out.println("Desactivada la llamada a testDesactivado");`
	`}`

@BeforeEach: si anotamos un método con esta etiqueta, el código **será ejecutado antes de cualquier método de prueba**. Este método se puede utilizar para inicializar datos. Por ejemplo, en una aplicación de acceso a base de datos, si vamos a usar un array para las pruebas, se puede inicializar aquí.

Puede haber varios métodos en la clase de prueba con esta anotación.

@AfterEach: si anotamos un método con esta etiqueta, el código **será ejecutado después de la ejecución de cada método de prueba**. Se puede utilizar para limpiar datos. Puede haber varios métodos en la clase de prueba con esta anotación.

La clase **CalculadoraTest** incluyendo dos métodos con las anotaciones **@BeforeEach** y **@AfterEach** quedaría de la siguiente manera. En el primer método **creaCalculadora()** inicializamos el objeto calculadora y en el segundo **borraCalculadora()** lo limpiamos:

```
package calculadora;

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

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class CalculadoraTest2 {
	private Calculadora calcu;

	@BeforeEach
	public void creaCalculadora() {
		calcu = new Calculadora(20,10);
	}
	@AfterEach
	public void borraCalculadora() {
		calcu = null;
	}
	@Test
	void testSuma() {
		double valorEsperado = 30;
		double resultado = calcu.suma();
		assertEquals(valorEsperado,resultado,0);
	}

	@Test
	void testResta() {
		double valorEsperado = 10;
		double resultado = calcu.resta();
		assertEquals(valorEsperado,resultado,0);
	}
```

```
	@Test
	void testMultiplica() {
		double valorEsperado = 200;
		double resultado = calcu.multiplica();

		assertEquals(valorEsperado,resultado,0,"Fallo en la
multiplicación: ");
	}

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

}
```

Otras anotaciones a destacar son @BeforeAll y @AfterAll. Estas tienen algunas diferencias con respecto a las anteriores:

@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.

@AfterAll: solo puede haber un método con esta anotación. Este método será invocado una sola vez cuando finalicen todas las pruebas.

En este caso los métodos anotados con **@BeforeAll y @AfterAll** deben ser **static**, y por tanto los atributos a los que acceden también. Los métodos añadidos anteriormente quedarían así:

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

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

class CalculadoraTest3 {
	private static Calculadora calcu;
	private double resultado;

	@BeforeAll
	public static void creaCalculadora() {
		calcu = new Calculadora(20,10);
	}
	@AfterAll
	public static void borraCalculadora() {
		calcu = null;

...
```

Actividad 3

Escribe una clase de pruebas para probar los métodos de la clase TablaEnteros. En esta clase de prueba, crea un método con la anotación **@BeforeAll** en el que inicialices un array de enteros para usarlo en las pruebas de los métodos.

El método sumaTabla() suma los elementos del array y devuelve la suma.

El método mayorTabla() devuelve el elemento mayor de la tabla.

Y el método posicionTabla() devuleve la posición ocupada por el elemento cuyo valor se envía.

En el constructor se comprueba si el número de elementos de la tabla es nulo o 0, en este caso se lanza la excepción IllegalArgumentException con el mensaje “No hay elementos”.

El método posicionTabla también lanza la excepción, java.util.NoSuchElementException, en el caso de que no se encuentre el elemento en la tabla. Hay que añadir otros dos métodos de prueba para probar estas excepciones. La clase a probar es la siguiente:

```
package calculadora;

public class TablaEnteros {
	private Integer[] tabla;

	TablaEnteros(Integer[] tabla){
		if (tabla == null || tabla.length == 0)
			throw new IllegalArgumentException("No hay elementos");
		this.tabla = tabla;

	}

	//Devuelve la suma de los elementos de la tabla
	public int sumaTabla() {
		int suma = 0;
		for (int i = 0; i < tabla.length; i++) {
			suma += tabla[i];
		}
		return suma;
	}

	//Devuelve el mayor elemento de la tabla
	public int mayorTabla() {
		int max = -999;
		for (int i = 0; i < tabla.length; i++)
			if (tabla[i] > max)
				max = tabla[i];
		return max;
	}
```

```
	//Devuelve la posición de un elemento cuyo valor se pasa
	public int posicionTabla(int n) {
		for (int i = 0; i < tabla.length; i++)
			if (tabla[i] == n)
				return i;
		throw new java.util.NoSuchElementException("No existe: "+n);
	}

}
```

1.7. Pruebas parametrizadas

1.7.1 ValueSource

Las pruebas parametrizadas permiten ejecutar varias veces una prueba con diferentes valores en sus argumentos. Se declaran utilizando la anotación **@ParameterizedTest** en lugar de la anotación **@Test**. Además, se debe declarar al menos una fuente que proporcionará los argumentos para cada invocación y luego consumir los argumentos en el método de prueba.

Si nuestro método de prueba necesita que le pasemos solo un parámetro de tipo String o cualquier tipo primitivo de dato, usaremos la anotación **@ValueSource**. Esta anotación nos permite especificar una única matriz de valores literales y solo puede usarse para proporcionar un único argumento al método de prueba.

El siguiente ejemplo muestra una prueba parametrizada que utiliza la anotación **@ValueSource** para probar el método **mensajeNoNulo(String cadena)**, el argumento para el método lo tomará del array de cadenas, realizándose tantas pruebas como cadenas hay en el array. En el ejemplo se realizarán dos pruebas del método, una prueba con el valor “Hola” y otra con el valor “Mundo”:

```
@ParameterizedTest
@ValueSource(strings = {“Hola” “Mundo”})
void mensajeNoNulo(String cadena){
	assertNotNull(cadena);
}
```

Si queremos hacer un método de prueba que reciba un valor entero tenemos que especificar un array de enteros, para ello usaremos el atributo **ints** en lugar de **strings** de la siguiente manera:

```
@ValueSource(ints = {10,20,9})
```

Únicamente podemos hacer uso de los siguientes tipos de datos:

- short
- byte
- int
- long
- float
- double
- char
- java.lang.String
- java.lang.Class

Para datos de tipo primitivo **double** usaremos el atributo **doubles**, para los de tipo **float** usaremos **floats**, y así con el resto de tipos de datos primitivos.

1.7.2 CsvSource

Para proporcionar a un método de prueba varios parámetros tenemos que utilizar la anotación **@CsvSource**. Cuando agregamos esta anotación, tenemos que configurar los datos de prueba usando una matriz de objetos String. Al especificar los datos de prueba, debemos seguir estas reglas:

- Un objeto String debe contener todos los parámetros para una invocación del método.
- Los diferentes parámetros deben separarse con una coma.
- Los valores encontrados en cada línea deben seguir el mismo orden que los parámetros definidos en el método de prueba.

En los siguientes ejemplos se prueba el método **divide**() de la clase **Calculadora**. En el primer ejemplo configuramos los parámetros que se pasan al método de prueba **testDivide0(int,int,int)**, que acepta tres parámetros int, el valor del tercer parámetro especifica el resultado de la división esperada de los dos valores anteriores, por ejemplo (20,10 y 2).

Los valores para cada caso de prueba se definen entre llaves y encerrados entre comillas dobles y los parámetros separados por comas. Estos se asignarán a los parámetros del método en el orden en que están definidos: a=20, b=10 y valorEsperado=2:

```
@ParameterizedTest
@CsvSource({“20, 10, 2”})
public void testDivide0 (int a, int b, valorEsperado){

        Calculadora calcu=new Calculadora(a,b);
        int resultado = calcu.divide();
        assertEquals(valorEsperado, resultado);
}
```

En el segundo método de prueba testDivide1(int,int,int), se proporcionan tres casos de prueba, es decir, tres objetos String; cada prueba proporcionará los parámetros necesarios al método:

```
@ParameterizedTest
@CsvSource({“20, 10,2”, “30, -2,-15”, “5, 2,3” })
public void testDivide1(int a, int b, int valorEsperado){
        Calculadora calcu=new Calculadora(a,b);
        int resultado = calcu.divide();
        assertEquals(valorEsperado, resultado);
}
```

Comprobando la salida de la ejecución de la prueba, se observa que debajo de cada método de prueba se muestra entre corchetes la prueba de que se trata. Por ejemplo, dentro de testDivide0() → [1] 20,10,2 (0,021 s) prueba del grupo de valores {20,10,2}, dentro de testDivide1() → [3] 5,2,3 (0,024 s) prueba el grupo {5,2,3}.

1.7.3 MethodSource

Otra de las funcionalidades nuevas que han venido con JUnit 5 y los parametized Test es la posibilidad de pasar métodos para verificar nuestros test. Ya que los ejemplos que hemos visto hasta ahora únicamente nos permiten pasar valores simples, vamos a ver como haciendo uso de @**MethodSource** podemos invocar a un método para devolver argumentos.
Este metodo de realizar tests parametrizados es muy útil si tenemos la necesidad de pasar a nuestros tests objetos de tipo array o cualquier otro

Los valores que puede devolver el método que se invoca son los siguientes:

- Stream
- Iterable
- Iterator
- Array de argumentos

Vamos a ver un ejemplo. Imagina que tenemos una clase para obtener el día de la semana asociado a un número y queremos analizar que la relación es correcta:

```
@ParameterizedTest
@MethodSource("numeroDia")
void numberToDay(int day, String name) {
        assertEquals(name, DayOfWeek.of(day));
}

private static Stream<Arguments> numeroDia() {
        return Stream.of(
                arguments(1, "Sunday"),
                arguments(2, "Monday"),
                arguments(3, "Tuesday")
                arguments(4, "Wednesday"),
                arguments(5, "Thursday"),
                arguments(6, "Friday"),
                arguments(7, "Saturday")
        );
}
```

Ejemplo de como se parametrizan tests en Junit en los cuales esten involucrados objetos tipo array.

```
@Test
@ParameterizedTest
@MethodSource("generaArrayEntradaIntEsperado")
public void testArray(int a[], int length) {

        System.out.println("Test longitud array.");
        assertEquals(length, instance.arrayLength(a));

}

private static Stream<Arguments> generaArrayEntradaIntEsperado() {
   return Stream.of(
         arguments( new int[]{13, 14, 65, 456, 31, 83}, 6),
         arguments( new int[]{13, 14, 65, 456}, 4)
);
```

Actividad 4

Realiza pruebas parametrizadas para los métodos suma(), resta(), multiplica() de la clase Calculadora.

Actividad 5

Realiza pruebas parametrizadas para las clases Factorial de la Actividad 2. Una clase de prueba para cada excepción y otra para el cálculo correcto.

Actividad 6

Desarrolla una batería de pruebas para probar el método DevuelveFecha() de la clase Fecha que se expone a continuación. El método recibe un número entero y devuelve un String con un formato de fecha que dependerá del valor de dicho número. Si el número recibido es distinto de 1, 2 y 3 el método devuelve "ERROR". La clase es la siguiente:

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class Fecha {
	SimpleDateFormat formato;
	Date hoy;

	public Fecha() {
		hoy = new Date();
	}
	public String DevuelveFecha(int tipo) {
		String cad = "";
		switch (tipo) {
		case 1: {
			formato = new SimpleDateFormat("yyyy/MM");
			cad = formato.format(hoy);
			break;
		}
		case 2: {
			formato = new SimpleDateFormat("MM/yyyy");
			cad = formato.format(hoy);
			break;
		}
		case 3: {
			formato = new SimpleDateFormat("MM/yy");
			cad = formato.format(hoy);
			break;
		}
		default:
			cad = "ERROR";
		}
		return cad;
	}
}
```