TESTING

Migrating from JUnit 4 to JUnit 5: Important Differences and Benefits

Improvements and new features make JUnit 5 compelling.

by Brian McGlauflin

April 2, 2020

JUnit 5 is a powerful and flexible update to the JUnit framework, and it provides a variety of improvements and new features to organize and describe test cases, as well as help in understanding test results. Updating to JUnit 5 is quick and easy: Just update your project dependencies and start using the new features.

If you've been using JUnit 4 for a while, migrating tests may seem like a daunting task. The good news is that you probably don't need to convert any tests; JUnit 5 can run JUnit 4 tests using the `Vintage` library.

That said, here are four solid reasons to start writing new tests in JUnit 5:

- JUnit 5 leverages features from Java 8 or later, such as lambda functions, making tests more powerful and easier to maintain.
- JUnit 5 has added some very useful new features for describing, organizing, and executing tests. For instance, tests get better display names and can be organized hierarchically.
- JUnit 5 is organized into multiple libraries, so only the features you need are imported into your project. With build systems such as Maven and Gradle, including the right libraries is easy.
- JUnit 5 can use more than one extension at a time, which JUnit 4 could not (only one runner could be used at a time). This means you can easily combine the Spring extension with other extensions (such as your own custom extension).

Switching from JUnit 4 to JUnit 5 is quite simple, even if you have existing JUnit 4 tests. Most organizations don't need to convert old JUnit tests to JUnit 5 unless new features are needed. When that's the case, use these steps:

1. Update your libraries and build systems from JUnit 4 to JUnit 5. Be sure to include the junit-vintage-engine artifact in your test runtime path to allow your existing tests to execute.
2. Start building new tests using the new JUnit 5 constructs.
3. (Optional) Convert JUnit tests to JUnit 5.

Important Differences

JUnit 5 tests look mostly the same as JUnit 4 tests, but there are a few differences you should be aware of.

Imports. JUnit 5 uses the new `org.junit.jupiter` package for its annotations and classes. For example, `org.junit.Test` becomes `org.junit.jupiter.api.Test`.

Annotations. The `@Test` annotation no longer has parameters; each of the parameters has been moved to a function. For example, here's how to indicate that a test is expected to throw an exception in JUnit 4:

```
@Test(expected = Exception.class)
public void testThrowsException() throws Exception
    // ...
}
```

In JUnit 5, this has changed to the following:

```
@Test
void testThrowsException() throws Exception {
    Assertions.assertThrows(Exception.class, () ->
        //...
    });
}
```

Similarly, timeouts have changed. Here's an example in JUnit 4:

```
@Test(timeout = 10)
public void testFailWithTimeout() throws Interrupte
    Thread.sleep(100);
}
```

In JUnit 5, it changes to the following:

```
@Test
void testFailWithTimeout() throws InterruptedExcept
    Assertions.assertTimeout(Duration.ofMillis(10),
}
```

Here are other annotations that have changed:

- `@Before` has become `@BeforeEach`.
- `@After` has become `@AfterEach`.
- `@BeforeClass` has become `@BeforeAll`.
- `@AfterClass` has become `@AfterAll`.
- `@Ignore` has become `@Disabled`.
- `@Category` has become `@Tag`.
- `@Rule` and `@ClassRule` are gone; use `@ExtendWith` and `@RegisterExtension` instead.

Assertions. JUnit 5 assertions are now in `org.junit.jupiter.api.Assertions`. Most of the common assertions, such as `assertEquals()` and `assertNotNull()`, look the same as before, but there are a few differences:

- The error message is now the last argument, for example: `assertEquals("my message", 1, 2)` is now `assertEquals(1, 2, "my message")`.
- Most assertions now accept a lambda that constructs the error message, which is called only when the assertion fails.

- `assertTimeout()` and `assertTimeoutPreemptively()` have replaced the `@Timeout` annotation (there is an `@Timeout` annotation in JUnit 5, but it works differently than in JUnit 4).
- There are several new assertions, described below.

Note that you can continue to use assertions from JUnit 4 in a JUnit 5 test if you prefer.

Assumptions. Assumptions have been moved to `org.junit.jupiter.api.Assumptions`.

The same assumptions exist, but they now support `BooleanSupplier` as well as Hamcrest matchers to match conditions. Lambdas (of type `Executable`) can be used to execute code when the condition is met.

For example, here's an example in JUnit 4:

```
@Test
public void testNothingInParticular() throws Except
    Assume.assumeThat("foo", is("bar"));
    assertEquals(...);
}
```

In JUnit 5, it becomes this:

```
@Test
void testNothingInParticular() throws Exception {
    Assumptions.assumingThat("foo".equals(" bar"),
        assertEquals(...);
    });
}
```

Extending JUnit

In JUnit 4, customizing the framework generally meant using an `@RunWith` annotation to specify a custom runner. Using multiple runners was problematic and usually required chaining or using an `@Rule`. This has been simplified and improved in JUnit 5 using extensions.

For example, building tests with the Spring framework looked like this in JUnit 4:

```
@RunWith(SpringJUnit4ClassRunner.class)
public class MyControllerTest {
    // ...
}
```

With JUnit 5, you include the Spring extension instead:

```
@ExtendWith(SpringExtension.class)
class MyControllerTest {
    // ...
}
```

The `@ExtendWith` annotation is repeatable, meaning that multiple extensions can be combined easily.

You can also define your own custom extensions easily by creating a class that implements one or more interfaces from

`org.junit.jupiter.api.extension` and then adding it to your test with `@ExtendWith`.

Converting a Test to JUnit 5

To convert an existing JUnit 4 test to JUnit 5, use the following steps, which should work for most tests:

1. Update imports to remove JUnit 4 and add JUnit 5. For instance, update the package name for the `@Test` annotation, and update both the package and class name for assertions (from `Asserts` to `Assertions`). Don't worry yet if there are compilation errors, because completing the following steps should resolve them.

2. Globally replace old annotations and class names with new ones. For example, replace all `@Before` with `@BeforeEach` and all `Asserts` with `Assertions`.

3. Update assertions; any assertions that provide a message need to have the message argument moved to the end (pay special attention when all three arguments are strings!). Also, update timeouts and expected exceptions (see above for examples).

4. Update assumptions if you are using them.

5. Replace any instances of `@RunWith`, `@Rule`, or `@ClassRule` with the appropriate `@ExtendWith` annotations. You may need to find updated documentation online for the extensions you're using for examples.

Note that migrating parameterized tests will require a little more refactoring, especially if you have been using JUnit 4 `Parameterized` (the format of JUnit 5 parameterized tests is much closer to `JUnitParams`).

New Features

So far, I've discussed only existing functionality and how it has changed. But JUnit 5 offers plenty of new features to make your tests more descriptive and maintainable.

Display names. With JUnit 5, you can add the `@DisplayName` annotation to classes and methods. The name is used when generating reports, which makes it easier to describe the purpose of tests and track down failures, for example:

```
@DisplayName("Test MyClass")
class MyClassTest {
    @Test
    @DisplayName("Verify MyClass.myMethod returns t
    void testMyMethod() throws Exception {
        // ...
    }
}
```

You can also use a display name generator to process your test class or method to generate test names in any format you like. See the JUnit document for specifics and examples.

Assertions. JUnit 5 introduced some new assertions, such as the following:

- `assertIterableEquals()` performs a deep verification of two iterables using `equals()`.

- `assertLinesMatch()` verifies that two lists of strings match; it accepts regular expressions in the `expected` argument.
- `assertAll()` groups multiple assertions together. The added benefit is that all assertions are performed, even if individual assertions fail.
- `assertThrows()` and `assertDoesNotThrow()` have replaced the `expected` property in the `@Test` annotation.

Nested tests. Test suites in JUnit 4 were useful, but nested tests in JUnit 5 are easier to set up and maintain, and they better describe the relationships between test groups, for example:

```
@DisplayName("Verify MyClass")
class MyClassTest {
    MyClass underTest;

    @Test
    @DisplayName("can be instantiated")
    public void testConstructor() throws Exception
        new MyClass();
    }
    @Nested
    @DisplayName("with initialization")
    class WithInitialization {
        @BeforeEach
        void setup() {
            underTest = new MyClass();
            underTest.init("foo");
        }

        @Test
        @DisplayName("myMethod returns true")
        void testMyMethod() {
            assertTrue(underTest.myMethod());
        }
    }
}
```

In the example above, you can see that I use a single class for all tests related to `MyClass`. I can verify that the class is instantiable in the outer test class, and I use a nested inner class for all tests where `MyClass` is instantiated and initialized. The `@BeforeEach` method applies only to tests in the nested class.

The `@DisplayNames` annotations for the tests and classes indicate both the purpose and organization of tests. This helps you to understand the test report, because you can see the conditions under which the test is performed (`Verify MyClass` with initialization) and what the test is verifying (`myMethod` returns true). This is a good test design pattern for JUnit 5.

Parameterized tests. Test parameterization existed in JUnit 4, with built-in libraries such as `JUnit4Parameterized` or third-party libraries such as `JUnitParams`. In JUnit 5, parameterized tests are completely built in and adopt some of the best features from `JUnit4Parameterized` and `JUnitParams`, for example:

```
@ParameterizedTest
@ValueSource(strings = {"foo", "bar"})
@NullAndEmptySource
void myParameterizedTest(String arg) {
    underTest.performAction(arg);
}
```

The format looks like `JUnitParams`, where parameters are passed to the test method directly. Note that the values to test with can come from several different sources. Here, I just have a single parameter so it's easy to use an `@ValueSource`. `@EmptySource` and `@NullSource` indicate that you want to add an empty string and a null, respectively, to the list of values to run with (and you can combine them, as shown above, if you are using both). There are multiple other value sources, such as `@EnumSource` and `@ArgumentsSource` (a custom value provider). If you need more than one parameter, you can also use `@MethodSource` or `@CsvSource`.

Another test type added in JUnit 5 is `@RepeatedTest`, where a single test is repeated a specified number of times.

Conditional test execution. JUnit 5 provides the `ExecutionCondition` extension API to enable or disable a test or container (test class) conditionally. This is like using `@Disabled` on a test but it can define custom conditions. There are multiple built-in conditions, such as these:

- `@EnabledOnOs` and `@DisabledOnOs`: Enables or disables a test only on specified operating systems
- `@EnabledOnJre` and `@DisabledOnJre`: Specifies the test should be enabled or disabled for particular versions of Java
- `@EnabledIfSystemProperty`: Enables a test based on the value of a JVM system property
- `@EnabledIf`: Uses scripted logic to enable a test if scripted conditions are met

Test templates. Test templates are not regular tests; they define a set of steps to perform, which can then be executed elsewhere using a specific invocation context. This means that you can define a test template once, and then build a list of invocation contexts at runtime to run that test with. For details and examples, see the documentation.

Dynamic tests. Dynamic tests are like test templates; the tests to run are generated at runtime. However, while test templates are defined with a specific set of steps and run multiple times, dynamic tests use the same invocation context but can execute different logic. One use for dynamic tests would be to stream a list of abstract objects and perform a separate set of assertions for each based on their concrete types. There are good examples in the documentation.

Conclusion

Although you probably won't need to convert your old JUnit 4 tests to JUnit 5 unless you want to use new JUnit 5 features, there are compelling reasons to switch to JUnit 5. For example, JUnit 5 tests are more powerful and easier to maintain. In addition, JUnit 5 provides many useful new features, only the features you use are imported, and you can use more than one extension and even create your own custom extensions. Together, these changes and new features provide a powerful and flexible update to the JUnit framework.

Brian McGlauflin

Brian McGlauflin is a software engineer at Parasoft with experience in full stack development using Spring and Android, API testing, and service virtualization. He is currently focused on automated software testing for Java applications with Parasoft Jtest.

Share this Page