JUnit

Contenido

Pruebas unitarias

Las pruebas unitarias son comprobaciones de fragmentos de código claramente delimitados, lo que comúnmente asociamos con funciones o métodos. A pesar de esto, lo normal es que, para asegurarnos de tener un código de calidad, se hagan pruebas unitarias del programa completo. Cada una de estas pruebas consiste en asegurarnos de que un fragmento de código da el resultado que nosotros esperemos que dé. Además, también es normal realizar pruebas de excepciones que podrían saltar y otros comportamientos esperados que no tienen que estar relacionados estrictamente con el resultado.

Estas pruebas no intervienen en el código ni modifican su comportamiento. Sólo sirve como herramienta para que el desarrollador pueda hacer un código libre de errores con mayor facilidad. Sin embargo, las pruebas unitarias no comprueban la integración total del código, para lo que habría que realizar otro tipo de pruebas. Además, el desarrollador tiene que planear con cuidado las pruebas para manejar todos los posibles casos que podrían suceder en la ejecución de una función.

El correcto uso de las pruebas unitarias nos permitirá detectar errores a tiempo y localizarlo fácilmente, lo que permite un ahorro de tiempo y costos a largo plazo.

¿Qué es JUnit?

JUnit es uno de los frameworks más famosos para hacer tests unitarios de programas escritos en Java. Actualmente, los tests escritos con JUnit pueden ejecutarse de muchas formas:

- desde un IDE (IntelliJ, Eclipse, Netbeans, VS Code…),
- con un sistema de tareas (Gradle, Ant, Maven),
- o directamente desde la consola usando JUnit Platform.

¿Cómo está estructurado JUnit v5?

La versión más reciente de JUnit divide el framework en tres paquetes distintos:

- **JUnit Jupiter.** API para escribir tests que además contiene un motor que los entiende y es capaz de ejecutarlos.
- **JUnit Vintage.** Igual que Jupiter pero para tests escritos con versiones anteriores de JUnit.
- **JUnit Platform.** Es la base de JUnit. Presta soporte para la implementación de motores que ejecuten alguna clase de tests.

Eclipse viene con JUnit integrado

Como ejecutar los tests desde la terminal, sin ningún IDE

El primer paso es **descargar JUnit Platform Console Launcher** desde el repositorio en Maven (la versión más reciente es la **1.4.0**) y desde ahí, ejecutar el siguiente comando:

```
1   java -jar junit-platform-console-standalone-1.4.0.jar -cp <bin> --scan-classpath
```

donde `<bin>` es el directorio que contiene las clases de los tests ya compiladas.

JUnit necesita que las pruebas estén compiladas

De modo que en `bin/` o en otro cualquier directorio (o en algunos de los subdirectorios del proyecto, porque el comando funciona de manera recursiva) deben estar los ficheros `.class` de las respectivas clases.

Es muy importante que la notación sea la adecuada: por norma general es que la clase debe empezar por `Test` o acabar por `Test`. JUnit se puede configurar (con expresiones regulares) para que tenga en cuentra otro tipo de nomenclaturas, pero esas son las que vienen por defecto. Para más información consulte la ayuda del comando (`java -jar junit-platform-console-standalone-1.4.0.jar -h`).

Cómo se escribe un test

Aserciones

Las aserciones son un conjunto de métodos que nos permiten hacer comprobaciones, para saber si nos da el resultado que nosotros esperamos. En el caso de no coincidir la función hace saltar la excepción `AssertionFailedError`.

En JUnit, generalmente, para cada aserción hay otras dos complementarias del mismo nombre. Una añade un `String` como último parámetro y otra un `Supplier<String>`. Ambas servirán como mensaje de error en caso de que falle la aserción. La diferencia entre ambas es que con `Supplier<String>` se hace una evaluación perezosa, es decir, se evalua cuando es necesario llamarla.

Además, como veremos, es muy común ver una aserción que significa la negación de otra.

```
1   @Test
2   void comprobacionesConMensajesDeError() {
3       assertTrue(true);
4       assertTrue(true,"Esto está mal");
5       assertTrue(false,() -> "Esto está mal, pero con supplier.");
6   }
```

`assertFalse()` , `assertTrue()`

Comprueba si una expresión es falsa o verdadera. Es posible usarse con el tipo primitivo `boolean`, o bien con `BooleanSupplier`.

```
1   @Test
2   void comprobacionesBooleanas() {
3       int a = 5;
4       assertTrue(a == 5);
5       assertFalse(a < 2);
6       assertFalse(() -> (a < 2));
7   }
```

`assertNull()` , `assertNotNull()`

Comprueba si un objeto es nulo o no lo es.

```java
@Test
void comprobacionesDeNulos() {
    Integer a = null;
    assertNull(a);
    a = 8;
    assertNotNull(a);
}
```

`assertEquals()` , `assertNotEquals()`

Comprueba si dos variables tienen el mismo valor o no. En este caso hay una diferencia entre ambos casos. `assertEquals()` funciona con todos los tipos primitivos y con cualquier objeto (por lo que es recomendable sobrecargar el método `equals()` para asegurarnos de que la comparación es la correcta), mientras que `assertNotEquals()` sólo funciona con la clase `Object` .

Sin embargo, desde la versión 1.5 de Java se hace el casteo automático de los tipos primitivos a sus respectivas clases. Por lo tanto, aparentemente funcionarán igual aunque internamente no hagan lo mismo.

```java
@Test
void comprobacionesDeIgualdades() {
    assertEquals("prueba", "prueba");
    assertEquals(5, 5);
    assertNotEquals("prueba", "prueba diferente");
    assertNotEquals(5, 7);
    // más código
}
```

`assertSame()` , `assertNotSame()`

A diferencia de `assertEquals()` ahora sólo funciona con `Object` , porque ahora no comprobamos el valor sino la identificación. En otras palabras, son dos referencias al mismo objeto y por lo tanto ocupan el mismo espacio en memoria. En este caso también tenemos su versión negada.

```java
@Test
void comprobacionesDeIgualdades() {
    // código por aquí...
    String a = "prueba";
    assertSame(a, a);
    assertNotSame(a, new String("prueba"));
    // código por allá...
}
```

`assertArrayEquals()` , `assertIterableEquals()`

El funcionamiento de `assertArrayEquals()` es muy similar a `assertEquals` , pero en vez de trabajar con un único valor trabajamos con un array de valores. Es decir, los arrays tienen que contener los mismos elementos en las mismas posiciones. Esos valores pueden ser de cualquier tipo primitivo o de la clase `Object` .

Por otro lado, `assertIterableEquals()` hace lo mismo pero funciona con `Iterable<>` . Esto cuenta con la ventaja de que los dos objetos no tienen que ser necesariamente iguales, pueden ser dos clases diferentes que implementen `Iterable<>` .

```java
@Test
```

```
    void comprobacionesDeIgualdades() {
        // código por aquí...
        int[] b = {1,2,3};
        int[] c = {1,2,3};
        assertArrayEquals(b,c);

        ArrayList<String> primerIterable = new ArrayList<>();
        primerIterable.add("abc");
        primerIterable.add("segundaPalabra");
        primerIterable.add("última comprobación");
        Vector<String> segundoIterable = new Vector<>();
        segundoIterable.add("abc");
        segundoIterable.add("segundaPalabra");
        segundoIterable.add("última comprobación");
        assertIterableEquals(primerIterable, segundoIterable);
        // código por allá...
    }
```

`assertThrows()` , `assertAll()`

En ocasiones es interesante comprobar si un fragmento de código lanza una determinada excepción. Para ello tenemos `assertThrows()` , que recibe la clase de la excepción que creemos que debería saltar y un `Executable` con el código a comprobar.

También podemos usar `assertAll()` para detectar las excepciones de varios ejecutables. En este caso no concretaremos la excepción, sino que se considerará una prueba satisfactoria si ningún ejecutable lanza excepciones.

```
    @Test
    void comprobacionesDeExcepciones() {
        assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
            int[] a = { 1, 2, 3, 4 };
            System.out.println(a[5]);
        });

        assertAll(() -> {
            int[] a = { 1, 2, 3, 4 };
            System.out.println(a[2]);
        }, () -> {
            System.out.print("No se me ocurre de que hacer las pruebas");
        });
    }
```

`assertTimeout()` , `assertTimeoutPreemptively()`

Ambas opciones permiten comprobar el tiempo que tarda en ejecutarse un fragmento de código. Para ello requiere que le pasemos el tiempo máximo que estimamos que debería tardar con un objeto de tipo `Duration` . La diferencia es que `assertTimeout` se ejecutará en el mismo hilo en el que fue llamado, por lo que si sobrapasa el tiempo estimado tendremos que esperar a que termine igualmente. Con `assertTimeoutPreemptively` , al ejecutarse en otro hilo, se aborta su ejecución si sobrepasa dicho tiempo.

```
    @Test
    void comprobacionesDeTiempoDeEjecucion() {
        assertTimeout(Duration.ofMillis(500), () -> {
```

```
            int total = 0;
            for(int i = 0; i < 10000; i++) {
                total += i;
                //System.out.println(i);
            }
        });
        assertTimeoutPreemptively(Duration.ofMillis(500), () -> {
            int total = 0;
            for(int i = 0; i < 2147483647; i++) {
                total += i;
                //System.out.println(i);
            }
        });
    }
```

`fail()`

Hay casos en los que quizás sea más fácil determinar las condiciones de un fallo por nosotros mismos o nos haga falta una opción más avanzada de la que ofrece JUnit. En estos casos podemos usar `fail()`. Con este método directamente decimos que algo ha fallado sin atender a condiciones.

Con está opción podemos proporcionar un mensaje de error (con una `String` o un `Supplier<String>`) o la causa subyacente con un `Throwable`.

```
@Test
void fallosIncondicionales() {
    fail("Falla porque lo digo yo.");
}

@Test
void fallosIncondicionalesConThrowable() {
    int a = -1;
    if(a < 0) {
        fail(new IllegalArgumentException());
    }
}
```

Otras aserciones

Además de las mencionadas en el documento hay otras aserciones (e incluso otras sobrecargas de estas aserciones) que recomendamos que vean[1]. Entre las aserciones que faltan se encontrarán con `assertLinesMatch()`, pero realmente no es muy utilizada.

Asunciones

Son un conjunto de métodos que permiten abortar tests si no se cumple una condición que se tenía asumido que pasaría. Se creó pensando en aquellos casos en los que las pruebas no tuvieran sentido para un determinado supuesto.

`assumeTrue()`, **`assumeFalse()`**

Comprueba que una expresión booleana es verdadera o falsa, de lo contrario lanzará la excepción `TestAbortedException`, evitando que se ejecuten el resto de instrucciones del test. También es posible pasarle como parámetro un `String` o `Supplier<String>` como mensaje de la excepción. Además, también permite usar `BooleanSupplier` en vez de un valor `boolean`.

```
1   @Test
2   void asuncionesBooleanas() {
3       assumeTrue(true);
4       assumeFalse(false);
5
6       assertNull(null);
7   }
```

`assumingThat()`

Hay casos en los que no queremos que deje de ejecutar el resto del test, sino que deje de ejecutar una parte concreta. Para eso usamos `assumingThat()`, que comprueba una condición booleana y ejecuta un `Executable` en el caso de que dicha condición se valide. Si no se valida continuará con el resto de instrucciones directamente.

```
1   @Test
2   void asuncionesConEjecuciones() {
3       final int a = 5;
4       assumingThat(a < 0 , () -> {
5           assertEquals(a,5);
6       });
7       assumingThat(a > 0 , () -> {
8           assertEquals(a,5);
9       });
10  }
```

Anotaciones

Las anotaciones en Java son metadatos que proporcionan información al compilador sobre el programa, aunque no tengan efecto directo sobre él. Pueden ser utilizadas con clases, métodos, metadatos, campos, parámetros, variables locales, y paquetes.

Las capacidades de las anotaciones no están claramente delimitadas. Una te permite decir al compilador que no te muestre unos determinados warnings (`@SuppressWarnings`), otra te permite informar opcionalmente de que un método es una sobrecarga de otro (`@Override`) y otras pueden indicar a herramientas externas como tienen que actuar respecto a ese código (`@Test`).

Para declarar una anotación se coloca un @ antes del nombre.

En el caso de JUnit se utilizan las etiquetas para configurar los tests. Ahora veremos el funcionamiento de las más importantes.

Anotaciones experimentales

Hay muchas etiquetas de JUnit que se encuentran en estado experimental, y son las que faltan mayormente en este informe. Para buscarlas recomendamos identificar las anotaciones en la documentación de JUnit y luego buscar la información en la api, ya que en la api se encuentran separadas de las estables. Además, la api mostrará información adicional sobre como deben ser declarados los métodos para cada etiqueta.

Tests

Para ejecutar pruebas simples lo indicamos con la anotación `@Test`. Las denominamos simples ya que no es la única forma especificar que es una prueba.

```
 @Test
public void testDeEjemplo() {
    assertTrue(true);
}
```

Tests parametrizados

También específica que un método es una prueba. En este caso nos permite ejecutar dicha prueba más de una vez con el uso de parámetros distintos.

Para proporcionar dichos podemos usar varias etiquetas. La primera de ellas es `@ValueSource`. Esta etiqueta solo admite determinados tipos, que deben escribirse de una manera determinada. Se listan a continuación:

Tipo de valores	Nomenclatura
`short[]`	`shorts`
`byte[]`	`bytes`
`int[]`	`ints`
`long[]`	`longs`
`float[]`	`floats`
`double[]`	`doubles`
`char[]`	`chars`
`String[]`	`strings`
`Class[]`	`classes`

> Consulta la API de JUnit en el apartado **Source of Arguments** para ver otras maneras de dar valores a los parámetros de los tests.

Un par de ejemplos simples:

```
// Este ejemplo ejecuta el tests tres veces:
// primero con a = 1, luego con a = 2 y finalmente con a = 3.

@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
public void testParametrizado(int a) {
    assertEquals(a, 2);             // solo acertará una vez
}
```

Fábrica de pruebas

Hasta ahora hemos visto como escribir tests estáticos. Estos tests están definidos a la hora de compilarlos, y no pueden ser cambiados en tiempo de ejecución.

Sin embargo, podría interesarnos crear un set de tests durante la ejecución de las pruebas. Esto lo hacemos con la anotación `@TestFactory`.

```
@TestFactory
public Stream<DynamicTest> testsDinamicos() {

    /*
     * Estas dos listas representan los parametros de entrada
     * (lo que enviamos al test) y los parametros de salida
     * (los parametros que esperamos que devuelva la función
     * que estamos testeando).
     */
    List<String> ejemploEntrada
        = new ArrayList<>(Arrays.asList("Hola", "que", "tal"));
    List<String> ejemploSalida
        = new ArrayList<>(Arrays.asList("Hola1", "que1", "tal1"));

    /*
     * Las fábricas de tests deben devolver un stream de tests.
     * En esta lista es donde meteremos todos los tests que
     * crearemos dinámicamente.
     */
    Collection<DynamicTest> setDeTests = new ArrayList<>();

    for (int i = 0; i < ejemploEntrada.size(); i++) {
        String      entrada    = ejemploEntrada.get(i);
        String      salida     = ejemploSalida.get(i);
        Executable  aserto     = () -> assertEquals(salida, modifica(entrada));
        String      nombreTest = "Añade 1 al final de la palabra '" + entrada + "'";

        /*
         * Un test dinámico se crea a partir de dos parámetros:
         * > un Executable (un objeto que JUnit emplea para
         *   almacenar funciones anónimas que contiene el aserto
         *   en cuestión), y
         * > una string que represente el nombre del test.
         */
        DynamicTest testDinamico = DynamicTest.dynamicTest(nombreTest, aserto);

        setDeTests.add(testDinamico);
    }

    return setDeTests.stream();
}
```

Ejecución de métodos antes y después de otros tests

JUnit nos proporciona algunas anotaciones para ejecutar instrucciones antes y después de las pruebas. Además, también nos permite decidir si ejecutarlas una vez antes de cada una de las pruebas o una única vez. Son las siguientes:

Anotación	Descripción
`@BeforeAll`	Ejecuta el método una única vez antes de los tests.
`@BeforeEach`	Ejecuta el método antes de ejecutar cada uno de los tests.
`@AfterAll`	Ejecuta el método una única vez después de ejecutar todos los tests.
`@AfterEach`	Ejecuta el método después de ejecutar cada uno de los tests.

```
// Este método se ejecutará para todos los métodos marcados como
// tests, incluidos los de clases anidadas.

@BeforeEach
void antesDeCadaUno(TestInfo info) {
    System.out.println("Ejecutando " + info.getDisplayName());
}
```

Los métodos que lleven la anotación `@BeforeAll` o `@AfterAll` deben ser declarados estáticos.

```
@AfterAll
static void alFinalDeTodos() {
    System.out.println("Esto se va a" +
        "imprimir cuando todos los tests se hayan ejecutado");
}
```

Desactivar pruebas

En JUnit es posible escribir pruebas pero evitar que sean ejecutadas. Resulta más cómodo que comentarlo o quitar su correspondiente anotación, ya que se podría confundir con otros métodos que nunca han sido tests. De está forma podemos bloquear temporalmente su ejecución de forma más limpia. En el caso de usarlo con clases se desactivan todas las pruebas incluidas en ellas. Lo único que hay que hacer es poner la anotación `@Disabled` . Además, podemos indicar con una cadena de texto el motivo por el que está desactivado.

```
@Disabled
@Test
void testDesactivado() {
    assertEquals(2 + 2, 5);
}
```

Customizar los nombres de las pruebas

Por defecto, las pruebas se identifican en la salida por el nombre del método, pero quizás sería más claro si se redactará con espacios u otros símbolos que un método no te permite usar. Para ello podemos utilizar `@DisplayName` tanto para métodos como para clases.

```
@DisplayName("Hola. Soy un test de ejemplo")
@Test
public void testDeEjemplo() {
    assertTrue(true);
}
```

Pruebas encadenadas

Hasta ahora todas las pruebas estaban al mismo nivel, no podiamos establecer una jerarquía. Realmente si es posible. Dentro de la clase principal podemos crear otra con la anotación `@Nested` para especificar que esa clase también contiene pruebas. Si no lo especificaramos simplemente se ignoraría.

Una práctica muy común es utilizar este encadenamiento para formar frases en relación con el nombre sus métodos y con el nombre de la clase superior. De esta forma se puede leer de forma fluida que es lo que representa cada prueba.

Sin embargo, no es una regla fija y puede usarse de la forma que se vea conveniente. En el fondo es una forma de agrupar pruebas de forma que se aprecie visualmente en la salida.

```
1   @Nested
2   class otroTipoDeTests {
3       @Test
4       public void testA() {
5           // código...
6       }
7   }
```

Etiquetas

Tanto las clases como los métodos de prueba pueden etiquetarse con la anotación `@Tag`. Dicha anotación nos permite decidir qué conjunto de tests queremos ejecutar, por ejemplo. Desde Eclipse estas etiquetas pueden definirse en el apartado de configuración de JUnit, en `Run > Coverage Configurations`. Una vez allí seleccionamos/creamos la configuración para ejecutar JUnit y añadimos las etiquetas en donde pone *Include and exclude tags*.

En general, y de manera simplificada, una etiqueta:

- no puede estar vacía o en blanco,
- no debe tener espacios,
- no debe tener caracteres de control ISO,
- no debe tener ninguno de los siguientes caracteres reservados:

Símbolo	Descripción
`,`	coma
`(`	paréntesis izquierdo
`)`	paréntesis derecho
`&`	ampersand
`|`	barra vertical
`!`	signo de exclamación

El motivo por el que estos caracteres están reservados es porque sirven de operadores lógicos para decidir que tests queremos ejecutar.

Este es un ejemplo que hemos extraído y traducido de la API:

Expresión	Tests seleccionados
`producto`	todos los test que tengan la etiqueta `producto`
`catalogo | enviar`	todo los test marcados con `catalogo` **o** `enviar`
`catalogo & enviar`	todo los test marcados con `catalogo` y `enviar`
`producto & !puerta-a-puerta`	todos los test marcados con `producto` pero no `puerta-a-puerta`
`(pequeño | rápido) & (producto | enviar)`	todos los tests marcados con `pequeño` o `rápido` que además estén marcados con `producto` o `enviar` .

```java
@DisplayName("comprueba algo")
@Test
@Tag("A")
@Tag("B")
public void otroTestMas() {
    assertTrue(!false);
}

@DisplayName("comprueba otra cosa")
@Test
@Tag("A")
public void otroTestMas1() {
    assertTrue(!false);
}

@DisplayName("comprueba más cosas todavía")
@Test
@Tag("B")
public void otroTestMas2() {
    assertTrue(!false);
}
```

Salida

Eclipse

Eclipse tiene un panel izquierdo en el que se muestran los tests descubiertos junto con el resultado de su ejecución. En este caso los principales son:

Símbología / Color	Descripción
Gris	Test desactivado (cuenta como *skipped)*
Papel con símbolo de abortado gris	Asunción fallida (cuenta como *skipped)*
Verde	Test correcto
Amarillo con cruz azul	Test fallido
Rojo	Test no escrito correctamente
Hoja de papel	Representa una agrupación de tests, ya sea porque son tests parametrizados o porque están dentro de una clase anidada. La cruz que aparecerá depende de si todos los tests se ejecutaron correctamente (verde) o si hubo fallos (azul)

Los íconos dependen del tema empleado

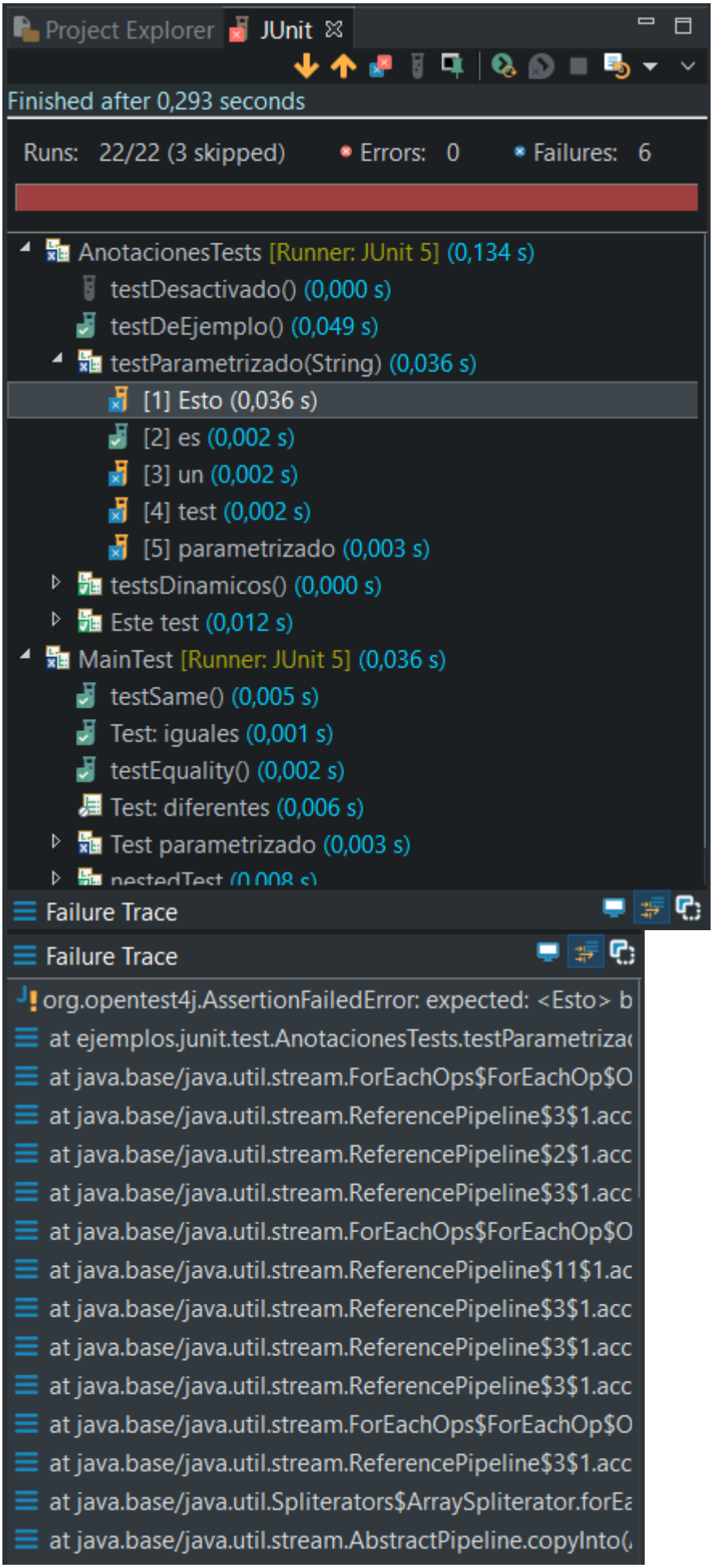
En algunas ocasiones Eclipse puede mostrar unas líneas amarillas, especialmente en condicionales, con el mensaje "3 of 4 branches missed", por ejemplo. Significa que no se han probado todos los valores de los booleanos en cuestión:

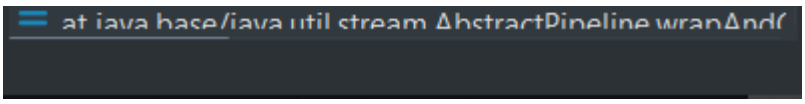
```
1   if (a && b) {
2       // código
3   }
```

En el snippet de código anterior, para que todas las ramas se ejecutasen el código tendría que comprobar los valores de `a` y `b` cuando:

- ambas son falsas (1 rama),
- alguna de ellas es falsa y la otra verdadera (2 ramas),
- ambas son verdaderas (1 rama).

Por debajo de ese panel se muestra la traza de errores que se han producido durante la ejecución de los tests: ya sea porque los tests han fallado (es decir, porque no se ha recibido el resultado esperado), o porque se ha producido un error de compilación (es decir, que los tests están mal escritos).


Project Explorer
JUnit
Finished after 0,293 seconds
Runs: 22/22 (3 skipped)
Errors: 0
Failures: 6
AnotacionesTests [Runner: JUnit 5] (0,134 s)
testDesactivado() (0,000 s)
testDeEjemplo() (0,049 s)
testParametrizado(String) (0,036 s)
[1] Esto (0,036 s)
[2] es (0,002 s)
[3] un (0,002 s)
[4] test (0,002 s)
[5] parametrizado (0,003 s)
testsDinamicos() (0,000 s)
Este test (0,012 s)
MainTest [Runner: JUnit 5] (0,036 s)
testSame() (0,005 s)
Test: iguales (0,001 s)
testEquality() (0,002 s)
Test: diferentes (0,006 s)
Test parametrizado (0,003 s)
Failure Trace
Failure Trace
org.opentest4j.AssertionFailedError: expected: <Esto> b
at ejemplos.junit.test.AnotacionesTests.testParametriza
at java.base/java.util.stream.ForEachOps$ForEachOp$O
at java.base/java.util.stream.ReferencePipeline$3$1.acc
at java.base/java.util.stream.ReferencePipeline$2$1.acc
at java.base/java.util.stream.ReferencePipeline$3$1.acc
at java.base/java.util.stream.ForEachOps$ForEachOp$O
at java.base/java.util.stream.ReferencePipeline$11$1.ac
at java.base/java.util.stream.ReferencePipeline$3$1.acc
at java.base/java.util.stream.ReferencePipeline$3$1.acc
at java.base/java.util.stream.ReferencePipeline$3$1.acc
at java.base/java.util.stream.ForEachOps$ForEachOp$O
at java.base/java.util.stream.ReferencePipeline$3$1.acc
at java.base/java.util.Spliterators$ArraySpliterator.forEa
at java.base/java.util.stream.AbstractPipeline.copyInto(



Terminal

La salida por terminal se parece bastante también a la que Eclipse muestra. En general, al ejecutar la plataforma, esta buscará los tests que estén disponibles y compilados en determinada carpeta (en el árbol se puede identificar incluso si los tests están escritos para Jupiter o si por el contrario están escritos en versiones anteriores de JUnit y deben ejecutarse con Vintage).

También son detectadas las clases principales y las anidadas, que muestra como contenedores al igual que los módulos antes descritos (en azul claro en este caso). Los tests que han fallado se muestran en rojo, los desactivados en violeta, los que no han cumplido alguna asunción se muestran en amarillo y los que se han ejecutado correctamente se muestran en azul (con el `[OK]` verde al lado).

```
.
+-- JUnit Jupiter [OK]
| +-- AnotacionesTests [OK]
| | +-- testDesactivado() [S] void ejemplos.junit.test.AnotacionesTests.testDesactivado() is @Disabled
| | +-- testDeEjemplo() [OK]
| | +-- testParametrizado(String) [OK]
| | | +-- [1] Esto [X] expected: <Esto> but was: <es>
| | | +-- [2] es [OK]
| | | +-- [3] un [X] expected: <un> but was: <es>
| | | +-- [4] test [X] expected: <test> but was: <es>
| | | '-- [5] parametrizado [X] expected: <parametrizado> but was: <es>
| | +-- testsDinamicos() [OK]
| | | +-- A±ade 1 al final de la palabra 'Hola' [OK]
| | | +-- A±ade 1 al final de la palabra 'que' [OK]
| | | '-- A±ade 1 al final de la palabra 'tal' [OK]
| | '-- Este test [OK]
| |   +-- comprueba otra cosa [OK]
| |   +-- comprueba mßs cosas todavÝa [OK]
| |   '-- comprueba algo [OK]
| '-- MainTest [OK]
|   +-- testSame() [OK]
|   +-- Test: iguales [OK]
|   +-- testEquality() [OK]
|   +-- Test: diferentes [A] Assumption failed: pues va a ser que no
|   +-- Test parametrizado [OK]
|   | +-- [1] 1 [X] expected: <1> but was: <2>
|   | +-- [2] 2 [OK]
|   | '-- [3] 3 [X] expected: <3> but was: <2>
|   '-- nestedTest [OK]
|     +-- Test: diferentes [S] void ejemplos.junit.test.MainTest$nestedTest.testNoIguales2() is @Disabled
|     '-- Test: iguales [OK]
'-- JUnit Vintage [OK]

Failures (6):
```

Panel principal que enseña las pruebas descubiertas

```
'-- JUnit Vintage [OK]

Failures (6):
  JUnit Jupiter:AnotacionesTests:testParametrizado(String):[1] Esto
    MethodSource [className = 'ejemplos.junit.test.AnotacionesTests', methodName = 'testParametrizado', methodParameterTypes = 'java.lang.St
ring']
    => org.opentest4j.AssertionFailedError: expected: <Esto> but was: <es>
       org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
       org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:195)
       org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:184)
       org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:179)
       org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:1124)
       ejemplos.junit.test.AnotacionesTests.testParametrizado(AnotacionesTests.java:43)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
       java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
       java.base/java.lang.reflect.Method.invoke(Method.java:566)
       [...]
  JUnit Jupiter:AnotacionesTests:testParametrizado(String):[3] un
    MethodSource [className = 'ejemplos.junit.test.AnotacionesTests', methodName = 'testParametrizado', methodParameterTypes = 'java.lang.St
ring']
    => org.opentest4j.AssertionFailedError: expected: <un> but was: <es>
       org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
       org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:195)
       org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:184)
       org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:179)
       org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:1124)
       ejemplos.junit.test.AnotacionesTests.testParametrizado(AnotacionesTests.java:43)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
       java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
       java.base/java.lang.reflect.Method.invoke(Method.java:566)
       [...]
  JUnit Jupiter:AnotacionesTests:testParametrizado(String):[4] test
```

Traza de errores detallada que enseña por qué los tests fallaron

```
Test run finished after 157 ms
[         9 containers found      ]
[         0 containers skipped    ]
[         9 containers started    ]
[         0 containers aborted    ]
[         9 containers successful ]
[         0 containers failed     ]
[        22 tests found           ]
[         2 tests skipped         ]
[        20 tests started         ]
[         1 tests aborted         ]
[        13 tests successful      ]
[         6 tests failed          ]
```

Resumen final de la ejecución de los tests

Sobre el resumen final de ejecución:

- Los *containers* son los módulos donde están almacenados los tests (ya sea por formar parte de Jupiter o Vintage, o por estar dentro de una clase).
- Sobre los tests:
    - *found:* todos los tests descubiertos (incluidos los desactivados).
    - *skipped:* tests desactivados. Aquí no cuenta las asunciones fallidas como Eclipse.
    - *started:* tests que se ejecutaron.
    - *aborted:* tests que no cumplieron alguna asunción de los tests ejecutados.
    - *successful:* tests que pasaron correctamente.
    - *failed:* tests que fallaron.

Consulte la sección sobre como ejecutar los tests desde la terminal.

Desarrollo dirigido por pruebas

El TDD *(Test Driven Development)* es una práctica de desarrollo de software que se basa en realizar las pruebas antes del código. Al principio esas pruebas fallarán (al no haber código que comprobar), por lo que el siguiente paso es escribir el código que haga que las pruebas se ejecuten satisfactoriamente. El último paso consiste en refactorizar el código de forma que quede limpio y eficiente.

En síntesis:

1. Escribir las pruebas
2. Verificar que las pruebas fallan (para saber que lo que estamos probando no ha sido implementado antes)
3. Escribir el código que hace que las pruebas se ejecuten correctamente
4. Verificar que ahora las pruebas se ejecutan correctamente
5. Refactorizar

Con esta práctica nos obligamos a hacer un correcto diseño del programa antes de empezar a escribir, algo que no es recomendado aunque muchos lo hayamos hecho más de una vez.

Por otro lado, también nos permite centrarnos en el problema principal. Es decir, cuando vamos a crear un método intervienen cosas como el manejo de excepciones, pero esto no es su tarea principal. Siguiendo los pasos en el desarrollo de estas pruebas las excepciones se manejarían en el apartado de refactorización.

Aunque no es obligatorio, es común utilizarlo con pruebas unitarias. De está forma se fomenta que todo el código se encuentre bajo el control de las pruebas.

Aquí encontrarán un **repositorio que contiene un programa en Java cuyo proceso de desarrollo es TDD** , donde se ve que implementa cada uno de los commits.

Referencias

- **Beneficios de las pruebas unitarias**
- **Explicación de la arquitectura de JUnit 5: ¿por qué está dividido en tres módulos?**
- **API de JUnit 5 [HTML]**
- **Cómo empezar rápidamente un proyecto de Maven usando la terminal**
- **Dependencias en Maven de la API de JUnit Jupiter (artefactos)**
- **Desarrollo dirigido por pruebas**
- **Información sobre las anotaciones**
- **Repositorio con todos los ejemplos**
- **Ejemplo de TDD**

---

1. Consulte la **API de JUnit 5** para más información. ↩