JUnit: Introducción a las Pruebas Unitarias

Las **pruebas unitarias** juegan un papel fundamental en el desarrollo de software, ya que permiten verificar el comportamiento esperado de los componentes individuales del sistema, como métodos o clases. Su principal ventaja es que ayudan a identificar errores de manera temprana, asegurando que cada pieza de código funcione correctamente de forma aislada.

¿Qué es JUnit?

JUnit es un framework ampliamente utilizado para realizar pruebas unitarias en el ecosistema Java. Este marco facilita la escritura y ejecución de pruebas, lo que ayuda a los desarrolladores a verificar que su código esté funcionando según lo esperado. JUnit también proporciona un conjunto de herramientas que generan informes detallados sobre los resultados de las pruebas, permitiendo una rápida retroalimentación.

Estructura Básica de una Clase de Prueba con JUnit

A continuación, te mostramos un ejemplo básico de cómo estructurar una clase de prueba en JUnit. En este ejemplo, se utiliza la anotación `@Test` para marcar los métodos que deben ser ejecutados como pruebas unitarias.

```java
package test;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import src.Ejercicio1;

public class Ejercicio1Test {
    @Test
    public void testMetodo() {
        Integer resultado = Ejercicio1.metodo(2);
        Assertions.assertEquals(4, resultado);
    }
}
```

En este caso:

- **Clase de prueba**: `Ejercicio1Test` es la clase donde se realizan las pruebas de la clase `Ejercicio1`.
- **Método de prueba**: `testMetodo` utiliza la anotación `@Test` para indicar que debe ejecutarse como una prueba. Dentro del método, se invoca el método `metodo()` de la clase `Ejercicio1` y luego se valida su resultado con `Assertions.assertEquals()`.

Convenciones de Nombres en JUnit

Las convenciones de nomenclatura son cruciales para mantener el código organizado y legible. En el contexto de las pruebas unitarias, existen pautas específicas para nombrar las clases y métodos de prueba.

Convención de nomenclatura para clases

Convención de Nomenclatura para Clases	Detalles
Las clases de prueba deben seguir una convención clara que relacione la clase de prueba con la clase que está siendo probada. El nombre de la clase de	Ejemplo: "Calculadora" se convierte en "CalculadoraTest".

prueba debe ser el mismo que el de la clase original, pero añadiendo el sufijo `Test`.	

```
package test;
public class CalculadoraTest {
    // pruebas unitarias para Calculadora
}
```

Convención de nomenclatura para métodos

Los nombres de los métodos de prueba deben describir claramente el comportamiento que se está verificando. Existen diferentes estilos para nombrarlos, dependiendo de la complejidad de la prueba.

Convención de Nomenclatura para Métodos	Detalles
testMethod Name (Enfoque Antigüo) :	Los métodos comienzan con `test` seguido de la funcionalidad probada.

```
@Test
void testSuma() {
// código de prueba aquí...
}
```

should_MethodName_ExpectedBehavior_GivenCondition (Enfoque descriptivo)	Utiliza un formato más explícito, como `should_MethodName_ExpectedBehavior_GivenCondition`.

```
@Test
void should_Suma_ReturnCorrectSum_GivenMultipleNumberPairs() {
// código de prueba aquí...
}
```

MethodName_GivenCondition_ExpectedBehavior (Enfoque Given-When-Then):	-Esta convención describe las condiciones iniciales, la acción y el resultado esperado.

```
@Test
void suma_GivenMultipleNumberPairs_ReturnsCorrectSum() {
```

```
    // código de prueba aquí...
}
```

MethodName_Scenario_Expected Result (Enfoque compacto):	Utiliza un formato más sencillo, ideal para pruebas con pocos escenarios.

```
@Test
void suma_MultipleNumberPairs_CorrectSum() {
    // código de prueba aquí...
}
```

Anotación `@DisplayName`

A partir de JUnit 5, puedes usar la anotación **@DisplayName** para darle un nombre más legible y descriptivo a tu prueba, lo cual es útil para hacer que los resultados de las pruebas sean más comprensibles.

```
@Test
@DisplayName("Test del método suma() con múltiples pares de números: Debería retornar la suma correcta")
void testSuma() {
    // código de prueba aquí...
}
```

Assertions

Las assertions son declaraciones que verifican si cierta condición es verdadera. Son esenciales en las pruebas unitarias, ya que nos permiten garantizar que nuestro código funcione correctamente. JUnit 5 proporciona la clase **`org.junit.jupiter.api.Assertions`**, que incluye una variedad de métodos estáticos para realizar diferentes tipos de assertions:

Método	Descripción	Ejemplo de Uso

`assertEquals(expected, actual)`	Verifica si dos valores son iguales. Si no lo son, la prueba fallará.	`@Test`
		`void testSuma() {`
		`  // La suma debería ser 5`
		`  assertEquals(5, 2 + 3);`
		`}`

`assertNotEquals(expected, actual)`	Verifica si dos valores NO son iguales. Si lo son, la prueba fallará.	`@Test`
		`void testSuma() {`
		`  // La suma no debería ser 6`
		`  assertNotEquals(6, 2 + 3);`
		`}`

`assertTrue(condition)`	Verifica si una condición es verdadera. Si no lo es, la prueba fallará.	`@Test`
		`void testIsEven() {`
		`  // 4 debería ser par`
		`  assertTrue(4 % 2 == 0);`
		`}`

`assertFalse(condition)`	Verifica si una condición es falsa. Si no lo es, la prueba fallará.	`@Test`
		`void testIsOdd() {`
		`  // 4 no debería ser impar`
		`  assertFalse(4 % 2 != 0);`
		`}`

`assertNull(value)`	Verifica si un valor es nulo. Si no lo es, la prueba fallará.	`@Test`
		`void testNullValue() {`
		`  String str = null;`
		`  // La variable debería ser nula`
		`  assertNull(str);`
		`}`

`assertNotNull(value)`	Verifica si un valor NO es nulo. Si lo es, la prueba fallará.	`@Test`
		`void testNotNullValue() {`
		`  String str = "Hola mundo";`
		`  // La variable no debería ser nula`
		`  assertNotNull(str);`

		`}`

`assertSame(expected, actual)`	**Verifica si dos referencias de objetos apuntan al mismo objeto. Si no lo hacen, la prueba fallará.**	`@Test`
		`void testSameObject() {`
		`  String str1 = "Hola mundo";`
		`  String str2 = str1;`
		`  // Las variables deberían referenciar al mismo objeto`
		`  assertSame(str1, str2);`
		`}`

`assertNotSame(expected, actual)`	**Verifica si dos referencias de objetos NO apuntan al mismo objeto. Si lo hacen, la prueba fallará.**	`@Test`
		`void testNotSameObject() {`
		`  String str1 = new String("Hola mundo");`
		`  String str2 = new String("Hola mundo");`
		`  // Las variables no deberían referenciar al mismo objeto`
		`  assertNotSame(str1, str2);`
		`}`

`assertArrayEquals(expectedArray, actualArray)`	**Verifica si dos arrays son iguales. Si no lo son, la prueba fallará.**	`@Test`
		`void testArrayEquality() {`
		`  int[] array1 = {1, 2, 3};`
		`  int[] array2 = {1, 2, 3};`
		`  // Los arrays deberían ser iguales`
		`  assertArrayEquals(array1, array2);`
		`}`

`assertThrows(expectedType, executable)`	**Verifica si una operación lanza una excepción del tipo esperado.**	`@Test`
		`void testException() {`
		`  // Debería lanzar ArithmeticException`
		`  assertThrows(ArithmeticException.class, () -> {`
		`  int division = 5 / 0;`
		`  });`
		`}`

`assertEquals(double expected, double actual, double delta)`	**Compara números de punto flotante permitiendo una diferencia de precisión.**	`@Test`
		`public void testSquareRoot() {`
		`// El valor de la raíz cuadrada de 4 debería ser 2`
		`assertEquals(2.0, Math.sqrt(4.0));`
		`// La raíz cuadrada de 2 debería ser cercana a 1.4142`
		`assertEquals(1.4142, Math.sqrt(2.0), 0.0001);`
		`}`

Todos los métodos mencionados anteriormente **también tienen una sobrecarga que acepta un parámetro adicional de tipo String.** Este parámetro nos permite agregar un mensaje personalizado que se imprimirá en caso de que la assertion falle. El mensaje personalizado puede ser útil para proporcionar contexto y comprender por qué la prueba ha fallado.

```
@Test
void testSuma() {
    assertEquals(5, 2 + 3, "La suma debería ser 5");}
```

Estrategia triple A (Arrange, Act, Assert)

La estrategia **AAA** (Arrange, Act, Assert) es un patrón utilizado para organizar las pruebas unitarias en tres fases claras:

1. **Arrange (Organizar)**: Configuración del entorno de prueba, como la creación de objetos y la definición de las condiciones iniciales.
2. **Act (Actuar)**: Ejecución de la acción que se va a probar (por ejemplo, llamar a un método).
3. **Assert (Afirmar)**: Verificación de que el resultado es el esperado.

Ejemplo:

```
public class CalculadoraTest {
    @Test
    void testSuma() {
        // Arrange
        int numero1 = 4;
        int numero2 = 5;
        // Act
        int resultado = Calculadora.suma(numero1, numero2);
        // Assert
        assertEquals(9, resultado, "La suma de 4 y 5 debería ser 9"); }
}
```

Ciclo de vida de las pruebas unitarias

JUnit permite controlar el ciclo de vida de las pruebas mediante anotaciones que se ejecutan antes y después de cada prueba, o incluso antes y después de todas las pruebas en una clase.

Anotación	Descripción	Ejemplo de Uso
`@BeforeAll`	**Método estático que se ejecuta una vez antes de todos los métodos de prueba en la clase de prueba.**	`class MyTestClass { @BeforeAll static void initAll() { // Código para configurar el estado antes de todas las pruebas ... } }`
`@BeforeEach`	**Método no estático que se ejecuta antes de cada método de prueba individual en la clase de prueba.**	`class MyTestClass { @BeforeEach void setUp() { // Código para configurar antes de cada prueba ... } }`
`@Test`	**Anotación que se aplica a cada método de prueba individual.**	`class MyTestClass { @Test void myTest() { // Código de la prueba ... } }`

		`}`
`@AfterEach`	**Método no estático que se ejecuta después de cada método de prueba individual en la clase de prueba.**	`class MyTestClass { @AfterEach void tearDown() { // Código para limpiar el estado después de cada prueba... } }`
`@AfterAll`	**Método estático que se ejecuta una vez después de todos los métodos de prueba en la clase de prueba.**	`class MyTestClass { @AfterAll static void tearDownAll() { // Código para limpiar el estado después de todas las pruebas aquí } }`

Estas anotaciones permiten configurar y limpiar el estado de las pruebas, asegurando que no haya interferencias entre pruebas.

💡**DOCUMENTACIÓN OFICIAL:**

Existen más formas de realizar pruebas parametrizadas que involucran el uso de objetos, las cuales se pueden explorar en la documentación de JUnit 5 en el apartado de pruebas parametrizadas: **Documentación de JUnit 5**