

---

Auditoria, Calidad y Gestión de Sistemas (ACG)

**Introducción a JUnit**

Curso 22/23

---

---

# Introducción a JUnit

---

## Índice

1. El framework JUnit
2. Test JUnit
3. Clase Assert
4. Annotations
5. Tests Parametrizados
6. Guía para escribir pruebas
7. JUnit y Eclipse
8. Temas Avanzados
  - 8.1. Mocks
  - 8.2. Test Suite

## 1. El framework JUnit

JUnit<sup>1</sup> es un framework de pruebas que sirve para agilizar las pruebas unitarias de código Java. JUnit utiliza anotaciones para identificar los métodos que especifican un test. Lo único necesario para utilizar las anotaciones y clases del framework JUnit es importar las librerías adicionales.

## 2. Test JUnit

Un *test JUnit* (o prueba JUnit) es un método escrito en Java por el desarrollador que ejecuta una pequeña pieza del código a probar y verifica si ha hecho lo que se esperaba. Un método de prueba normalmente tiene el nombre del método que prueba con el prefijo o sufijo *Test*. Estos métodos están contenidos en clases usadas únicamente para las pruebas. A este tipo de clase se les llama *clases de prueba* (Test class). Las clases de prueba suelen tener el mismo nombre que la clase a probar con el prefijo o sufijo *Test*. La implementación de las clases de prueba es igual que la de cualquier clase Java.

Cuando escribimos un test JUnit dentro de una clase de prueba simplemente tenemos que anotar ese método con la etiqueta `@Test`. Esta etiqueta identifica al método como un método de prueba. Los métodos de prueba son utilizados durante la ejecución de las pruebas. Estos métodos se implementan y compilan de la misma forma que un método normal. Tienen que ser públicos, sin parámetros y devolver void.

El código del método de prueba debe:

1. establecer todas las condiciones necesarias para hacer la prueba (para esto JUnit proporciona las *annotations* (ver sección 4)),
2. llamar al método a probar,
3. verificar que el método a probar ha funcionado como esperábamos (para esto JUnit proporciona la clase Assert con métodos útiles (ver sección 3)),
4. dejar el estado del sistema como estaba (con *annotations* de nuevo).

---

<sup>1</sup> <http://junit.org>

Habitualmente, las pruebas se crean en un proyecto diferente al proyecto normal a testear para evitar que los códigos se mezclen.

Generalmente por cada clase a probar (se pueden probar todas las clases o solo aquellas que su código tenga un mínimo de complejidad) existirá una clase de test que prueba sus métodos públicos no triviales.

El siguiente código muestra una clase de prueba con un método de prueba. Obsérvese que el método de prueba lleva la anotación `@Test`:

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

public class MiClaseDeTest{

    @Test
    public void miMetodoDeTest(){
        ...
    }
}
```

JUnit crea una nueva instancia de la clase de prueba antes de invocar a cada `@Test` method.

### 3. Clase Assert

El framework JUnit ofrece la clase `Assert` con una serie de métodos útiles para escribir pruebas. Estos métodos permiten comprobar que los resultados que evalúa la prueba son correctos, es decir, nos permiten verificar si el método a probar ha funcionado como esperábamos.

Algunos de los métodos más importantes de esta clase son los siguientes:

- `void assertEquals(String expected, String actual)`, comprueba que dos primitivas sean iguales
- `void assertFalse(boolean condition)`, comprueba que una condición sea falsa
- `void assertNotNull(Object object)`, comprueba que un objeto no es nulo
- `void assertTrue(boolean condition)`, comprueba que una condición sea cierta

A continuación, se muestra un ejemplo de código de un método de prueba utilizando estos métodos.

Supongamos que se quiere probar el método `largest` de la clase `Largest`, que devuelve el entero mayor de una lista:

```
public class Largest {

    public static int largest (int lista[]){
        ...
    }
}
```

Creamos la clase de prueba LargestTest y el método de prueba largestTest:

```
import static org.junit.Assert.*;

public class LargestTest {

    int []list = new int[]{8,7,9};

    @Test
    public void largestTest() {
        assertEquals(9, Largest.largest(list));
    }
}
```

## 4. Annotations

JUnit proporciona los siguientes elementos "*annotations*" que permiten configurar las pruebas. Estas anotaciones etiquetan a métodos de las clases de prueba.

- `@BeforeEach`, el método se invoca antes de la ejecución de **cada** método de prueba de la clase
- `@AfterEach`, el método se invoca después de la ejecución de **cada** método de prueba de la clase

También se pueden definir métodos para que sean invocados antes o después de la ejecución de **todos** los métodos de prueba de una clase de prueba. Estos métodos vienen anotados con las etiquetas:

- `@BeforeAll`, el método se invoca antes de la ejecución de **todas** las pruebas
- `@AfterAll`, el método se invoca después de la ejecución de **todas** las pruebas

A continuación, se muestra un ejemplo de pruebas que utilizan este tipo de métodos. Las pruebas se definen en la clase TestDB. Suponer que necesitamos algún tipo de objeto de conexión a base de datos para poder hacer nuestras pruebas (testAccountAccess y testEmployeeAccess). En lugar de incluir código para conectarnos y desconectarnos a la base de datos en cada método de prueba podríamos

utilizar los métodos `@BeforeAll` y `@AfterAll` para que lo hicieran. También podríamos tener métodos anotados con `@BeforeEach` y `@AfterEach` para que crearan un objeto cuenta antes de cada método de prueba y lo destruyeran después de cada método de prueba de la clase `TestDB`.

```
public class TestDB {  
  
    static private Connection dbConn;  
    static private Account acc;  
  
    @BeforeAll  
    public static void setUpBeforeClass() {  
        dbConn = new Connection("oracle",15,"fred", "f");  
        dbConn.connect();  
    }  
    @AfterAll  
    public static void tearDownAfterClass() {  
        dbConn.disconnect();  
        dbConn = null;  
    }  
    @BeforeEach  
    protected void setUp() {  
        acc = new Account();  
    }  
    @AfterEach  
    protected void tearDown() {  
        acc = null;  
    }  
    @Test  
    public void testAccountAccess() {  
        ... // Uses dbConn  
    }  
    @Test  
    public void testEmployeeAccess() {  
        ... // Uses dbConn  
    }  
}
```

Al ejecutar las pruebas el orden de ejecución de los métodos será el siguiente:

1. `setUpBeforeClass`
2. `setUp`
3. `testAccountAccess`
4. `tearDown`
5. `setUp`
6. `testEmployeeAccess`
7. `tearDown`
8. `tearDownAfterClass`

## 5. Tests parametrizados

Los test parametrizados posibilitan ejecutar un test con diferentes argumentos. Estos tests son declarados como los tests básicos (@Test methods) pero usan la etiqueta @ParameterizedTest. Además, es necesario declarar al menos una fuente que proporcionará los argumentos para cada invocación del método. Esto se puede hacer con la anotación @ValueSource y un array de valores literales (existen también otras formas de hacerlo, por ejemplo con valores de un fichero csv utilizando la anotación @CsvFileSource).

El siguiente ejemplo muestra un test parametrizado que utiliza la anotación @ValueSource para especificar una cadena de strings con los argumentos para el método de prueba.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromesTest(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

Cuando se ejecuta el método de test parametrizado, cada invocación se hace por separado. La consola de la ejecución mostrará lo siguiente:

```
palindromesTest(String) ✓
├─ [1] candidate=racecar ✓
├─ [2] candidate=radar ✓
└─ [3] candidate=able was I ere I saw elba ✓
```

## 6. Guía para escribir pruebas

A continuación, se presenta una guía con los pasos recomendados para realizar pruebas unitarias utilizando el framework JUnit.

- Elegir qué métodos queremos probar. Se debe partir de la idea de que se deberían probar todos los métodos, es decir, el 100% del código debería testearse. En aquellos casos que se necesite reducir el esfuerzo en el diseño de las pruebas, algunos candidatos a quedarse fuera de las pruebas son aquellos métodos con código muy simple (como podrían ser setters y getters que no realizan ninguna comprobación adicional).
- Para cada método a probar decidir qué verificaciones queremos hacer sobre el método. En este punto tenemos que definir los casos de prueba. Cada caso de prueba

tendrá: un contexto sobre el que se prueba el método, unos valores de entrada para el método y un resultado esperado.

- Construir una clase de prueba por cada clase en la que tenemos un método a probar. Las clases de prueba suelen definirse en un paquete diferente a los del código normal. De esta forma se consigue tener el código mejor organizado.
- Implementar los métodos de prueba en las clases de prueba. La implementación de los métodos se hará utilizando los casos de prueba. El método debe:
  1. Establecer el estado inicial
  2. Llamar al método de prueba
  3. Verificar que el método hace lo que tiene que hacer
  4. Dejar al sistema en el mismo estado
- Construir una Test Suite para organizar la ejecución de los métodos de prueba.
- Ejecutar la Test Suite y comprobar resultados.

## 7. JUnit y Eclipse

Las últimas versiones de Eclipse incluyen las librerías JUnit 5 (podéis comprobarlo en Eclipse -> Acerca de Eclipse -> Installation Details -> Plugins), por lo que no se requiere de ninguna instalación adicional.

Las librerías JUnit permiten al diseñador crear automáticamente esquemas de clases de prueba para las clases Java. Para ello se debe utilizar la opción de menú New-> JUnit Test Case del menú contextual que aparece al posicionar el ratón sobre clases Java.

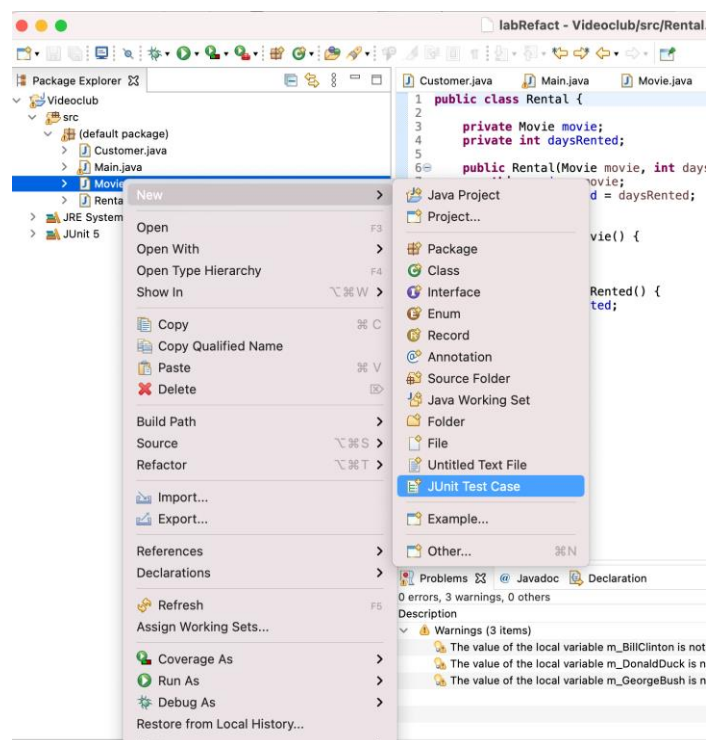


Figura 1. Creación de un caso de prueba JUnit



Al seleccionar esta opción el asistente nos da la opción de elegir los métodos de prueba que queremos crear.

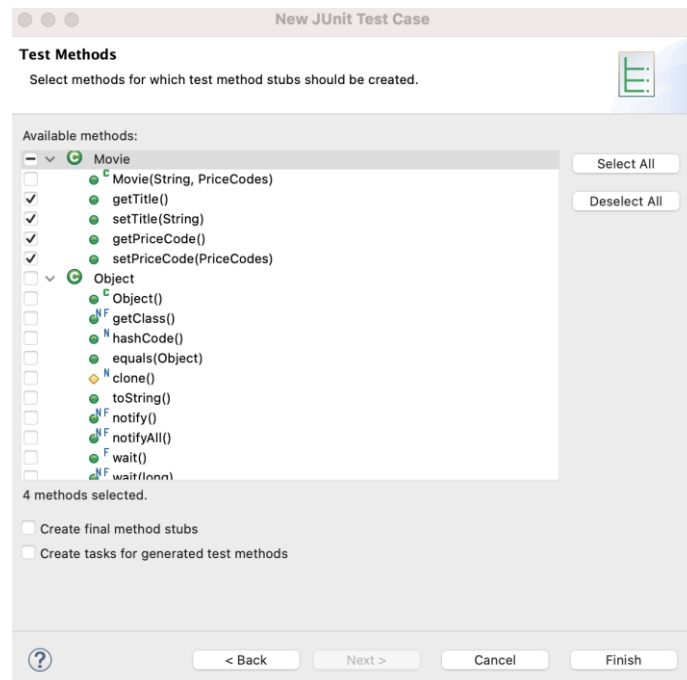




Figura 2. Diagrama de clases inicial

Para ejecutar las pruebas en Eclipse se debe seleccionar la opción Run As->JUnit Test en el menú contextual de las clases de Prueba.

JUnit para Eclipse ofrece una vista que muestra de forma visual si las pruebas han fallado o no. La Figura 1 muestra una captura de pantalla de la interfaz gráfica de ejecución de pruebas de JUnit.

La vista JUnit se divide en dos zonas. La zona superior muestra los resultados de las pruebas. El icono  indica una ejecución de pruebas exitosa. Mientras que el icono  indica una ejecución de pruebas con fallos. Al seleccionar un fallo, en la zona inferior se muestra la traza de ejecución del fallo en forma de árbol. Haciendo doble clic sobre la traza del fallo se accede al código correspondiente que ha provocado el fallo.

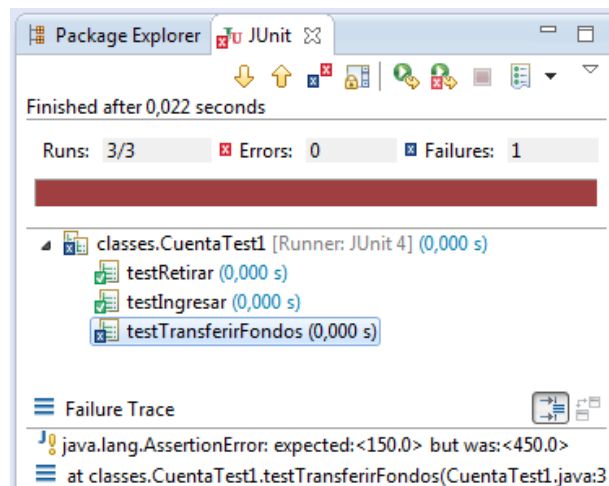


Figura 3. Vista *JUnit*

## 8. Temas Avanzados

### 8.1 Mocks

Un mock es un objeto simulado que sustituye a un objeto real utilizado por la clase o fragmento de código a probar. Los objetos mock son los que se usarán durante la ejecución de la prueba unitaria, lo que posibilitará que no necesitemos el objeto real y que no dependamos de él para poder probar correctamente y de manera completa el módulo.

El concepto de mock es sencillo; si tanto el objeto real como el objeto falso implementan la misma interfaz y el módulo que estamos probando trabaja contra interfaces en lugar de contra objetos, podremos hacer uso indistintamente de objeto falso o del objeto real sin que esto afecte al módulo, siempre y cuando ambos objetos tengan el mismo comportamiento.

El uso de Mocks debería ser un recurso constante dentro de nuestras pruebas, pero la generación de mocks puede resultar un poco pesada, lo que provoca que no siempre se empleen. Existen varios frameworks, como RhinoMock o NMock, que facilitan la generación de estos objetos falsos, aunque cualquier objeto falso que creemos puede considerarse un mock, aún sin hacer uso de un framework.

#### Ejemplo con Proxy.

Veamos un ejemplo. Imaginemos que un método `calculaSalarioNeto` que para calcular los tramos de las retenciones actuales se conecta a una aplicación de la Agencia Tributaria a través de Internet. En ese caso el resultado que devolverá dependerá de la información almacenada en un servidor remoto que no controlamos. Es más, imaginemos que la especificación del método nos dice que nos devolverá `BRException` si

no puede conectar al servidor para obtener la información. Deberíamos implementar dicho caso de prueba, pero en principio nos es imposible especificar como entrada en JUnit que se produzca un fallo en la red. Tampoco nos vale cortar la red manualmente, ya que nos interesa tener una batería de pruebas automatizadas.

La solución para este problema es utilizar objetos mock. Supongamos que el método `calculaSalarioNeto` está accediendo al servidor remoto mediante un objeto `ProxyAeat` que es quien se encarga de conectarse al servidor remoto y obtener la información necesaria de él. Podríamos crearnos un objeto `MockProxyAeat`, que se hiciese pasar por el objeto original, pero que nos permitiese establecer el resultado que queremos que nos devuelva, e incluso si queremos que produzca alguna excepción.

A continuación, mostramos el código que tendría el método a probar dentro de la clase `EmpleadoBR`:

```
public float calculaSalarioNeto(float salarioBruto) throws BRException {
    float retencion = 0.0f;

    if(salarioBruto < 0) {
        throw new BRException("El salario bruto debe ser positivo");
    }

    ProxyAeat proxy = getProxyAeat();
    List<TramoRetencion> tramos;
    try {
        tramos = proxy.getTramosRetencion();
    } catch (IOException e) {
        throw new BRException(
            "Error al conectar al servidor de la AEAT", e);
    }

    for(TramoRetencion tr: tramos) {
        if(salarioBruto < tr.getLimiteSalario()) {
            retencion = tr.getRetencion();
            break;
        }
    }

    return salarioBruto * (1 - retencion);
}

ProxyAeat getProxyAeat() {
    ProxyAeat proxy = new ProxyAeat();
    return proxy;
}
```

Ahora necesitamos crear un objeto `MockProxyAeat` que pueda hacerse pasar por el objeto original. Para ello haremos que `MockProxyAeat` herede de `ProxyAeat`,

sobrescribiendo los métodos para los que queramos cambiar el comportamiento, y añadiendo los constructores y métodos auxiliares que necesitemos. Debido al polimorfismo, este nuevo objeto podrá utilizarse en todos los lugares en los que se utilizaba el objeto original:

```
public class MockProxyAeat extends ProxyAeat {

    boolean lanzarExcepcion;

    public MockProxyAeat(boolean lanzarExcepcion) {
        this.lanzarExcepcion = lanzarExcepcion;
    }

    @Override
    public List<TramoRetencion> getTramosRetencion()
        throws IOException {
        if(lanzarExcepcion) {
            throw new IOException("Error al conectar al servidor");
        }

        List<TramoRetencion> tramos = new ArrayList<TramoRetencion>();
        tramos.add(new TramoRetencion(1000.0f, 0.0f));
        tramos.add(new TramoRetencion(1500.0f, 0.16f));
        tramos.add(new TramoRetencion(Float.POSITIVE_INFINITY, 0.18f));
        return tramos;
    }
}
```

Ahora debemos conseguir que dentro del método a probar se utilice el objeto mock en lugar del auténtico, pero deberíamos hacerlo sin modificar ni el método ni la clase a probar. Podremos hacer esto de forma sencilla si hemos utilizado métodos para crear y acceder a los objetos proxy. Podemos crear una subclase de EmpleadoBR en la que se sobrescriba el método que se encarga de obtener el objeto ProxyAeat, para que en su lugar nos instancie el mock:

```
class TestableEmpleadoBR extends EmpleadoBR {

    ProxyAeat proxy;

    public void setProxyAeat(ProxyAeat proxy) {
        this.proxy = proxy;
    }

    @Override
    ProxyAeat getProxyAeat() {
        return proxy;
    }
}
```

Si nuestra clase a probar no tuviese un método de `getProxyAeat` para crear el objeto `ProxyAeat`, siempre podríamos refactorizarla para extraer la creación del componente que queramos sustituir a un método independiente y así permitir introducir el mock de forma limpia.

El código de JUnit para probar nuestro método podría quedar como se muestra a continuación:

```
TestableEmpleadoBR ebr;

@BeforeEach
public void setUpClass() {
    ebr = new TestableEmpleadoBR();
    ebr.setProxyAeat(new MockProxyAeat(false));

    ebrFail = new TestableEmpleadoBR();
    ebrFail.setProxyAeat(new MockProxyAeat(true));
}

@Test
public void testCalculaSalarioNeto1() throws BRException {
    float resultadoReal = ebr.calculaSalarioNeto(2000.0f);
    float resultadoEsperado = 1640.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}
```

Podremos utilizar mocks para cualquier otro tipo de componente del que dependa nuestro método. Por ejemplo, si en nuestro método se utiliza un generador de números aleatorios, y el comportamiento varía según el número obtenido, podríamos sustituir dicho generador por un mock, para así poder predecir en nuestro código el resultado que dará.

## 8.2 Test Suite

JUnit ofrece la posibilidad de definir un Test suite. Un test suite es una combinación de clases de prueba que permite ejecutar todas las pruebas contenidas en dichas clases a la vez en un orden establecido.

Para crear una Test suit se debe:

- Crear una clase Java
- Anotar la clase con la etiqueta `@RunWith(JUnitPlatform.class)`
- Indicar las clases que forman parte de la suite con la etiqueta `@SelectClasses`, o `@SelectPackages`

A continuación, se muestra un ejemplo de código de la test suite AllTests que agrupa las pruebas de las clases LargestTest y MiClaseDeTest.

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses({ LargestTest.class, MiClaseDeTest.class })
public class AllTests {

}
```

Cuando lancemos a ejecución la test suite AllTests se ejecutarán primero las pruebas de la clase LargestTest, y a continuación las pruebas de la clase MiClaseDeTest.