# Blog

*Tecnología para Desarrollo*

Tecnología para Desarr...

Tecnología para Negocio

Eventos y seminarios

## ¿Qué nos espera con JUnit 5?

por Alberto Cebrin (https://www.paradigmadigital.com/author/acebrian/)

Madrid, España 16 de marzo del 2017 4 comentarios

Uno de los aspectos más importantes durante el desarrollo de cualquier aplicación es disponer de unas pruebas que detecten cualquier comportamiento anómalo en su funcionalidad.

Actualmente, **en el mundo de Java, hay muchos frameworks de pruebas**, algunos más conocidos que otros, y algunos más novedosos.

Quizá el más utilizado ha sido Junit (http://junit.org/), creado por Kent Beck (https://twitter.com/kentbeck?lang=es), Erich Gamma (https://twitter.com/erichgamma?lang=es) y David Saff (https://twitter.com/dsaff) hace más de diez años. Desde la versión 4, creada en el año 2006, hasta la versión 4.12 (finales del 2014) han ido apareciendo otros frameworks (por ejemplo Spock (https://www.paradigmadigital.com/dev/testing-orientado-bdd-spock-12/)) que han intentado "comerle el terreno".

Debido a esto, los chicos de JUnit se han puesto las pilas para crear una nueva versión mayor de su producto, que es sobre la que vamos a hablar en este post.

En un principio la  nombraron "Junit Lambda" (porque el cambio más grande es la introducción de lambdas (https://www.paradigmadigital.com/dev/pasate-la-programacion-funcional-olvida-los-nullpointerexception/) en muchos sitios), pero finalmente se quedó con el nombre de **JUnit 5**.


(https://www.paradigmadigital.com/wp-content/uploads/2017/03/1-2.png)

Aunque estaba previsto que fuese liberada a finales de 2016, aún no está disponible y la fecha que se contempla actualmente es verano de este año.

No ha sido una buena noticia para los que teníamos ganas de probar la nueva versión. La parte positiva es que el equipo de JUnit 5 liberó a finales de noviembre la versión Milestone 3, así que además de adelantarnos la mayoría de las novedades ¡ya la podemos ir probando!

Pero antes de ver las novedades generales, analicemos la estructura del proyecto. Y es que JUnit ya no es una única biblioteca, sino que es un conjunto de tres subproyectos: JUnit Platform (https://mvnrepository.com/artifact/org.junit.platform), JUnit Jupiter (https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine) y JUnit Vintage (https://mvnrepository.com/artifact/org.junit.vintage/junit-vintage-engine).

## JUnit Platform

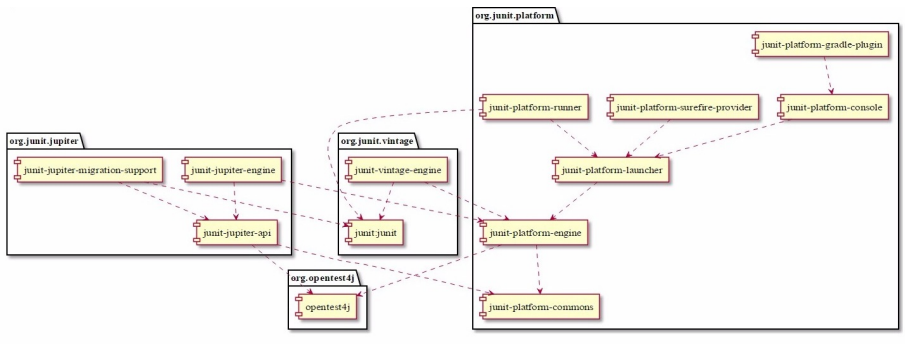
JUnit Platform es la base que nos **permite el lanzamiento de los frameworks de prueba en la JVM** y, entre otras cosas, también es el encargado de proporcionarnos la posibilidad de lanzar la plataforma desde línea de comandos y de los plugins para Gradle y Maven.

## JUnit Jupiter

JUnit Jupiter es el que más utilizaremos a la hora de programar. **Nos permite utilizar el nuevo modelo de programación para la escritura de los nuevos tests de JUnit 5**.

## JUnit Vintage

Junit Vintage **es el encargado de los tests de Junit 3 y 4**, por si alguien los echa de menos.




# Novedades


(https://www.paradigmadigital.com/wp-content/uploads/2017/03/3-1.png)

## Paquetería

Uno de los cambios que menos nos gusta a los programadores es que para que JUnit 5 sea compatible con versiones antiguas, han creado la paquetería org.junit.jupiter para evitar colisiones. ¡Así que ojo con importar algo de org.junit en vez de del nuevo paquete!

## Modificador de acceso public

Un cambio sutil, pero que a los fanáticos del diseño y de la semántica de los tests nos gusta, es que **ni la clase de test ni los propios métodos necesitan ser públicos** (tener el modificador de acceso public).

Esto no es un cambio respecto a la funcionalidad en sí, pero **permite que la clases no puedan ser instanciadas ni sus métodos invocados desde fuera del propio framework de JUnit 5**.

```
1 class MyTest {
2    @Test
3    void allwaysOk() {
4        assertTrue(true);
5    }
6 }
```

# Tags

En JUnit 4 ya había algo similar: las "Categories". En el caso de los tags de JUnit 5 **nos permitirá lanzar conjuntos específicos de tests en función de las etiquetas** utilizando la anotación **org.junit.jupiter.api.Tag** y con el valor que queremos dar como atributo:

```
1  class MyTest {
2
3      @Test
4      @Tag("testExample")
5      void allwaysOk() {
6          assertTrue(true);
7      }
8
9      @Test
10     void allwaysKo() {
11         assertTrue(false);
12     }
13 }
```

En este caso, si ejecutamos todos los tests nos fallará alwaysKo(), pero si especificamos que solo se ejecuten los test con el tag "testExample", los test se ejecutarán correctamente porque alwaysKo será ignorado. Esto es muy útil, por ejemplo, si queremos crear distintos tests por entornos.

Pero esto no es todo. **JUnit 5 nos permite el uso de anotaciones personalizadas para usarlas en los tests** (Composed Annotations). Esto es muy útil, por ejemplo, como sustitutivo de @Tag("value"), pudiendo hacer lo siguiente:

```
1  @Target({ ElementType.TYPE, ElementType.METHOD })
2  @Retention(RetentionPolicy.RUNTIME)
3  @Tag("testExample")
4  @Test
5  public @interface TestExample {
6
7  }
```

Y con esto tendríamos creada nuestra anotación @TestExample, e incluso podríamos modificar el ejemplo anterior para que quedase más limpio y mantenible.

```
1  class MyTest {
2
3      @TestExample
4      void allwaysOk() {
5          assertTrue(true);
6      }
7
8      @Test
9      void allwaysKo() {
10         assertTrue(false);
```

```
    }
}
```

Básicamente lo que hemos hecho ha sido sustituir @Tag(«testExample») y @Test por nuestra anotación **@TestExample**. Y ahora viene la pregunta del millón: **¿cómo hago para ejecutar (o no) unos tests dependiendo de estos tags?**

Según la documentación de JUnit, se puede agregar esa información al plugin de maven:

```
<build>
    <plugins>
        ...
        <plugin>
            <artifactId>maven-surefire-
plugin</artifactId>
            <version>2.19</version>
            <configuration>
                <properties>
                    <includeTags>testExample</includeTags>
                    <excludeTags>integration,
regression</excludeTags>
                </properties>
            </configuration>
            <dependencies>
                ...
            </dependencies>
        </plugin>
    </plugins>
</build>
```

Hacerlo con Graddle:

```
junitPlatform {
    // ...
    filters {
        engines {
            include 'junit-jupiter'
            // exclude 'junit-vintage'
        }
        tags {
            include 'testExample', 'smoke'
            // exclude 'slow', 'ci'
        }
        packages {
            include 'com.sample.included1',
'com.sample.included2'
            // exclude 'com.sample.excluded1',
'com.sample.excluded2'
        }
        includeClassNamePattern '.*Spec'
        includeClassNamePatterns '.*Test', '.*Tests'
    }
    // ...
}
```

O de otras múltiples formas que os animo a investigar en la propia documentación, ya que estos aspectos seguramente sufran modificaciones cuando la versión sea liberada.

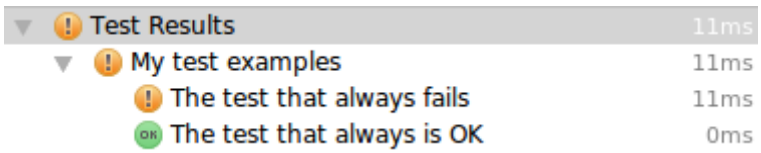
## Nombrar tests

Algunos programadores se quejaban de los nombres sumamente descriptivos que se le ponen a los métodos porque terminaban con nombres muy largos.

Probablemente por esto, el equipo de JUnit decidió agregar la anotación **@DisplayName**, que nos permite dar un nombre más amistoso tanto a clases como a métodos.

La utilidad que tiene esto es que las herramientas que nos muestran los tests lo hagan con el nombre de la anotación en vez de con el propio nombre de la clase o el método.

```
1  @DisplayName("My test examples")
2  class MyTest {
3
4    @Test
5    @DisplayName("The test that always is OK")
6    void allwaysOk() {
7       assertTrue(true);
8    }
9
10   @Test
11   @DisplayName("The test that always fails")
12   void allwaysKo() {
13      assertTrue(false);
14   }
15
16 }
```




(https://www.paradigmadigital.com/wp-content/uploads/2017/03/4-1.png)

Aunque esto sirve de excusa para cambiar el nombre de los métodos por cosas menos descriptivas como "test1()" en vez de "allwaysOk()", no recomiendo que se tienda a hacer nombres de métodos pocos descriptivos.

Los nombres de los test deberían seguir sirviendo para documentar el código que prueban, ya que desde el equipo de JUnit no pueden asegurar que el IDE que estás utilizando para

ejecutar tus tests sea capaz de leer el nombre que tú has establecido con la anotación @DisplayName.

Yo soy partidario de seguir el patrón BDD a la hora de nombrar los test, siguiendo el patrón **Given-When-Then**, como por ejemplo "givenNameAndAddressThenBuildPerson()", y utilizar @DisplayName para agregar documentación al test, como por ejemplo "Given name and address then build the person with this values".

Pero también es cierto que es trabajo doble por lo que, en mi opinión, quizás esta anotación no tenga gran utilidad. Aún así, es mejor tener la opción y no usarla, que querer hacerlo y no poder.

Lo más divertido de dar nombre a los test es que se pueden poner emoticonos, como por ejemplo con @DisplayName(«?»). Realmente no sé si esto es bueno o malo porque todos sabemos que en algún momento nos vamos a encontrar un informe de fallos de los test llenos de emoticonos.

## Aserciones de excepciones

Antes la forma de comprobar que un método devolvía una excepción consistía en: o bien agregar esa información en una anotación a nivel de método, o bien capturar la excepción y chequearla manualmente (eso solía quedar bastante sucio).

Ahora tenemos otras posibilidades, como por ejemplo utilizar el método **"assertThrows()"**, que recibe la excepción que se espera recibir y un Executable, que debería invocar al código que queremos comprobar si lanza una excepción.

Después, capturada la respuesta del método "assertThrows()", dispondríamos de la excepción para hacer tantas aserciones en ella como queramos, con la seguridad de que la excepción se ha lanzado, ya que el propio método comprueba que así sea.

```
1  @Test
2  @DisplayName("Check the expectThrows")
3  void checkExceptions(){
4     String message = "Paradigma rules!";
```

```
    Throwable exception =
assertThrows(IllegalArgumentException.class, () ->
{
        throw new IllegalArgumentException(message);
    });
    assertEquals(message, exception.getMessage());
}


@Test
@DisplayName("Check the expectThrows if there is no
exception throwed")
void checkExceptionNotThrowed(){
    String message = "Paradigma rules!";
    Throwable exception =
assertThrows(IllegalArgumentException.class, () ->
{});
    assertEquals(message, exception.getMessage());
}
```

Ojo, que en mucha documentación previa a la versión M3, se dice que se utilice el método expectThrows, pero este está deprecado a favor del método assertThrows.

## Aserciones múltiples

Muchas veces, a la hora de hacer tests, nos encontrábamos con el problema de tener que ejecutar una gran cantidad de aserciones, que la primera de ellas fallase y no saber si las siguientes eran correctas o no.

Esto con JUnit no pasa, gracias a esta funcionalidad que nos permite ejecutar una colección de aserciones y mostrar todos los errores de estas en el reporte.

El método assertAll recibe, además de un String con el mensaje en caso de que falle la aserción, un N Executable o un Stream<Executable>, siendo Executable una interfaz funcional del propio API de JUnit 5. Por lo que podemos utilizar lambdas para cada una de las aserciones que queramos que compruebe el assertAll.

```
assertAll("Try multiple things",
        () -> assertTrue(true),
        () -> assertTrue(false),
        () -> assertEquals(1, 1),
        () -> assertEquals(1, 2),
        () -> assertEquals(5, 2));
```

Este código nos mostrará, además de las trazas de las excepciones, información sobre cada una de las aserciones que han fallado, lo que nos permite comprobar a la vez varias aserciones independientemente de que una o varias fallen.

```
org.opentest4j.MultipleFailuresError: Try multiple things
(3 failures)
  <no message> in org.opentest4j.AssertionFailedError
  expected: <1> but was: <2>
  expected: <5> but was: <2>
```

## Aserciones de timeouts

¿Alguna vez te has planteado que un error se deba a que cierta funcionalidad tarda mucho en ejecutarse? Yo sí y JUnit nos da una respuesta mediante el método assertTimeout o assertTimeoutPreemptively, a los que se les debe especificar el tipo de Duration que queremos que se compruebe, la cantidad y, en una lambda, el código a probar en ese tiempo.

```
1  @Test
2  void timeoutSuccess(){
3     assertTimeout(Duration.ofMinutes(1), ()->{});
4  }
5  @Test
6  void timeoutFailed(){
7     assertTimeout(Duration.ofMillis(1), ()-
   >Thread.sleep(100));
8  }
```

Una herramienta muy útil si uno de los requisitos que medimos es su tiempo de ejecución.

La diferencia entre estas dos aserciones es que assertTimeout espera a que el código que se estaba comprobando termine, incluso nos indica por cuánto tiempo ha fallado el test. Mientras tanto, assertTimeoutPreemptively aborta el proceso una vez se excede el tiempo.

## Mensajes de aserciones como lambdas

Muchas veces no nos acordamos de pequeños detalles, como el uso de memoria al utilizar String. JUnit 5 piensa por nosotros, así que nos permite utilizar lambdas para generar los mensajes de error de las aserciones, de forma que estos mensajes no sean instanciados a menos que sea necesario. Esto evita construir mensajes complejos en memoria si la aserción es correcta, y solo lo hace en caso de error.

```
1  assertTrue(true, () -> "It will never shown and
   build");
```

## Deshabilitar en vez de ignorar

Aunque a menudo nos sintamos mal por ignorar test, cuando lo hagamos con JUnit 5 debemos recordar que ahora **los test no se ignoran, sino que se deshabilitan**; que es lo mismo, pero con distintas palabras.

Es decir, en vez de tener que poner @Ignore, ahora habrá que escribir **@Disabled**, y que podemos ponerlo tanto a nivel de clase como de método, incluso en "inner classes" de las que hablaremos más adelante.

## Test en inner classes

Con esta versión también **se agrega la posibilidad de probar "inner classes" para estructurar mejor nuestras clases de tests**. Para esto habrá que agregar una inner class y anotarla con **@Nested**.

Después, simplemente crearemos los métodos que queremos probar dentro de la inner class, o incluso crear otra inner class en su interior, anidando todas las que consideremos correctas.

```
1  @DisplayName("My test examples")
2  class MyTest {
3
4         @Nested
5         class InnerClass{
6
7           @Test
8           @DisplayName("InnerClass - The test that
   always is OK")
9           void allwaysOk() {
10            assertTrue(true);
11          }
12
13          @Test
14          @DisplayName("InnerClass - The test that
   always fails")
15          void allwaysKo() {
16            assertTrue(false);
17          }
18        }
19 }
```

Esto permitirá a algunos IDE's colocar los resultados de los test anidados de forma que quede más claro y ordenado.

> **Nota**: Aunque lo repitamos más adelante, las inner class tienen la limitación de que al no poder crear métodos estáticos, no podemos utilizar la anotación @BeforeAll ni @AfterAll.

## Asunciones

Aunque no es muy conocido, en JUnit 4 ya encontraron la solución al problema de que parte de nuestros test se ejecutaran sólo en determinadas circunstancias.

Esto está resuelto con **las asunciones**, que **permiten decidir si cierta parte del código se debe ejecutar o no dependiendo de ciertas excepciones**.

Al igual que ha ocurrido con los mensajes de las aserciones, JUnit 5 nos permite introducir lambdas con el código que queremos que se ejecute sólo en caso de que la asunción sea correcta.

```
1  @Test
2  void tryAssumeTrue() {
3     assumeTrue(true,
4              () -> "Message that will be never
   shown");
5     assumeTrue(false,
6              () -> "Message that will be shown
   because the assumtion is true");
7  }
8
9  @Test
10 void tryAssumeThat() {
11    assumingThat(true,
12             () -> assertEquals(1, 2, ()->"Only
   tested if the assumtion is true"));
13 }
```

## Cambios de nombres del ciclo de vida

Este es un cambio menor, aunque en mi opinión bastante bueno, ya que aclara el manejo de los métodos que queremos que se lancen en un test teniendo en cuenta su ciclo de vida.

Son las anotaciones: **@BeforeEach**, **@AfterEach**, **@BeforeAll**, **@AfterAll** que sustituyen a las anotaciones de JUnit 4 @Before, @After, @BeforeClass, @AfterClass respectivamente, siendo los nuevos nombres bastantes más descriptivos:

1. Si dividimos las palabras de las nuevas anotaciones para comprender su funcionamiento, "Before" y "After" hacen referencia a si se ejecuta antes o después.
2. "Each" se refiere a que será antes o después de cada uno de los tests

3 "All" que será antes o después e la ejecución del conjunto de todos los tests de la clase.

Las anotaciones del ciclo de vida "All" no pueden utilizarse en clases anotadas con @Nested, ya que al ser inner classes, no permiten el uso de métodos estáticos.

## Aserciones con bibliotecas de terceros

Si bien con JUnit 4 disponíamos de una integración directa para utilizar bibliotecas de terceros para hacer nuestras aserciones, por ejemplo con "assertThat()", la nueva versión no dispone de una integración directa con estas bibliotecas.

Pese a esto, **JUnit 5 permite utilizar directamente estas bibliotecas sin disponer de un método que los haga compatibles**.

En la propia documentación ejemplifican cómo hacerlo:

```
1  import static org.hamcrest.CoreMatchers.equalTo;
2  import static org.hamcrest.CoreMatchers.is;
3  import static
   org.hamcrest.MatcherAssert.assertThat;
4
5  import org.junit.jupiter.api.Test;
6
7  class HamcrestAssertionDemo {
8
9      @Test
10     void assertWithHamcrestMatcher() {
11         assertThat(2 + 1, is(equalTo(3)));
12     }
13
14 }
```

## Inyección de dependencias

Sí, el equipo de JUnit 5 ha hecho uno de los mayores cambios del framework: inyección de dependencias, una funcionalidad demandada ya en la anterior versión pero imposible de llevar a cabo porque era necesario utilizar otros runner.

Con esta nueva funcionalidad podemos agregar parámetros de entrada al constructor de la clase de test o a cualquier método anotado con:

- @Test
- @TestFactory

- @BeforeEach
- @AfterEach
- @BeforeAll o @AfterAll

Aunque no podemos inyectar cualquier valor, disponemos de un par de objetos bastante útiles, además de la posibilidad de agregar cualquier objeto a la clase de test o al método anotado de una forma más complicada:

- **TestInfo**: Podemos inyectar este objeto, que dispone de datos sobre la propia ejecución del test. Puedes encontrar más información en este enlace.
    - **getDisplayName()**: Devuelve el nombre del test que esté puesto en la anotación de @DisplayName o el nombre del método en caso de que no exista.
    - **getTags()**: Devuelve una colección con todos los tags que estén afectado al método, tanto las anotaciones @Tag directamente sobre ese método como sobre la clase, sobre la clase que contiene la inner class que contiene el método... y así hasta que el grado de profundidad de métodos, inner classes y la clase principal dejen la película de Origen como un juego de niños.
    - **getTestClass()**: Devuelve la clase propietaria del método que se está ejecutando.
    - **getTestMethod()**: Devuelve el método que se está ejecutando.

```
1  @Test
2  @DisplayName("Test things about the TestInfo")
3  @Tag("ParadigmaTest")
4  void testInfoThings(TestInfo testInfo) {
5     assertEquals("Test things about the
   TestInfo", testInfo.getDisplayName());
6     assertTrue(testInfo.getTags().contains("ParadigmaTest"));
7  }
```

- **TestReporter**: Nos permite modificar o agregar información sobre la ejecución del test.  Esta clase dispone de dos métodos, uno para publicar un mapa de clave-valor y otro para un único conjunto de clave valor.

- **Otros**: Para agregar otros parámetros, hay que extender el inyector de JUnit 5 con la anotación @ExtendedWith, indicando la clase que extiende la inyección. Para más información, recomiendo leer la propia documentación de JUnit al respecto. Un ejemplo puede verse si utilizamos MockitoExtension, que actualmente está en este repositorio.

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>2.6.3</version>
</dependency>
```

```
1  interface Developer{
2     String getName();
3  }
4
5  @ExtendWith(MockitoExtension.class)
6  class MyMockitoTest {
7
8     @BeforeEach
9     void init(@Mock Developer person) {
10        when(person.getName()).thenReturn("Alberto");
11    }
12
13    @Test
14    void simpleTestWithInjectedMock(@Mock Developer
   person) {
15        assertEquals("Alberto", person.getName());
16    }
17
18 }
```

## Métodos default de interfaces

**JUnit aporta también una nueva funcionalidad: la posibilidad de crear tests basándonos en los default method agregados en Java 8**. De forma que, agregando la implementación de un par de métodos, se realiza una serie de test más potentes.

El ejemplo que nos da la documentación de JUnit es bastante bueno en este aspecto, ya que nos permite crear una interfaz para probar el método equals() de una clase y otra para el compareTo().

La idea consiste en crear una interfaz con un método para crear una instancia básica del objeto a probar y posteriormente extenderlo con la interfaz, que nos probará el equals() y con la interfaz que lo hará con el compareTo().

```
1  public interface Testable<T> {
2     T createValue();
3  }
```

La interfaz que probará el equals dejará a implementar un método para crear una instancia distinta a la básica, y luego se escribirán los distintos métodos para comprobar la funcionalidad del equals.

```
1  public interface EqualsContract<T> extends
   Testable<T> {
2
3     T createNotEqualValue();
4
5     @Test
6     default void valueEqualsItself() {
7         T value = createValue();
8         assertEquals(value, value);
9     }
10
11    @Test
12    default void valueDoesNotEqualNull() {
13        T value = createValue();
14        assertFalse(value.equals(null));
15    }
16
17    @Test
18    default void valueDoesNotEqualDifferentValue() {
19        T value = createValue();
20        T differentValue = createNotEqualValue();
21        assertNotEquals(value, differentValue);
22        assertNotEquals(differentValue, value);
23    }
24
25 }
```

Y para probar el compareTo(), se hará una interfaz con un método para crear una instancia menor a la básica, además de, nuevamente, agregar las implementaciones por defecto de una serie de tests que nos permitirán comprobar todas las casuísticas.

```
1  public interface ComparableContract<T extends
   Comparable<T>> extends Testable<T> {
2
3     T createSmallerValue();
4
5     @Test
6     default void returnsZeroWhenComparedToItself() {
7         T value = createValue();
8         assertEquals(0, value.compareTo(value));
9     }
10
11    @Test
12    default void
   returnsPositiveNumberComparedToSmallerValue() {
13        T value = createValue();
14        T smallerValue = createSmallerValue();
15        assertTrue(value.compareTo(smallerValue) >
   0);
16    }
17
18    @Test
19    default void
   returnsNegativeNumberComparedToSmallerValue() {
20        T value = createValue();
21        T smallerValue = createSmallerValue();
22        assertTrue(smallerValue.compareTo(value) <
   0);
```

```
    }

}
```

De esta forma es muy sencillo hacer distintos tests para probar las comparaciones y las igualdades.

```
class StringTests implements
ComparableContract<String>, EqualsContract<String>
{

    @Override
    public String createValue() {
        return "Speaker for the Dead";
    }

    @Override
    public String createSmallerValue() {
        return "Ender's Game";
    }

    @Override
 public String createNotEqualValue() {
        return "Xenocide";
    }

}

@Nested
class IntegerTests implements
ComparableContract<Integer>,
EqualsContract<Integer> {

    @Override
    public Integer createValue() {
        return 42;
    }

    @Override
    public Integer createSmallerValue() {
        return 13;
    }

    @Override
    public Integer createNotEqualValue() {
        return 54;
    }

}
```

## Test dinámicos

Otra de las grandes novedades de JUnit 5 son los **test dinámicos**. Hasta ahora los test se construían en tiempo de compilación, mientras que con esta novedad podemos hacerlo en tiempo de ejecución.

Para poder crear estos tests, por ahora (aunque está previsto que en la Release haya más formas), hay que crearlos a través de un **@TestFactory**, una anotación a nivel de método que nos permite crear un test que será autoejecutado de forma perezosa.

Cabe destacar que para los test creados con la factoría, las anotaciones @BeforeEach y @AfterEach no tienen ninguna utilidad, ya que no nos permiten modificar nada del comportamiento.

Debido a que este aspecto parece el más propenso a sufrir modificaciones desde esta versión M3 a la Release, sugiero comprobar en [la documentación de JUnit (http://junit.org/junit5/docs/current/user-guide/#writing-tests-dynamic-tests)](http://junit.org/junit5/docs/current/user-guide/#writing-tests-dynamic-tests).

**Los test dinámicos están formados por un nombre y una implementación de la interfaz funcional Executable**, que será la que tenga la ejecución del propio test. Esta implementación puede, y casi debe, ser una lambda.

Además, cualquier factoría de tests debe devolver o bien una colección de DynamicTest o un stream de los mismos, ya que serán los que se irán ejecutando.

```
1  @TestFactory
2  Collection<DynamicTest> dynamicTestWithCollection()
   {
3     return Arrays.asList(
4              dynamicTest("1st case", () ->
   assertTrue(true)),
5              dynamicTest("2nd case", () ->
   assertEquals(1, 1))
6     );
7  }
8
9  @TestFactory
10 Stream<DynamicTest> givenNumberTestIfTheyArePair()
   {
11    return Stream.of(2, 4, 6).map(
12             f -> dynamicTest("Given " +f+" test that
   it is pair", () -> Assert.assertEquals(0,f%2)));
13 }
```

## ¿Cómo lo voy probando?

Llegados a este punto espero que a la mayoría, como me ocurrió a mí, os hayan entrado ganas de trastear con JUnit 5 para ir mitigando las ganas de tener entre nuestros teclados la primera release.

En el momento de escribir este post, el único IDE que he comprobado que tiene integración con JUnit 5 es **IntelliJ IDEA**. Entiendo que en un futuro todos lo tendrán. Hasta entonces tenemos la opción de lanzar la ejecución por consola o utilizar el IntelliJ.

Eso sí, y esto es muy importante, **JUnit 5 no soporta ninguna versión de Java inferior a la 8**.

Aunque, por supuesto, también puede hacerse con Gradle. En mi caso he hecho todas las pruebas con maven agregando el plugin.

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.19.1</version>
  <configuration>
     <includes><!--Include here all the names of the
classes who are tests-->
        <include>**/Test*.java</include>
        <include>**/*Test.java</include>
        <include>**/*Tests.java</include>
        <include>**/*TestCase.java</include>
     </includes>
     <properties>
           <!--<includeTags>Introduce here the tags to
execute only them</includeTags> -->
           <!--<excludeTags>Introduce here the tags to
NOT execute them</excludeTags> -->
     </properties>
  </configuration>
  <dependencies>
     <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-surefire-
provider</artifactId>
        <version>1.0.0-M3</version>
     </dependency>
     <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.0.0-M3</version>
     </dependency>
     <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <version>4.12.0-M3</version>
     </dependency>
  </dependencies>
</plugin>
```

Además de las dependencias, por supuesto:

```
<dependencies>
  <dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter-api</artifactId>
     <version>5.0.0-M3</version>
     <scope>test</scope>
  </dependency>
  <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.12</version>
     <scope>test</scope>
  </dependency>
</dependencies>
```

Y si os interesa hacer funcionar JUnit 5 con un proyecto Gradle, podéis echar un ojo a [la documentación oficial (http://junit.org/junit5/docs/current/user-guide/#running-](http://junit.org/junit5/docs/current/user-guide/#running-)

tests-build).

## @AfterAll

Si después de todo lo que he explicado en este post tenéis dudas o inquietudes, lo cierto es que el manual de usuario de JUnit (http://junit.org/junit5/docs/current/user-guide/) es bastante completo. De hecho, es la fuente principal que he utilizado para escribir este post, por lo que os recomiendo visitarlo.

También, si sois muy inquietos y os interesa echar un ojo a su código, podéis hacerlo en su repositorio de Github (https://github.com/junit-team/junit5/milestone/6?closed=1). También tenemos accesible el javadoc (http://junit.org/junit5/docs/current/api/).



### Alberto Cebrin

Arquitecto de software y friki en toda regla, apasionado por lo creativo, imaginativo y diferente. Siempre dispuesto a aprender sobre nuevas tecnologías y metodologías de trabajo. Me encanta terminar el día aprendiendo algo nuevo.

Ver toda la actividad de Alberto Cebrin

## 4 comentarios

**Serandel** 16 marzo, 2017 a las 10:43

«Realmente no sé si esto es bueno o malo porque todos sabemos que en algún momento nos vamos a encontrar un informe de fallos de los test llenos de emoticonos.»

¡Crack! :D

Responder

**Álvaro Navarro** 16 marzo, 2017 a las 23:00

Muy interesantes todas las novedades de JUnit 5. Muy buen post Alberto.
Enhorabuena!

Responder

**Carlos Pedrero** 19 marzo, 2017 a las 06:24

Gran post, muy detallado. Enhorabuena.
Habrá que cacharrear un poco con estos nuevos cambios.

Responder

**Roc Boronat** 21 marzo, 2017 a las 15:06

Bestial el post. Súper divulgativo. Creo que me has ahorrado un montón de horas de lectura y fracasos solucionados a base de stackoverflow. Gracias mil!

Responder

## ESCRIBE UN COMENTARIO

Nombre *

Mail (no será publicado) *

Página web

Comentario:

Enviar comentario

## En Twitter

**@paradigmate**
Terminamos este #TechBrunch (https://twitter.com/search?q=%23TechBrunch&src=hash) reflexionando sobre cómo cambiar el mundo a través de la tecnología. Gracias a Nacho Ri… https://t.co/w2o0Zj94gl (https://t.co/w2o0Zj94gl)

## En nuestro blog

**¿Cómo crear microservicios sobre la nube de Google?**
Desde hace un tiempo hasta ahora, el auge de las nubes públicas es cada vez mayor. Se ha escrito mucho sobre arquitecturas de microservicios en anteriores...


{paradigma

**91 352 59 42**

El responsable de los datos es
Paradigma Digital SL
Vía de las Dos Castillas, 33 - Ática 2, 28224
Pozuelo de Alarcón (Madrid)



contacto

aviso legal y política de privacidad

cookies

## newsletter

Your e-m

Suscribirme

Acepto Los Tratamientos De Datos Indicados En La Política De Privacidad *

Utilizaremos la información que nos facilitas para suscribirte a nuestro envío periódico de newsletters así como mandarte otras publicaciones de Paradigma que puedan ser de tu interés. Nuestra legitimación es tu consentimiento. Tus datos no se cederán a terceros, y puedes modificarlos, darte de baja o eliminarlos cuando quieras. Aquí encontrarás toda la información sobre nuestra política de privacidad. (https://www.paradigmadigital.com/legal/)