By Jupiter!

An Introduction to JUnit 5

Document Facets

Purpose	Introduce the power of JUnit 5.
Audience	Technical for Software Developers
Benefits	New features such as Exception Verification and upgraded Parameterized Tests.
Actions	Adopt JUnit 5 for testing.

Abstract

We don’t write tests just to write more code: we write tests to insure the quality of our product. JUnit 5 is a tool with many features to support this endeavor, matching the subtle earthquake caused by Java’s adoption of lambdas and the functional paradigm. Join me for an overview of this important new tool.

Bio

Steve Gelman is a long-time secure software engineer, full-stack developer, and code janitor. With experience from mainframes to mobile devices, his passion is to master new tools, apply new concepts, conquer new platforms, and pass the knowledge on.
In his off-hours, he sings in a choir, exchanges bad jokes and horrible puns with his wife and two sons, and fixes things around his house. Lots of things around his house.

1 Introduction

1.1 Why Test

Proper testing can double, or even triple, the size of the codebase. It also takes a lot of time to write and maintain the tests. So why do it.

1.2 Opening Joke

“The Koala tea of Mercy.”

1.2.1 To Insure Quality

This is the primary reason we write tests. We need to insure:

- Is the code *accurate*? Does the code return the correct results?
- Is the code *functional*? Does the code perform the correct task?
- Is the code defect-free? Testing helps simplify the debugging process by narrowing the location of defects.

1.2.2 To Document The Business Requirements And Design

How do we document the business requirements and design? In the “bad old waterfall” days, we used to create UML diagrams to architect and communicate design. We would write user manuals. We created data dictionaries. In our new agile world, I see most places put these in a poorly-organized collection of JIRA tickets. User stories are generally incomplete, or worse, written by the same developers writing the code. Unit test can document the system when nothing else does.

- Testing a piece of code forces you to define for what that code is responsible.
- Writing the test first forces you to think through your design and what it must accomplish before you write the code. Save your notes!

1.2.3 To Make The Process More Agile

Agile methodologies are less cumbersome, which is the point. Testing helps by:

- Making refactoring a safer process.
- Facilitates small, incremental changes.

On the other hand, testing makes re-purposing the code harder.

1.2.4 Makes Coding Better, Faster, and Cheaper

Testing helps agile realize its potential to make the development process less expensive. It helps by:

- Finding defects sooner.

- Keeping each change small and focused.

1.3 Presentation Goals

So these are the goals of this presentation:

1. Identify the pieces of JUnit5
2. Show how to run JUnit5
3. Give examples of the basic features of JUnit5
4. Provide a summary of advanced features

And realize my daily goal of eat something I'll regret tomorrow

1.4 The Pieces of JUnit5

JUnit5 has three pieces:

JUnit5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

Where:

- JUnit Platform:
  - Test engine for running tests on the JVM
  - Includes the test engine API for developing testing frameworks
  - Existing engines include
    - Console Launcher
    - JUnit4 runner
    - IDE support for Eclipse, Intellij, Netbeans, Visual Studio Code, etc.
    - Build tools: Ant, Maven, Gradle, etc.
- JUnit Jupiter:
  - JUnit5 programming model
  - JUnit5 extension model
- JUnit Vintage:
  - Works with JUnit3 and JUnit4 APIs (except for rules).
  - Provides a test engine for JUnit3 and JUnit4

Generally, the terms *JUnit5* and *JUnit Jupiter* are used interchangeably. While this is not correct semantically, most people will understand by context.

1.5 Why Move To JUnit5

So why bother? Why move to JUnit5?

- Gentle migration: little conflict between JUnit3, JUnit4, and JUnit Jupiter
- Enhanced reporting tools
- Finer-grained execution control
- Dependency injection
- Multiple executions of tests with different values
- Test templates, dynamic tests, and an extension framework
- Works with Kotlin

1.6 Alternative Testing Frameworks

Before we dive into the depths of JUnit5, I do want to mention there are other frameworks:

- JUnit4
- NGTest
- Groovy
- Spock

The advantages of JUnit5 is:

- Migration from JUnit4 is gentle.
- Not having used NGTest, I can’t make a comparison.
- Groovy is another language to learn. Not everyone knows it.
- Spock is a nice framework, but it rests on Groovy. See above.

2 Presentation

2.1 Running JUnit5

JUnit 5 can be run in many ways. However, if we set up our projects in a Maven or Gradle environment, we can let the build tool do the “heavy lifting” to pull in the libraries and configure itself to run.

So here are various ways to run JUnit5

2.1.1 JUnit5 Alone In Maven

Example: a010-solo-maven

Running JUnit5 without any additional libraries is simple:

1. Add a dependency for *org.junit.jupiter:junit-jupiter:5.4.0*
2. Make sure your *maven-surefire-plugin* is version *2.22.1* or higher.

2.1.2 JUnit5 In Gradle

Example: a011-solo-gradle

Running JUnit5 without any additional libraries is simple in Gradle as well:

1. Add a dependency for *org.junit.jupiter:junit-jupiter:5.4.0*
2. In the *test* task add the *useJUnitPlatform()* command

See the Gradle Release Notes for more details.

2.1.3 Running JUnit5 With JUnit4 In Maven

Example: a020-junit4-and-5-maven

JUnit 5 can run with JUnit 4, as well, with just a few more complications:

1. Add a dependency for *org.junit.jupiter:junit-jupiter:5.4.0*
2. Make sure your *maven-surefire-plugin* is version *2.22.1* or higher.

And also:

3. *Add a dependency for org.junit.jupiter:junit-jupiter-engine:5.4.0*
4. Add a dependency for *junit:junit:4.12*
5. *Add a dependency for org.junit.vintage:junit-vintage-engine:5.4.0*

Step 3 above is not strictly necessary. However, the Intellij IDEA 2018.3.4 IDE had random issues without the Jupiter test engine being explicitly set as a dependency. As I was never able to identify the issue, I recommend keeping step 3 in as a work-around.

2.1.4 Running JUnit5 With Spring Test And Mockito

Example: a030-spring-test-mockito-maven

Now we're adding in more complexity, but the dependency list is still shorter than the pre-Spring Boot ecosystem. What we need is:

1. *org.springframework.boot:spring-boot-starter-parent:<version>*
   The example uses version 2.1.4-RELEASE.
2. *org.springframework.boot:artifactId>spring-boot-starter-web*
   No versions are needed because the *spring-boot-starter-parent* sets them all.
3. *org.springframework.boot:artifactId>spring-boot-starter-test*

4. *org.junit.jupiter:junit-jupiter-api*
5. *org.junit.jupiter:junit-jupiter-engine*
6. *org.mockito:mockito-junit-jupiter*

And you're off to the races!

2.1.5 Recap

You basically have to add the JUnit5 API and engine and you're set. You don't even have to add the engine because the Maven Surefire plugin already uses the engine in most circumstances. Gradle just needs the junitPlatform() command added to the test task.

2.2 JUnit5 Features

So, now that we're all experts in running JUnit5, what can we do with it?

2.2.1 Display Names

b010-display-names-and-nesting-maven

2.2.1.1 DisplayNamesAsSentencesTest

This first test suite shows the basic usage of the DisplayName annotation. You can simply add a more readable name to a test, as shown in the englishName test.

You can add a name in another language, as shown in the spanishName test. You can use any character in the codepage you are working in. These files a in UTF-8.

You can even use a serious of emojis, as shown in the emojiName test. Why would you want to do such a thing? Who cares why? You can!

2.2.1.2 IntegerCalculatorTest

The problem illustrated in the test suite above is that you need to add a method name and a DisplayName, which violates the DRY software development principle: Don't Repeat Yourself!

To get around this, you can use the DisplayNameGeneration annotation, either on each test method or the entire class. This annotation lets you specify a class implementing the DisplayNameGenerator interface, as shown in the IntegerCalculatorTest class. By using a display name generator, JUnit Jupiter will automatically change your Java identifier to a more English-like name.

2.2.1.3 DisplayNamesGeneratedCustomTest

This class has a number of features.

1. Line 25 has a DisplayNameGenerator annotation.
2. Line 29 has a test with some foreshadowing: dependency injection.
3. Line 34 overrides the generated test name with a DisplayName annotation.

4. Starting in line 39 is a nested inner class with its own DisplayNameGeneration annotation.

2.2.2 Nested Tests

b010-display-names-and-nesting-maven

The previous example showed an example of nesting tests. The purpose for nesting tests is to better control the test life-cycle.

2.2.2.1 LifeCycleDemoATopLevelOnlyTest

Running this test shows how the life-cycle runs on a test without nesting. There are two methods, test1 and test2, and the methods annotated with BeforeAll, BeforeEach, AfterEach, and AfterAll run as you would expect.

This is the standard life-cycle of JUnit5 tests. Other than the new names of the annotations, this is your JUnit4 life-cycle.

2.2.2.2 LifeCycleDemoBNestedTest

Adding in a nested class makes things a little more interesting. If you nest with inner classes, notice a few things:

1. Inner classes, aka nested classes, can't have static methods, so no BeforeAll and AfterAll methods for the nested class. It will still use those methods in the top-level class.
2. In between the tests in the child class, the BeforeEach and AfterEach methods of the top-level class fire.
3. In the grandchild class, those methods fire for both the top-level and child class.

Nested classes are ideal for testing sub-conditions that require additional changes to the environment, such as adding data to databases or altering the behavior of mock objects.

2.2.3 Execution Control

One of the enhanced features with JUnit5 is execution control. You can exclude tests by tags: annotations that act like JUnit4's categories on steroids, expanded assumptions, and setting the order of test execution.

2.2.3.1 Tags Provided By JUnit5

JUnit5 gives you a bunch of tags that provide conditional test execution.

- We no longer @Ignore tests, we @Disable them:
  @Disabled("Optional comment (which you better add or I'll break your fingers!)")

- We can enable or disable tests by operating system:
  @EnabledOnOs({OS.LINUX, OS.WINDOWS})
  @DisabledOnOs(OS.MAC)
- Likewise with the JVM version:
  @EnableOnJre({Java_8, JAVA_9})
- @DisableOnJre(Java_7)
- System Properties:
  @EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
  @DisabledIfSystemProperty(named = "ci-server", matches = "true")
- Environmental Variables:
  @EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
  @DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
- Scripts (experimental!):
  @EnabledIf("2 * 3 == 6" && Math.random() < 0.314159)
  @DisabledIf("/32/.test(systemProperty.get('os.arch'))")) // Supports pattern matching!

2.2.3.2 Custom Tagging And Filtering

b040-tagging-and-filtering-maven

Tags, in JUnit5, are like Categories in JUnit4, but on steroids. Basically, they are markers that tell the compiler something important about a class. This is similar to the ancient Clone and Serialization interfaces.

Note that you have to explain the tag to the compiler at some point. This isn't magic.

2.2.3.2.1 EndToEndTest Tag

The EndToEndTest annotation tags a class or method as an end-to-end test. Notice on line 22 the Test annotation, which lets you use this tag with or without a separate test annotation.

The UserAcceptanceTest class shows the usage of the EndToEndTest tag. There are two methods: testAReallyLongProcess and testSomethingShort. If we run both tests, the first one will fail, so we filter out the tag by excluding it as a group in the maven-surefire-plugin.

The testSomethingShort has two flavors: one tagged with TestOnWindows, the other EnabledOnOs(MAC). This is a custom tag I created using an EnabledOnOs tag. It runs on my Windows machine. The MAC version only runs on a Mac.

2.2.3.2.2 Flash Tag

The Flash tag should mean the tagged class or method is executed very quickly. Again, you will have to tell the compiler what Flash means.

Note that the test annotation is not included in this tag, so you would need to specify both the flash and the test tags to make something a fast test.

The UltraFastTest shows how to use the Flash tag.

2.2.3.3 Assumptions

b040-tagging-and-filtering-maven

One of JUnit4's lesser used features is Assumptions. An Assumption must be true or JUnit will disable the test. Class AssumptionTest show three basic flavors of assumptions.

1. Method testOnlyOnCiServer demonstrates a positive assumption: it assumes there is an environmental variable ENV set to "CI" and will not run if that is false.
2. Method testNotOnACiServer demonstrates a negative assumption: it assumes there is NOT an environmental variable ENV set to "CI." It will not run if this is true.
3. Method testInAllEnvironments will always run and always execute the second test. It will only execute the first test if there is an environmental variable ENV set to "CI."

JUnit5 has beefed up the JUnit4 assumptions with additional methods, including those that use lambda expressions, as shown in tests 2 and 3.

NOTE: Some versions of JUnit5 are missing a class definition related to assumptions, so to use this feature in JUnit5, add a dependency for JUnit4. You do not need to add the junit-vintage-engine.

2.2.3.4 Ordering Test Execution Sequence

b050-ordering-maven

The @TestMethodOrder annotation lets you sequence the execution of the tests. This is supposedly a no-no in testing; after all, each test should be isolated from the others. However, there are times when the order of execution will affect the results of the tests. For example: integration tests with databases.

@TestMethodOrder lets you specify the ordering of the execution in four ways:

1. Alphanumeric: sorts test methods alphanumerically based on their names and formal parameter lists.
2. OrderAnnotation: sorts test methods numerically based on values specified via the @Order annotation. This is demonstrated~~demostrated~~ in test B_OrderedIntegerCalculatorTest of the example.
3. Random: orders test methods pseudo-randomly and supports configuration of a custom seed.
4. Roll your own by implementing a custom MethodOrderer interface.

2.2.4 Exception Handling

b060-exception-handling-maven

JUnit5 turbocharges the way exceptions can be handled in tests. The @Test annotation no longer has an *expected* clause. You can either go back to the JUnit3 way of catching exceptions, i.e. a try-catch block, or use the *assertThrows* statement.

2.2.5 Lambdas

2.2.5.1 Assertions

b070-lambdas-maven

JUnit5 uses lambda expressions in a number of situations. As the LambdaAssertsTest class shows, in addition to the assertThrows from the previous topic, all of the assertions now have a lambda forms. In addition to standard assertions, the lambdas allow assertions to be grouped or dependent on other assertions.

2.2.5.2 Assumptions

b070-lambdas-maven

Assumptions don't seem to be as flexible. So, the class LambdaAssumptionTest show the basic usage of lambdas in assumptions, repeated from an earlier project.

2.2.6 Dependency Injection

b080-dependency-injection-maven

I'm going to cover three more things, and these are the coolest features to me. I'm going to start with Dependency Injection, or Injection of Dependencies. Whatever.

Up to now, test constructors and methods were not allowed to have parameters. Now they have them, giving us greater flexibility and dependency injection. An API, *ParameterResolver*, is an extension to JUnit5 that can resolve parameters of a constructor or method AT RUNTIME!

There are three built-in ParameterResolvers, or you can roll your own. The built-in ones are:

1. TestInfoParameterResolver: supplies information about the test to the test using the TestInfo interface. It's the big brother of the TestName rule from JUnit 4.
2. TestReporterParameterResolver: allows you to publish additional information about the current test run in the, well, test runner. This uses the TestReporter interface.
3. RepetitionInfoParameterResolver: supplies information about current and total number of repetitions of a repeated test, something I'll talk about in the next section, using the RepetitionInfo interface.

2.2.6.1 TestInfoTest

TestInfoTest shows some uses of the TestInfo interface. It is how to get some goodies into your tests.

2.2.6.2 TestReporterTest

TestReporterTest shows how to get information out of the test.

2.2.7 Repeated Tests

b090-repeated-tests-maven

The next really cool thing JUnit5 give us is the ability to repeat a test a specified number of times using the @RepeatedTest annotation. For these tests, you can customize the display name and inject the RepetitionInfo interface into your methods for repetition information.

The class RepeatedTestsTest shows some uses of repeated tests.

2.2.8 Parameterized Tests

b100-parameterized-tests-maven

Parameterized tests are currently an experimental feature, but they seem to work as expected. They are similar to Spock's @Unroll/where data table mechanism. The main difference is that in JUnit5, the data are handled as injected dependencies.

There are three basic flavors to these tests, what I call co-located data, stream data, and external data tests. Let's take a look.

2.2.8.1 Co-located Data Test

The ParameterizedTestsFromGuideTest class mainly consists of internal data tests. The data driving the tests are located with the tests themselves. I took the tests from the JUnit 5 Users Guide because they demonstrate parameterized tests more clearly than my original examples.

The method testWithValueSource shows the basics of the tests. You tag the test as @Parameterized, you add a @ValueSource annotation containing the data, and you include the data type and parameter name in the method signature. Reminds me of how Spring 5 handles inject data from web pages in @Controller classes.

You can use enumerators with or without exclusions or inclusions – we'll look at that in a minute – arrays of most primitives or Strings. You can also use a class to supply data, similar to Spring's @Configuration classes.

--- demo ---

This is a nice tool to avoid DRY code, but it has two major flaws: the allows types of arguments is limited, and only one parameter is allowed.

But JUnit5 does not leave you high and dry! Using the @CsvSource annotation, you can supply multiple argument using a list of comma-separated values.

2.2.8.2 Stream Data Tests

But all is not lost! JUnit5 can use any Java stream as a value source via the @MethodSource. This annotation allows you to refer to a static factory method to supply the test data. This can be an inner class or an external classes. The factory method simply needs to generate a stream of data for the test method to consume.

Method testWithExplicitLocalMethodSource tests a stream of strings.

Method testWithIntRangeMethodSource returns a stream of int primitives.

Stream-based parameterized tests can return multiple parameters: they just need to return a collection, stream, or array of Arguments instances, or object arrays.

Method testWithMultiArgMethodSource show multiple parameters passed by a stream of Arguments.

2.2.8.3 External Data Tests

Both of the previous flavors of parameterized tests used data located in the same class as the tests. A third type of parameterized tests uses external classes or files to supply the data.

2.2.8.4 Additional Features

There is another provider called an @ArgumentSource. It is an interface that allows you to create a custom, reusable data factories.

JUnit5 providers automatic conversion to simplify primitive boxing and unboxing, numeric conversion, and other conversions to simplify passing in data.

2.3 Additional JUnit5 Features Not Covered

- Enhanced handling for time-out conditions.
- Creating your own ParameterResolver.
- Test Interfaces: for adding default methods to test classes by implementing an interface.
- Test Templates: a generalization of repeated and parameterized tests.
- Dynamic Tests: tests created at runtime by factories. A generalization of Assumptions.
- Parallel Execution: tests running in parallel.
- Extension Model: plugins that add abilities to JUnit.
- Platform API: build your own launchers and engines

3 Summary

3.1 Advantages of JUnit5

JUnit5 brings testing into the Java 8+ ecosystem with a low learning curve for experienced JUnit testers. Streams, lambdas, and other modern features are now available for use, still in a single language: Java. And it does this simply and in a straightforward manner.

Other testing platforms are fine: in my opinion, Spock is comparable in power and flexibility. However, Spock requires Groovy and other dependencies, plus a knowledge of Groovy to take advantage of its power. JUnit5 is Java.

3.2 Power

JUnit5 has many advanced features. My top five are:

1. Single test methods to handle multiple scenarios through repeated and parameterized tests.
2. Better control over displayed methods.
3. More sophisticated error handling.
4. Enhanced execution control.
5. Advanced use of lambdas and streams.

The enhanced features help build tests better and faster, making them cheaper to develop.

3.3 Final Thoughts

Even with the increased power of automated testing, we still very much need effective QA teams. No matter how well we test, we will always miss something. It helps to have another set of eyes on our code with the focus of functional quality: does the application perform the function it is intended to? The mission of a QA team is to catch functional problems before they reach the customers.

QA teams are the coders' best friends. With diligent automated testing, we programmers and engineers can catch the simple stuff, making their lives easier. And who wouldn't want to help out their best friend?

The quality of applications should never be strained. JUnit5 can make the lifting a little easier.

4 Questions

5 References

https://docs.gradle.org/4.6/release-notes.html

https://junit.org/junit5/docs/current/user-guide/

https://maven.apache.org/surefire/maven-surefire-plugin/examples/junit-platform.html