Material descargable

Teoría **JAVA VII**

¡Te damos la bienvenida! 👋

Hoy vamos a adentrarnos en el emocionante mundo de las **pruebas unitarias** en Java y descubrir una herramienta fundamental: **JUnit**. Aprenderemos cómo estructurar nuestras pruebas con JUnit y seguir las **convenciones de nomenclatura** adecuadas. Además, exploraremos las **aserciones** y la **estrategia triple A** para escribir pruebas claras y comprensibles.

También abordaremos el **ciclo de vida de las pruebas unitarias**, que nos permitirá personalizar las acciones antes y después de cada prueba. Descubriremos la **repetición de pruebas** y las **pruebas parametrizadas**, herramientas que nos ayudarán a probar nuestro código en diferentes escenarios.

¡Empecemos! 🚀

Pruebas unitarias

Las **pruebas unitarias (o "testing unitario") desempeñan un papel fundamental en la validación del software al verificar el funcionamiento preciso de los componentes individuales**.

En estas pruebas, se examina la funcionalidad de métodos o clases de manera aislada, sin tener en cuenta sus dependencias (que se abordarán más adelante en el curso).

De esta manera, se garantiza que cada componente trabaje correctamente y cumpla con sus objetivos específicos.

JUnit: Un marco de pruebas para Java

JUnit es un framework (marco de trabajo) popular para realizar pruebas unitarias en Java. Es simple, eficaz y ampliamente utilizado en la comunidad de desarrolladores Java.

Con JUnit, **puedes fácilmente escribir pruebas que verifiquen el comportamiento de tu código** y luego ejecutar esas pruebas para obtener informes detallados de los resultados.

Estructura de una clase Test con JUnit

Al utilizar JUnit para realizar pruebas unitarias, es necesario importar las clases y anotaciones correspondientes. A continuación, te compartimos un ejemplo de cómo se estructura una clase de prueba en JUnit:

```
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 ejemplo, la clase de prueba se asemeja a una clase Java normal y no contiene un método main. *Observamos el uso de la **anotación @Test**, que indica a JUnit que el método es una prueba unitaria y debe ser ejecutado como tal por el marco de trabajo ("framework").*

En la prueba unitaria, se invoca el método *"metodo()"* de la clase Ejercicio1 que se desea probar. Luego, mediante la clase *"Assertions"* (de la cual hablaremos más adelante), se verifica si el resultado es igual al valor esperado.

Si el método *"assertEquals"* devuelve true, JUnit informará a la extensión de VS Code que la prueba se completó correctamente. En caso de que *"assertEquals"* devuelva false, JUnit indicará a la extensión de VS Code que la prueba ha fallado y proporcionará un mensaje detallado que muestra la diferencia entre el resultado esperado y el resultado actual.

Convención de nomenclatura para nombres de clases y métodos

Convención de nomenclatura para clases

La convención generalmente utilizada para nombrar clases de prueba es agregar la palabra *"Test"* al final del nombre de la clase que se está probando. Por ejemplo, si tenemos una clase llamada *"Calculadora"*, la clase de prueba correspondiente se llamaría *"CalculadoraTest"*.

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

Esta convención establece una relación clara entre la clase que está siendo probada y la clase de prueba, lo que facilita la búsqueda de las pruebas asociadas a una clase específica.

Convención de nomenclatura para métodos

Existen diferentes convenciones de nomenclatura para nombrar los métodos de prueba en Java. Aquí se presentan algunas opciones:

- **testMethodName**: Esta es la convención más antigua pero menos descriptiva. Aunque es menos preferible, todavía se utiliza en algunos casos. El nombre del método comienza con "test" seguido del nombre de la funcionalidad que se está probando.

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

- **should_MethodName_ExpectedBehavior_GivenCondition:** Esta convención es descriptiva y clara. El nombre del método sigue un formato *should-when-given*, describiendo el comportamiento esperado dada una condición específica.

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

- **MethodName_GivenCondition_ExpectedBehavior:** Esta convención coloca la condición antes del comportamiento esperado. Algunas personas prefieren este enfoque, ya que se lee como una sentencia de causa y efecto.

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

- **given_Precondition_When_StateUnderTest_Then_ExpectedBehavior:** Esta convención de nombres sigue el enfoque Given-When-Then del desarrollo guiado por el comportamiento. Es muy descriptiva, pero puede resultar en nombres de métodos largos.

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

- **MethodName_Scenario_ExpectedResult:** Esta convención es útil cuando se prueban diferentes escenarios. Proporciona un formato claro que destaca lo que se está probando, en qué condiciones y qué resultado se espera. Es concisa y sigue siendo descriptiva.

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

```
}
```

DisplayName

Desde JUnit 5, puedes usar la anotación **@DisplayName** para asignar un nombre más descriptivo y en formato de texto libre a tu prueba. Esto puede ser útil para describir la intención de la prueba de una manera más legible y además reemplaza el nombre del método en la publicación del resultado.

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

Aserciones

Las aserciones ("assertions") o afirmaciones son declaraciones que verifican si cierta condición es verdadera. Son fundamentales en las pruebas unitarias, ya que nos permiten asegurarnos de que nuestro código funcione correctamente.

JUnit 5 proporciona la clase *"org.junit.jupiter.api.Assertions"* con una variedad de métodos estáticos para realizar diferentes tipos de afirmaciones:

- **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.

```java
@Test
void testException() {
    // Debería lanzar ArithmeticException
    assertThrows(ArithmeticException.class, () -> {
        int division = 5 / 0;
    });
}
```

❗ *El parámetro "executable" en el ejemplo puede construirse utilizando una **expresión lambda.** En futuras clases, abordaremos qué son las expresiones lambda y cómo se utilizan.*

- **assertEquals(double expected, double actual, double delta):** Se utiliza cuando se comparan números de punto flotante y se desea permitir un margen de error debido a la limitada precisión de estos números. Puedes utilizar float en lugar de double si es necesario.

```java
@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);
}
```

En el segundo caso, se permite una diferencia de hasta 0.0001 entre el valor esperado y el valor actual, lo que tiene en cuenta la precisión limitada de los números de punto flotante.

Mensaje personalizado de error

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 afirmación falle. El mensaje personalizado puede ser útil para proporcionar contexto y comprender por qué la prueba ha fallado.

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

```
}
```

Estrategia triple A

La **estrategia AAA (Arrange, Act, Assert)** es un patrón ampliamente utilizado *para organizar y escribir pruebas unitarias*. Estas tres fases representan los pasos principales en la realización de una prueba unitaria:

- **Arrange (Organizar):** En esta fase, se configura el entorno de prueba. Se crean los objetos e instancias necesarios y se establecen los estados iniciales. También, en esta etapa, se pueden configurar objetos simulados/mocks (los cuales abordaremos en próximas clases)..
- **Act (Actuar):** Durante esta fase, se invoca el código que se está probando. Por lo general, implica llamar a un método y proporcionar los parámetros necesarios.
- **Assert (Afirmar):** En esta fase, se verifica si la acción produjo el resultado esperado. Se utilizan afirmaciones (assertions) para realizar esta verificación. Las afirmaciones son declaraciones que lanzarán una excepción si la condición especificada no se cumple.

A continuación, te mostramos un ejemplo de cómo se aplica la *estrategia AAA* en una prueba:

```
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");
    }
}
```

En el ejemplo, se puede observar lo siguiente:

- **Arrange:** Se definen dos números que serán sumados.
- **Act:** Se llama al método estático *suma()* de la clase Calculadora, pasando los dos números como argumentos.
- **Assert:** Se verifica que el resultado de la suma sea igual a 9. En caso contrario, la prueba fallará y se mostrará el mensaje *"La suma de 4 y 5 debería ser 9"*.

Esta estructura de pruebas ayuda a que sean más legibles y comprensibles, ya que se puede identificar claramente la configuración utilizada, la acción probada y los resultados esperados.

Ciclo de vida de las pruebas unitarias

El **ciclo de vida de las pruebas unitarias** se refiere al *proceso y los pasos que JUnit sigue cuando se ejecutan los tests de una clase*. Estos pasos pueden ser personalizados mediante el uso de anotaciones específicas:

- **@BeforeAll:** Esta anotación se aplica a un método que se ejecuta una vez antes de todos los métodos de prueba en la clase de prueba. Es útil para realizar tareas que se pueden reutilizar en todas las pruebas y no necesitan ser reiniciadas en cada test. El método debe ser estático.

```
class MyTestClass {
    @BeforeAll
    static void initAll() {
   // Código para configurar el estado antes de todas las pruebas ...
    }
}
```

- **@BeforeEach:** Esta anotación se aplica a un método que se ejecuta antes de cada método de prueba individual en la clase de prueba. Es útil para realizar tareas comunes necesarias para las pruebas, como restablecer el estado de algún componente.

```
class MyTestClass {
    @BeforeEach
    void setUp() {
        // Código para configurar antes de cada prueba ...
```

```
    }
}
```

- **@Test:** Esta anotación se aplica a cada método de prueba individual. Cada uno de estos métodos se ejecuta una vez por ejecución de la prueba.

```
class MyTestClass {
    @Test
    void myTest() {
        // Código de la prueba ...
    }
}
```

- **@AfterEach:** Esta anotación se aplica a un método que se ejecuta después de cada método de prueba individual. Es útil para limpiar cualquier estado que se haya configurado para la prueba.

```
class MyTestClass {
    @AfterEach
    void tearDown() {
        // Código para limpiar el estado después de cada prueba...
    }
}
```

- **@AfterAll:** Esta anotación se aplica a un método que se ejecuta una vez después de todos los métodos de prueba en la clase de prueba. Es útil para limpiar el estado que se configuró en el método @BeforeAll. El método debe ser estático.

```
class MyTestClass {
    @AfterAll
    static void tearDownAll() {
   // Código para limpiar el estado después de todas las pruebas aquí
    }
}
```

Estas anotaciones proporcionan un gran control sobre el ciclo de vida de las pruebas, permitiendo configurar y limpiar el estado para pruebas individuales

o para todas las pruebas. Cuando identifiques código repetitivo en tus pruebas, es muy probable que puedas mejorar su legibilidad y mantenibilidad al extraer ese código repetitivo a un método y utilizar una de las anotaciones del ciclo de vida.

Repetición de pruebas

La funcionalidad de **repetición de pruebas** (test) en JUnit 5 **permite ejecutar automáticamente la misma prueba varias veces**. Esta característica resulta útil en situaciones en las que deseamos probar el comportamiento de un método con datos aleatorios o en pruebas no deterministas, es decir, pruebas que pueden generar resultados diferentes en cada ejecución.

Para realizar repeticiones de pruebas en JUnit 5, podemos utilizar la anotación *@RepeatedTest* en lugar de *@Test,* y pasar como argumento el número de repeticiones que deseemos realizar para la prueba.

```
class MyTestClass {
    @RepeatedTest(5)
    void myRepeatedTest() {
        // Código de la prueba aquí...
    }
}
```

En este caso, la prueba "*myRepeatedTest*" se ejecutará 5 veces.

Además, JUnit 5 proporciona un objeto de tipo *"RepetitionInfo"* que podemos utilizar para controlar y obtener información sobre la repetición actual. Podemos inyectar este objeto como argumento en el método de prueba. Por ejemplo:

```
class MyTestClass {
    @RepeatedTest(5)
    void myRepeatedTest(RepetitionInfo repetitionInfo) {
        int currentRepetition = repetitionInfo.getCurrentRepetition();
        int totalRepetitions = repetitionInfo.getTotalRepetitions();
        // Código de la prueba aquí...
    }
}
```

En este caso, *"getCurrentRepetition()"* nos proporcionará el número de la repetición actual (1 a 5), y *"getTotalRepetitions()"* siempre devolverá el número total de repeticiones (5).

Si deseas obtener más información, puedes consultar la documentación de JUnit 5 en el apartado de repetición de pruebas en el siguiente enlace:
👉**Documentación de JUnit 5 - Repetición de pruebas**

Pruebas parametrizados

Las **pruebas parametrizadas** (o tests parametrizados) en JUnit 5 **permiten ejecutar la misma prueba con diferentes conjuntos de datos de entrada**. Esto resulta útil cuando deseas probar un método o función con varios valores diferentes para verificar que funcione correctamente en diversos casos. De esta manera, puedes asegurarte de que tu código sea robusto y funcione como se espera para un rango de valores de entrada.

Para crear una prueba parametrizada en JUnit 5, **debemos utilizar la anotación @ParameterizedTest** en lugar de *@Test*. Luego, debemos proporcionar los datos que se utilizarán en la prueba, lo cual podemos hacer utilizando una de varias anotaciones de origen de datos (@ValueSource, @MethodSource, @CsvSource, @CsvFileSource, entre otras).

- **@ValueSource:** Esta anotación se utiliza para especificar una matriz de valores literales de un solo tipo que se utilizarán como argumentos en las pruebas parametrizadas. Los tipos que puedes utilizar con *@ValueSource* incluyen: short, byte, int, long, char, float, double, boolean, string y class.

```
class MyTestClass {
    @ParameterizedTest
    @ValueSource(ints = {2, 3, 5, 7, 11, 13, 17, 19})
    public void testEsPrimo(int number) {
        assertTrue(Primos.esPrimo(number), "El número debería ser
primo");
    }
}
```

En este caso, la prueba *"testEsPrimo"* se ejecutará ocho veces, una vez para cada valor en la lista de enteros proporcionada en *@ValueSource*.

❗ *Es importante mencionar que @ValueSource solo puede proporcionar un solo argumento para cada invocación del método de prueba. Si necesitamos proporcionar múltiples argumentos, debemos utilizar alguna de las otras anotaciones como @MethodSource, @CsvSource o @CsvFileSource.*

- **@MethodSource:** Esta anotación se utiliza para hacer referencia a un método que proporcionará los parámetros para una prueba parametrizada. El método debe devolver una matriz de objetos u otra estructura de datos que implemente Stream, Iterable o Iterator.

```
@ParameterizedTest
@MethodSource("provideStrings")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

// Este método debe ser estático y no debe tener parámetros.
static String[] provideStrings() {
    return new String[]{"manzana", "banana"};
}
```

En este ejemplo, el método *"provideStrings()"* proporciona los parámetros para la prueba *"testWithExplicitLocalMethodSource()"*.

- **@CsvSource:** Esta anotación permite proporcionar valores para una prueba parametrizada en formato de cadena de texto CSV (Comma Separated Values / Valores separados por comas). Cada fila en el CSV se considera un conjunto de parámetros para la prueba.

```
class CalculadoraTest {
    @ParameterizedTest
    @CsvSource({"1, 2, 3", "2, 3, 5", "3, 5, 8"})
    void testSuma(int num1, int num2, int expectedResult) {
        assertEquals(expectedResult, Calculadora.suma(num1, num2), "La
suma debería ser correcta");
    }
```

```
}
```

En este caso, cada fila en *@CsvSource* representa un conjunto de parámetros para la prueba. La primera y segunda columnas son los números a sumar, y la tercera columna es el resultado esperado de la suma.

- **@CsvFileSource:** Similar a *@CsvSource*, esta anotación permite proporcionar valores para una prueba parametrizada en formato CSV, pero en este caso, los valores se leen de un archivo CSV en lugar de proporcionarse directamente en la anotación.

```
@ParameterizedTest
@CsvFileSource(resources = "/test/day-week-data.csv", numLinesToSkip =
1) // suponiendo que tienes un archivo llamado day-week-data.csv en tu
carpeta de test
void testWithCsvFileSource(String dayWeek, Integer expectedResult) {
    Integer result = getNumberDay(dayWeek);
    assertEquals(expectedResult, result);
}
```

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 - Pruebas parametrizadas**