JUNIT 5 – what the new Unit Framework can do

20 November 2017 | **Software Engineering** | **Igor Spasic**
Reading time: 7 minutes

It seems like this year Jupiter is the star of the solar system. JUnit, a unit testing framework, saw its version number five come to life just a few weeks ago, after two years of work. It consists of the JUnit platform, and, well, Jupiter – the name already given to the combination of a new programming model and an extension model for writing tests and extensions. Here is an overview.

Setup

With the new version come new artifacts and Java packages. Let's start with Gradle script, where you run the new JUnit using:

```
group: 'org.junit.jupiter', name: 'junit-jupiter-engine', version: '5.0.0'
```

Or, if more familiar with Maven:

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
   <version>5.0.0</version>
</dependency>
```

The new packages include:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
...
```

As I said, we'll see more of Jupiter. Oh, before I forget it: the new JUnit requires Java 8 at runtime!

Writing tests, a brief overview

The main point is the same: we use *@Test* annotation to denote test methods. To assert conditions in tests, we use static methods of the Assertions class. The test class no longer needs to be *public*. There's also a *@DisplayName annotation* that declares a custom display name for the test class or test method. Assertion methods have been improved. First off, we can use lambda expressions instead of an assertion message. Lambda expressions then return the following message:

```
assertTrue(foo > 7, () -> “Value must be greater than 7”);
```

This might not be the best example, as this makes more sense when the message is constructed of parts. With lambda expressions, this is postponed until we really need the message, which leads to better performance.

What’s more interesting is the new method – *assertAll*:

```
assertAll(“foo-bar”,
    () -> assertTrue(foo > 7),
    () -> assertEquals(bar == 12)
);
```

All listed tests will be executed, regardless of failures. In case of a failure, it returns *exception* with information on all the errors.

When testing arrays and collections, we can use *assertArrayEquals* and *assertIterableEquals()*. Another interesting feature is that we can test a list of strings against a regular expression:

```
List<String> expectedLines = Collections.singletonList(“(.*)@(.*)”);
List<String> emails = Arrays.asList(“igor@gmail.com”);
assertLinesMatch(expectedLines, emails);
```

As JUnit 5 runs on Java 8, we can also use default methods as test methods, and we can also apply JUnit annotations to denote them. This feature can significantly improve test code organisation.

Assumptions

Assumptions check test execution based on a condition:

```
assumeTrue(external != null);
assertEquals(3, external.foo);
...
assumingThat(
    external != null,
    () -> assertEquals(3, external.foo);
);
```

In other words, if the assumption is not met, the test is aborted (triggers

TestAbortedException). This doesn't mean that the test has failed, only that the prerequisites for its execution are not met. Normally, such a test is simply marked separately (to say that it's been exempt from testing). We use assumptions to verify states defined outside of tests, but required for the test to run properly – they are not directly related to the subject of testing.

Exceptions

We now finally have help when testing *exceptions*:

```
Throwable throwable = assertThrows(SomeException.class, () -> {
  callMethodThatThrowsSomeException();
});
```

Gone are those boring *try-catch* blocks and *@Expected* annotations.

Disabled

A new annotation used to disable tests is *@Disabled*. It's analogous to the previously used *@Ignore* annotation.

Nested tests

Nested tests are another way to group tests and express the relationship among such groups. For instance, imagine there's a business class *BookService* with several methods: *addBook(), updateBook(), deleteBook()*. A nested test could look something like this:

```
public class TestBookService {
@Nested
@DisplayName("Test Add Feature")
class AddFeature {
  @Test
  void testBookNameMissing()...
  }

@Nested
@DisplayName("Test Update Feature")
class UpdateFeature {
  @Test
  void testChangeBookName()...
  }
}
```

Simply put, nested tests are an inner class of the test class.

Tags

Moving on to tags. We used to have categories, now there are tags – another way to group tests and then filter them:

```
@Tag("Test case")
public class TaggedTest {
  @Test
  @Tag("important")
  void testMethod()…
}
```

Tests can now be enabled and disabled by simply stating a list of tags.

So long, runner, rule, and classrule!

One of the things I really like is that JUnit 5 has removed *Runner, Rule,* and *ClassRule* classes. I've never really liked how they were used in practice. In the new JUnit version, they have been wisely replaced by a single concept-extension model.

We can register extensions declaratively by annotating a test class, or an *@ExtendedWith* method. For example, if we have a particular test where some fields in the class need to be initialised with mockups, we can annotate it like this:

```
@ExtendedWith(MockitoExtension.class)
class MyMockedTest {
   @Mock
   Book book;
   …
}
```

Before and after

Annotations that denote methods initialising and finalising tests have been tidied up. The new annotations are: *BeforeEach, @BeforeAll, @AfterEach, @AfterAll*. Simple and to the point.

The new version of JUnit works the same way as before: it creates a new instance of each test class before executing each test method. This is the default behaviour and is called "per-method". However, you can change this: annotate the test class with

@TestInstance(Lifecycle.PER_CLASS). JUnit then switches to “per-class” behaviour. When using that mode, only one instance is created for each test class.

Dynamic tests

Similar to the idea of *parameterized* tests, JUnit 5 introduces *@TestFactory* and dynamic tests:

```
@TestFactory
public Stream<DynamicTest> createStringGetLengthTests() {
 final String[][] data = {
   {“oblac”, “5”},
   {“jodd”, “4”},
   {“”, “0”}
 };
 return Stream.of(data).map(in -> dynamicTest(
   “test: ” + in[0],
   () -> assertEquals(in[1], String.valueOf(in.getLength(in[0])))));
}
```

@TestFactory, as the name suggests, generates a series of dynamic tests.

Repeated Tests

You can now repeat a test by annotating a method with @RepeatedTest:

```
@RepeatedTest(3)
public void test(RepetitionInfo info) {
   assertTrue(1 == 1);
   logger.info(“Repetition #” + info.getCurrentRepetition());
}
```

The test method now allows an argument of type RepetitionInfo which contains metadata on repetition. I’m still not quite sure when to use this option, but I guess it will become more clear in time.

Dependency Injection

Besides *RepetitionInfo*, JUnit 5 test methods allow parameters of type *TestInfo* and *TestReporter*. These classes contain metadata about the test itself (class, method, name…)

Looking unter the hood

JUnit 5 is more than merely an enhanced JUnit version. The essence of the new version is in

the new architecture. JUnit 5 is constructed to allow using other test libraries and various implementations. As long as they build on JUnit 5 architecture, it's possible to choose and change the test library, without changing the code. This also makes it easier for IDEs, as the tests are treated and executed the same way, regardless of the implementation. Behind all of this is an initiative put forward by the Open Test Alliance, the official body for test standardisation.

JUNIT 5 aced the test

JUnit 5 is certainly an important step forward and conceptual change and crossroads in the field of unit tests. I'm under the impression that things are now conceptually balanced. I'm glad that the creators didn't hesitate to change or remove some of the aspects that also bothered me. They have been replaced with new, clearly well-thought functionalities, which make development easier. I don't know about you guys, but I'll start adopting it in my ongoing projects first thing.