TESTING

JUnit 5.6 Makes Testing Easy with New Features

New features such as the ability to define test execution order and run tests in parallel make this an important release.

by Mert Çalişkan

May 4, 2020

JUnit, the widely used Java unit testing framework, had a major milestone release with JUnit 5 in 2017, which provided a complete rewrite of the whole framework after 10 years of evolution on version 4.x. After the big 5.0.0 release, the JUnit team set a rapid development pace with new minor releases every four to five months; the latest minor version was 5.6.0 on January 20, 2020, which was updated to 5.6.1 on March 22.

I'll revisit the framework, showcasing the newest features it brings to the table by demonstrating them with code samples. I have also annotated the descriptions of relevant features with "(since 5.*x*)" to highlight in which version those features were introduced into the framework.

First, a quick terminology definition: JUnit 5 is composed of three separate modules.

- JUnit Platform is the foundation for launching testing frameworks in the JVM; it is supported by many IDEs and build tools.
- JUnit Jupiter is the newest programming model as well as the `TestEngine` for JUnit 5 tests.
- Finally, there's JUnit Vintage, which is the `TestEngine` for older JUnit 3 and JUnit 4 tests.

(Editor's note: If you have JUnit 4 tests in your toolbox, read Brian McGlauflin's "Migrating from JUnit 4 to JUnit 5: Important Differences and Benefits.")

If you are using Maven, the dependency for JUnit 5.6 can be easily added as shown in **Listing 1**.

Listing 1.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
```

If you are using Gradle, the dependency can be added as shown in **Listing 2**.

Listing 2.

```
testCompile group: 'org.junit.jupiter', name: 'junit
```

Let's move on to the new features!

Defining the Order of Test Execution

You can define an execution order between the test methods of a given class alphanumerically, randomly, with a given `@Order` annotation (since 5.4), or with a custom order definition. The `@TestMethodOrder` annotation (since 5.4) that is placed on the class level will do the trick to enable the order that you'd like to apply.

Executing test methods according to method names alphanumerically is shown in **Listing 3**.

Listing 3.

```
@TestMethodOrder(MethodOrderer.Alphanumeric.class)
class OrderedTest {

    @Test
    void testC() {
        System.out.println("Test C");
    }

    @Test
    void testZ() {
        System.out.println("Test Z");
    }

    @Test
    void testA() {
        System.out.println("Test A");
    }
}
```

In **Listing 3**, the execution order will be as follows: `testA() -> testC() -> testZ()`.

It's also possible to define a random order with the definition `@TestMethodOrder(MethodOrderer.Random.class)`, where execution of the methods will be selected randomly. This would be meaningful when there is no inter-test dependency and you want to make sure to keep it that way, so no unnecessary relationships are built between your tests in upcoming stages.

If you want to explicitly define the order of the tests, you can use `@Order` annotation, as shown in **Listing 4**.

Listing 4.

```
class OrderedTest {

    @Test
    @Order(1)
    void testZ() {
        System.out.println("Test Z");
    }
```

```
    @Test
    @Order(2)
    void testC() {
        System.out.println("Test C");
    }
}
```

In **Listing 4**, the lowest value will have the highest precedence, so the `testZ()` method will execute before the `testC()` method. The methods that are not annotated with `@Order` will have the default value `Integer.MAX_VALUE / 2` (since 5.6).

It's also possible to define a custom ordering mechanism and provide it to the `@TestMethodOrder` annotation. The implementation given in **Listing 5** orders the test methods according to the length of the method name, so short method names execute before long ones.

Listing 5.

```
class CustomOrder implements MethodOrderer {

    @Override
    public void orderMethods(MethodOrdererContext co
        context.getMethodDescriptors().sort(
        Comparator.comparingInt(
            methodDescriptor -> methodDescriptor.get
                                                .get
                                                .len

    }
}
```

The `CustomOrder` implementation should be provided to the `@TestMethodOrder` annotation, as shown in **Listing 6**.

Listing 6.

```
@TestMethodOrder(CustomOrder.class)
class OrderedTest {

    @Test
    void a_very_long_test_method() {
    }

    @Test
    void short_mthd() {
    }
}
```

The execution order will be `short_mthd() -> a_very_long_test_method()`.

Defining Declarative Timeouts

JUnit has the option to mark any test as failed if it takes more time than permitted by a specified threshold. This is very helpful for diagnosing any long-running tests; beginning with JUnit 5.5, it's possible to define a timeout limit with the `@Timeout` annotation. **Listing 7** shows the usage.

Listing 7.

```
@Test
@Timeout(value = 1)
void checkJobDoesNotExceedLimit()
```

```
            throws InterruptedException {
    Thread.sleep(1500);
}
```

In the `checkJobDoesNotExceedLimit()` test method, the timeout value is set to 1 second, which is the default time unit. Obviously, this test will fail since its execution lasts for 1,500 milliseconds. A different unit can be specified as shown in **Listing 8**.

Listing 8.

```
@Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
```

It's also possible to define the timeout value with system properties. **Listing 9** defines the timeout limit as 100 milliseconds for each test that runs.

Listing 9.

```
-Djunit.jupiter.execution.timeout.test.method.defau
```

If a value is also set by the annotation, that will override the value defined by the system property. Here's a list of all the system property keys along with their explanations:

- `junit.jupiter.execution.timeout.default`: Default timeout value for all testable and lifecycle methods
- `junit.jupiter.execution.timeout.testable.method.de` : Default timeout value for all testable methods
- `junit.jupiter.execution.timeout.test.method.defaul` : Default timeout value for methods annotated with @Test
- `junit.jupiter.execution.timeout.testtemplate.metho` : Default timeout value for methods annotated with `@TestTemplate`
- `junit.jupiter.execution.timeout.testfactory.method` : Default timeout value for methods annotated with `@TestFactory`
- `junit.jupiter.execution.timeout.lifecycle.method.d` : Default timeout value for all lifecycle methods
- `junit.jupiter.execution.timeout.beforeall.method.d` : Default timeout value for methods annotated with `@BeforeAll`
- `junit.jupiter.execution.timeout.beforeeach.method.` : Default timeout value for methods annotated with `@BeforeEach`
- `junit.jupiter.execution.timeout.aftereach.method.d` : Default timeout value for methods annotated with `@AfterEach`
- `junit.jupiter.execution.timeout.afterall.method.de` : Default timeout value for methods annotated with `@AfterAll`

The `@Timeout` annotation can be placed at the class level to provide a default timeout value for each test method that class contains. Also, the `@Timeout` annotation can be used on the lifecycle methods annotated with `@BeforeAll`, `@BeforeEach`, `@AfterEach`, or `@AfterAll`. The `@Timeout` annotation is marked as experimental with JUnit 5.6, so it may be subject to change in the upcoming releases.

Conditional Test Execution

As of JUnit 5.1, it's possible to either enable or disable test executions programmatically based on a condition with annotations defined. In this

section, I will list these annotations from different categories along with their usage.

JRE conditions. It's possible to enable or disable a test execution according to the version of the JRE. The `@EnabledOnJre` annotation (since 5.1) enables the execution of a test only on a specified JRE version. The test method in **Listing 10** will only execute on Java version 8.

Listing 10.

```
@Test
@EnabledOnJre(value = JRE.JAVA_8)
void shouldOnlyPassOnJava8() {
       // ...
   }
}
```

The `@DisabledOnJre` annotation (since 5.1) does the opposite and disables the test execution on a given JRE version. It's possible to define JRE versions ranging from 8 to 15 with JUnit version 5.6.

`@EnabledForJreRange` and `@DisabledForJreRange` (both since 5.6) are the two annotations for defining the execution conditions where the annotated method will either be enabled or disabled on a specific range of JREs. An example for enabling a test execution on a specific JRE version range is shown in **Listing 11**.

Listing 11.

```
@Test
@EnabledForJreRange(min = JRE.JAVA_9, max = JRE.JAVA
void shouldRunBetweenJava8AndJava11() {
    // ...
}
```

Operating system conditions. A test execution can also be enabled or disabled according to the version of the operating system (OS). The `@EnabledOnOs` annotation (since 5.1) enables a test execution on macOS, as shown in **Listing 12**.

Listing 12.

```
@Test
@EnabledOnOs(value = OS.MAC)
void shouldOnlyRunOnMacOs() {
    // ...
}
```

The `@DisabledOnOS` annotation (since 5.1) does the opposite and disables the test execution on a given OS version. Possible values for the OS versions are `AIX`, `LINUX`, `MAC`, `SOLARIS`, `WINDOWS`, and `OTHER`.

Environment variable conditions. A test execution can be enabled or disabled according to the existence of an environment variable. The `@EnabledIfEnvironmentVariable` annotation (since 5.1) enables a test execution if the environment variable with the specified name matches the given regular expression. The `@DisabledIfEnvironmentVariable` annotation (since 5.1) does the opposite by disabling test execution. **Listing 13** shows an example

where the test gets executed if the `HOME` environment variable is set to either `/Users/mcaliskan` or `/Users/mertcaliskan`.

Listing 13.

```
@Test
@EnabledIfEnvironmentVariable(named = "HOME",
      matches = "/Users/mcaliskan|/Users/mertcaliska
void shouldOnlyRunOnSpecifiedHOMEDirectory() {
    // ...
}
```

System property conditions. A test execution can be enabled or disabled according to the existence of a system property. The `@EnabledIfSystemProperty` annotation (since 5.1) enables a test execution if the system property with the specified name matches the given regular expression. The `@DisabledIfSystemProperty` annotation (since 5.1) does the opposite by disabling test execution. **Listing 14** demonstrates enabling a test according to the existence of the `os.arch` system property. The test will run only if the underlying JRE is a 64-bit version.

Listing 14.

```
@Test
@EnabledIfSystemProperty(named = "os.arch",
    matches = ".*64.*")
void worksOnlyOn64BitArchitecture() {
    // ...
}
```

In JUnit 5.6, `@EnabledIfEnvironmentVariable`, `@DisabledIfEnvironmentVariable`, `@EnabledIfSystemProperty`, and `@DisabledIfSystemProperty` each support repetition, so they can be defined multiple times on a test method. With JUnit 5.6, the `@EnabledIf` and `@DisabledIf` annotations, which were already deprecated with previous releases, were removed from the codebase. It's recommended to use the annotations defined in the categories above.

Programmatic Extension Registration

The main purpose for extensions is to provide the ability to extend behavior on test classes and methods and reuse that logic on multiple tests. JUnit 5 provided an elegant way by introducing an extension mechanism. Through the Extension API, it's possible to pause at defined points in the lifecycle of a test's execution and execute an extension that is hooked.

With the `@RegisterExtension` annotation (since 5.1), it's possible to register extensions programmatically. A sample extension for logging test method execution times is listed in **Listing 15**.

Listing 15.

```
public class LogTestExecutionTime
        implements BeforeTestExecutionCallback,
                   AfterTestExecutionCallback {

    private long startTime;

    public void beforeTestExecution(ExtensionContex
```

```
    {
        startTime = System.currentTimeMillis();
    }

    public void afterTestExecution(ExtensionContext
        long elapsedTime = System.currentTimeMillis
- startTime;
        System.out.printf("Test Execution took %d ms
    }
}
```

Registering the extension within a test class is shown in **Listing 16**.

Listing 16.

```
class ExtensionTest {

    @RegisterExtension
    LogTestExecutionTime logTestExecutionTime =
                         new LogTestExecutionTime()

    @Test
    void longRunningTest() {
        // …
    }
}
```

With JUnit 5.5, some constraints are applied while registering an extension. One is that the field annotated with `@RegisterExtension` cannot be private. The other is that the extension class must implement at least one of the Extension APIs. Prior to version 5.4, misconfigured extensions were silently ignored.

Executing Tests in Parallel

By default, JUnit Jupiter runs tests sequentially in a single thread, but beginning with version 5.3, it's possible to execute tests in parallel as well. There's an important caveat: This is an experimental feature.

First, the configuration parameter `junit.jupiter.execution.parallel.enabled` should be set to `true` via the system property. The test method, or its containing class, should be annotated with `@Execution(ExecutionMode.CONCURRENT)` (since 5.3). A simple example test is shown in **Listing 17**, where an endpoint is tested five times concurrently. The implementation uses the `@RepeatedTest` annotation where it composes the `@TestTemplate` annotation.

Listing 17.

```
@Execution(ExecutionMode.CONCURRENT)
@RepeatedTest(value = 5, name = "{displayName} {cur
void testEndpointConcurrently() {
   // …
}
```

System property keys that were defined for parallel execution configuration—`junit.jupiter.execution.parallel.mode.default (A)` and `junit.jupiter.execution.parallel.mode.classes.default (B)`—can also be set to either `concurrent` or `same_thread` to enable parallel execution.

Figure 1 demonstrates how the execution will differ between all methods within one class and top-level classes, respectively. The first column has the notation of (A, B) with the specified values `same_thread` and `concurrent`.

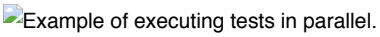

Figure 1. Example of executing tests in parallel

See the documentation for more details on the parallel execution of tests.

Conclusion

The JUnit team is intensively working on 5.x branches with a four-to-five-month release cadence, and it's great to have new features released with the redesigned version of JUnit.

I have described some of the major features shipped between releases 5.0 and 5.6, and there are more. One other thing worth mentioning is that the JUnit artifacts that are produced now contain OSGi metadata. You can easily use them within your favorite OSGi container as well. Another is that as of version 5.6, JUnit also publishes its Gradle module metadata, which is a fine-grained variant-aware dependency resolution mechanism for Gradle users.

Note that some of the JUnit 5 API is still subject to change; the team is annotating the public types with the `@API` annotation and assigning values such as `Experimental`, `Maintained`, and `Stable`.

Learn More

JUnit 5 official documentation

Mert Çalişkan

Mert Çalışkan (@mertcal) is a Java Champion and a coauthor of *PrimeFaces Cookbook* (Packt Publishing, 2013) and *Beginning Spring* (Wiley Publications, 2015). He currently is working on his latest book, *Java EE 8 Microservices*, and he works as a developer on the Payara Server inside the Payara Foundation.

Share this Page