# JUnit 5 Hands on Lab

JUnit 5 Hands on Lab 1

Reference 2

Step 0 Install tools 2

Step 1 Baseline 3

Step 1.1 – Clone latest code and instructions 3

Step 1.2 – Run project with Maven 4

Step 2 – Convert to JUnit 5 5

Step 2.1 – Update pom.xml 5

Step 2.2 – Run Maven build 6

Step 2.3 – Run unit tests in IDE 6

Step 3 – Convert basic tests to JUnit 5 7

Step 3.1 – Update the FishermansWharfTest 7

Step 3.2 – Update the SeaLionTest 8

Step 3.3 – Migrate assertThat 8

Step 4 – Migrate Parameterized Tests 9

Step 4.1 – Migrate FishermansWharfEnumTest 9

Step 4.2 – Migrate ThirtyNinthAnniversaryEventsTest 9

Step 4.3 – Refactor this test to use a ValueSource 10

Step 4.4 – Refactor this test to use a MethodSource 10

Step 5 – Migrate more features from JUnit 4 11

Step 5.1 – Migrate Soft Assertions 11

Step 5.2 – Migrate Mockito code 12

Step 5.3 – Migrate tests with an expected exception 12

Step 5.4 – Migrate tests with a timeout 13

Step 5.5 – Skip tests 13

Step 6 – Remove JUnit 4 support from the pom.xml 15

Step 7 – Playing with some new features 16

Step 7.1 – Using repeating tests 16

Step 7.2 – Displaying names 16

Step 7.3 – Bonus if time: write a test from scratch (putting it together) 17

## Reference

If at any point during this lab, you want to explore another feature, see the

<http://junit.org/junit5/docs/current/user-guide>.

## Step 0 Install tools

The required software is described at

<https://www.selikoff.net/2017/09/18/setup-for-hol-1695-starting-out-with-junit-5-at-javaone/>

## Step 1 Baseline

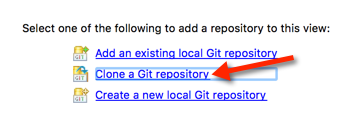
In this lab, we are going to be new developers on the san-francisco-tourism project. The project currently uses JUnit 4. The goal of step 1 is to confirm we have a known good state to start with.

### Step 1.1 – Clone latest code and instructions

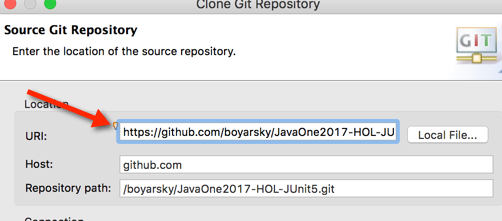
Clearly, the first thing we have to do is pull the latest code for the project! The repository is <https://github.com/boyarsky/JavaOne2017-HOL-JUnit5.git>

**Using Eclipse:**

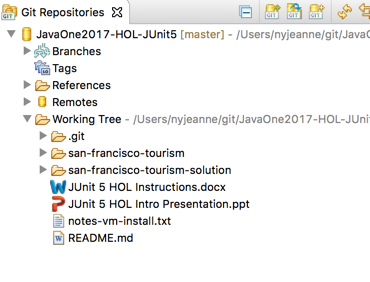
1. Change to Git perspective
2. Click “Clone Git repository”



1. Paste <https://github.com/boyarsky/JavaOne2017-HOL-JUnit5.git> into URL field



1. Click next/next/finish including all defaults
2. Expand “JavaOne2017-HOL-JUnit5”
3. Expand “Working Tree”



1. Right click “san-francisco-tourism”
2. Choose “Import projects”
3. Click Finish (this will create the project in Eclipse)
4. Switch back to Java perspective

**Using IntelliJ:**

1. VCS > Git > clone
2. <https://github.com/boyarsky/JavaOne2017-HOL-JUnit5.git>
3. Specify project name san-francisco-tourism

If you aren’t familiar with IntelliJ, I recommend not using it for this lab. I have a lot more experience with Eclipse and Maven so will be able to help you better ☺.

**Using Git Bash:**

git clone <https://github.com/boyarsky/JavaOne2017-HOL-JUnit5.git>

### Step 1.2 – Run project with Maven

This project currently uses JUnit 4. We want to make sure that we can run builds now before changing anything. After all, we want a known good state before writing code. This lab supports Eclipse, IntelliJ and standalone Maven so you can choose which you prefer.

**Eclipse**

1. Right click the project “san-francisco-tourism” and choose Run as > Maven Install
2. After it runs, check the output has “BUILD SUCCESS” near the end

**IntelliJ**

1. In menu bar, select Run > Run… > Maven
2. After it runs, check the output has “BUILD SUCCESS” near the end

**Maven**

1. Open a command line (Unix, DOS, PowerShell, Cygwin or Git Bash)
2. cd to your project directory
3. mvn install
4. After it runs, check the output has “BUILD SUCCESS” near the end

## Step 2 – Convert to JUnit 5

JUnit 5 is available and we want to use it. The goal of step 2 is run all the JUnit 4 style tests using a JUnit 5 runner. This is the first step in migration on a real project. Once all the plumbing is ready, it will be time to actually use JUnit 5 syntax.

### Step 2.1 – Update pom.xml

The following walk you through updating the pom.xml to use JUnit 5 with legacy JUnit 4 support. If you aren’t familiar with Maven (or aren’t sure what the updated pom.xml should look like based on these steps), feel free to look at the [solution guide pom.xml](https://github.com/boyarsky/JavaOne2017-HOL-JUnit5/blob/master/san-francisco-tourism-solution/pom.xml).

1. Add property to pom.xml so can run unit tests.

<surefire.version>2.19.1</surefire.version>

1. Add properties to pom.xml to pull JUnit. The first two are always required. The third is so you can run JUnit 5 tests.

<junit.jupiter.version>5.0.0</junit.jupiter.version>

<junit.platform.version>1.0.0</junit.platform.version>

<junit.vintage.version>4.12.0</junit.vintage.version>

1. Add the Surefire plugin to the <build> section so that a version of Surefire that works with JUnit 5 is used. (2.20 does not work as of this time so using 2.19).

*Note that this lab doesn’t have any integration tests. If your project does, you’ll need to do the same with the maven-failsafe-plugin.*

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>${surefire.version}</version>

<dependencies>

<dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-surefire-provider</artifactId>

<version>${junit.platform.version}</version>

</dependency>

</dependencies>

</plugin>

1. Add the following to the <dependencies> section to pull all required JUnit jars:

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>${junit.jupiter.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>${junit.jupiter.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-launcher</artifactId>

<version>${junit.platform.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>${junit.jupiter.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

<version>${junit.vintage.version}</version>

<scope>test</scope>

</dependency>

### Step 2.2 – Run Maven build

1. See step 1.2 for a review of how to run a Maven build.
2. Search in the output console for “surefire” and look at how many tests were run. If you got a non-zero answer, you updated the POM properly.

### Step 2.3 – Run unit tests in IDE

If you are choosing to use the Maven command line for this lab, skip this step and continue to step 3. For those using an IDE, the output is more interesting in the IDE view. Especially with parameterized tests.

**Eclipse**

1. Right click san-francisco-tourism project
2. Run As > JUnit test

**Intellij**

1. In menu bar, click Run >
2. Run…
3. All in san-francisco-tourism

Note how many tests were run in the console. It should also match.

## Step 3 – Convert basic tests to JUnit 5

Now that we’ve seen JUnit 5 can run JUnit 4 tests, it is time to convert tests to JUnit 5.

### Step 3.1 – Update the FishermansWharfTest

This test uses the most basic features of JUnit; literally just a setup method and a few assertions.

1. Change the static import to:

**import** **static** org.junit.jupiter.api.Assertions.\*;

Notice that there is a new package naming convention for JUnit 5 containing “jupiter.” Having a new package name allows running existing tests without wholesale changes. Also, note that the **Assert** class has been renamed to **Assertions**.

1. Change the regular import to follow the new naming convention as well so JUnit 5 can find the annotations:

**import** org.junit.jupiter.api.\*;

1. Speaking of annotations, the @Before annotation has been renamed to @BeforeEach so change that too.
2. One more compiler error to fix. In JUnit 5, the String message parameter is the last parameter instead of the first one. Just fix the assertion in oldestSeaLionForLastMessage() to make the code compile:

*assertEquals*(15, actual.getAge(), "oldest");

1. Now that the test compile, run the tests again. You should get one failing test. If you do not have a failing test, change your run configuration in the IDE to use JUnit 5. For example, in Eclipse, Run > Run Configuration… and change the test runner to JUnit 5:



1. Notice that the AssertionFailedError comes from opentest4j; which got pulled in as a transitive dependency. Also note that there is a bunch of Java 8 stack trace below the code that actually caused the error.



1. Now you can fix the failing test by changing the parameter order:

*assertEquals*("http://www.fishermanswharf.org", wharf.getUrl(), "url");

1. Run the test again and you get a green bar.

### Step 3.2 – Update the SeaLionTest

Your turn. Try to migrate the SeaLionTest class to JUnit 5. There’s only one thing that we haven’t seen yet. The @BeforeClass annotation has been renamed to @BeforeAll. I bet you can guess what the @AfterClass annotation was renamed to!

This time there aren’t step by step instructions because you already have all the information you need to convert it. Feel free to check out the solution guide if you aren’t sure.

*Note for real life projects: I wanted my codebase to be fully JUnit 5 so I didn’t have to look at the imports to read an assertion. Changing assertions, imports and annotations by hand got old fast so I wrote a tool to automate it:*

[*https://github.com/boyarsky/convert-junit4-to-junit5*](https://github.com/boyarsky/convert-junit4-to-junit5)

### Step 3.3 – Migrate assertThat

Let’s try to migrate FishermansWharfEnumTest. Try doing the same thing as for SeaLionTest. Uh oh. There’s no assertThat in Jupiter.

Now try it with the correct import:

**import** **static** org.hamcrest.MatcherAssert.*assertThat*;

That’s right. The *assertThat* method is no longer in core JUnit. It still works perfectly well though; just from inside Hamcrest.

This isn’t a great test though. If would only fail on the first enum value to fail the assert. It would have been far better to use a parameterized test. It was enough work in JUnit 4 to write a parameterized test. So it is easy to understand why it was easier to write a loop here. Let’s fix that now…

## Step 4 – Migrate Parameterized Tests

### Step 4.1 – Migrate FishermansWharfEnumTest

Let’s convert to a parameterized test and see how easy it is.

1. Copy FishermansWharfEnumTest to FishermansWharfEnumParamTest to preserve the original as a referenc.e
2. Change @Test to @ParameterizedTest. (The pom.xml already has junit-jupiter-params so this works)
3. On the line below this annotation add the following to tell JUnit to call this method once with each enum value as a parameter:

@EnumSource(FishermansWharfEnum.**class**)

1. Add imports to make the code compile:

**import** org.junit.jupiter.params.\*;

**import** org.junit.jupiter.params.provider.\*;

1. Add a method parameter:

FishermansWharfEnum current

1. Remove the for loop so the method body only has two statements:

LocalTime ninePm = LocalTime.*of*(12 + 9, 0);

*assertThat*(current + " should close after 9pm",

current.closes(), *greaterThanOrEqualTo*(ninePm));

1. Run the unit tests and notice how the method was called three times:



1. Optionally, check the Maven console and see that FishermansWharfEnumParam test has three tests in the output.

### Step 4.2 – Migrate ThirtyNinthAnniversaryEventsTest

Now we have a test in which the data was externalized so users could provide test data. The parameterized test reads from a file and builds tests for each of the scenarios. The logic to read the file and create the parameters is ugly, but does work. Let’s fix it, shall we?

1. This time we are going to replace the @Test annotation with an annotation with a parameter so the display name is printed nicely:

@ParameterizedTest(name = "{0}")

1. Now we have to tell JUnit where to find the CSV. It is worth noting that the file name doesn’t matter if the csv formatting is preserved. It is even smart enough to ignore comments! Also worth noticing is that JUnit is looking on the classpath so we don’t specify the directory src/test/resources anymore.

@CsvFileSource(resources = { "/39-tests.txt" })

1. Fix the imports like our previous parameterized test.
2. Next add the two parameters to the test method

LocalDate date, **boolean** expectedResult

1. Finally, delete all the unnecessary code: the @RunWith annotation, the two lines that begin with @Parameter and the entire static method. Feels good to remove that ugly code, doesn’t it. If you did this right, you have seven lines of not counting the imports or lines that only have a bracket on them.
2. Try to run the test. You’ll likely get an error about there not being runnable methods. (I see this inconsistently in both Maven and Eclipse). No worries. Just add a dummy method and try again:

@Test

**public** **void** dummyTestSoRuns() {

// no op

}

### Step 4.3 – Refactor this test to use a ValueSource

Let’s suppose the users decided they don’t need to edit the test file and we don’t need it externalized. Now we can use a value source. A value source lets you pass hard coded values to the parameterized test. This is useful when the values are simple and known in advance. Even better we can have two parameterized test methods – one for the special dates and one for the others.

There’s only one thing that’s tricky: a value source can’t take a LocalDate as a parameter. This means we need to use a String type and do the conversion inside the test.

Give this test a try. If you get stuck, the answer is inside: ThirtyNinthAnniversaryEventsValueSourceTest.

### Step 4.4 – Refactor this test to use a MethodSource

Tired of this test yet? Hope not as we are going to give it one more go! This time we are going to use a method source and actually return the LocalDate elements so you no longer need to manually parse the dates.

Given these two static generator methods, can you figure out how to rewrite the parameterized tests to use them?

**public** **static** List<LocalDate> specialDateGenerator() {

**return** Arrays.*asList*(LocalDate.*of*(2017, 1, 6),

LocalDate.*of*(2017, 2, 17),

LocalDate.*of*(2017, 10, 6),

LocalDate.*of*(2017, 10, 13));

}

**public** **static** Stream<LocalDate> nonSpecialDateGenerator() {

**return** Stream.*of*(LocalDate.*of*(2016, 12, 30),

LocalDate.*of*(2017, 1, 5),

LocalDate.*of*(2017, 2, 16),

LocalDate.*of*(2017, 10, 20));

}

If you get stuck, the answer is in ThirtyNinthAnniversaryEventsMethodSourceTest.

## Step 5 – Migrate more features from JUnit 4

In this step we will finish migrating our JUnit 4 tests.

### Step 5.1 – Migrate Soft Assertions

Ok. So soft assertions aren’t part of JUnit 4. They are part of a separate library called AssertJ. But with JUnit 5 they are built into JUnit so we get to migrate them!

The idea of a soft assertion is that they wait until the group has completed to fail. That way you know how many/which ones fail rather than just the first one. Let’s try to migrate the CableCarsTest class to JUnit 5.

1. Change the imports/annotations to JUnit 5 style. You know how to do that by now.
2. Get rid of the SoftAssert and assertAllLines.
3. Switch to the new built in API. The following example shows two assertions, but you should migrate all four. Note that the String parameter is the first parameter. That’s the header that groups them. Also note that Java 8 syntax (lambdas) is used so we can pass the assertions to be executed at runtime.

*assertAll*("lines",

() -> *assertTrue*(CableCars.*isLine*("California")),

() -> *assertTrue*(CableCars.*isLine*("Powell-Hyde"))

);

1. Go into the pom.xml and delete the AssertJ dependency. We don’t need it anymore.

<dependency>

<groupId>org.jboss.forge.addon</groupId>

<artifactId>assertj</artifactId>

<version>3.6.2</version>

</dependency>

1. Re-run the test to confirm it works.

### Step 5.2 – Migrate Mockito code

Mockito code is really easy to migrate from JUnit 4 to 5. The CableCarWait class uses Mockito. It’s up to date and uses the MockitoJUnitRunner. However, @RunWith is gone in JUnit 5. So let’s fix the code for CableCarWaitTest.

1. Guess what. The first step is to change the imports/annotations to JUnit 5 style. You might notice an odd looking import. It does tell you who contributed the code!
2. There is only one line of code that doesn’t compile. Replace the Runner line with: @ExtendWith(MockitoExtension.**class**)
3. The code still doesn’t compile because you don’t have MockitoJUnitRunner yet. No worries. Let’s add it to the pom.xml now:

<dependency>

<groupId>name.falgout.jeffrey.testing.junit5</groupId>

<artifactId>mockito-extension</artifactId>

<version>1.0.0-RC2</version>

</dependency>

1. Fix the imports and re-run the test to confirm it works.

Note: This is an opportunity. If you are using Spring and Mockito together, you haven’t been able to use the MockitoJUnitRunnner as only one @RunWith was allowed. JUnit 5 allows multiple @ExtendWith so you can now use both!

### Step 5.3 – Migrate tests with an expected exception

In JUnit 4, there were a variety of approaches for checking if an expected exception was thrown. Two of the three no longer work in JUnit , but there is a better way.

1. In EarthquakeTest, change the imports/annotations to JUnit 5. Remember you need a special import for assertThat().

If you’ve done this step correctly, the only compiler errors are in the rule annotation/instance variable, the noMessageChecking() test and the usingRule() test.

1. Now you know which two don’t work in JUnit 5. This is actually good as the code was harder to read with @Rule since it was split up. And the attribute encouraged ignoring the exception message. Say goodbye to them ☺.
2. Delete both tests containing code that doesn’t compile and run the remaining test. You should have passing tests.
3. The remaining test works, but can be rewritten in two statements without a try/catch. Notice the lambda expression that makes this possible. The first statement follows. The other asserts the message like before the changes.

ShakeException actual =

*assertThrows*(ShakeException.**class**, () -> earthquake.shake(**true**));

### Step 5.4 – Migrate tests with a timeout

There’s another attribute that was supported on @Test. It was less common, but it is also gone in JUnit 5 in favor of a better way.

1. Run the EarthquakeTimeoutTest and note how long it took to run.
2. Migrate the import/annotations to JUnit 5. Good at this yet? If you did it right, you have exactly one compiler error.
3. Figure out how many seconds the timeout is currently set to.
4. Remove the timeout parameter and parenthesis.
5. Replace the method body with:

*assertTimeout*(Duration.ofMillis(*6000)*,

() -> earthquake.waitForAftershock());

1. Run the test and confirm it took approximately the same amount of time as before you made changes.
2. Now we can improve the test. Change Duration.ofMillis(6000) to a duration that uses seconds so the code is easier to read.

### Step 5.5 – Skip tests

Time for an easy one. JUnit 5 changes the @Ignore annotation to @Disabled.

1. Add a System.out.println to ShakeExceptionTest. Does it run?
2. Convert the test to JUnit 5. You know everything you need to do this.
3. Run it again. Is there any output? (good to see the behavior remains the same)
4. Now try running your JUnit 5 test with @org.junit.Ignore. What happens? (This is why you can’t mix and match.)
5. Now change it back to disabled or delete the @Ignore altogether.

## Step 6 – Remove JUnit 4 support from the pom.xml

Now that we’ve gotten rid of all the JUnit 4 syntax, we should remove JUnit 4 support from our project. That way nobody will be tempted to add more old code. Plus, we won’t be tempted by seeing the old imports.

1. Remove the junit.vintage.engine dependency.
2. Remove the junit.vintage.version property.
3. Remove the junit (4.12) dependency.
4. Re-run the Maven build and watch it succeed.

If you get any failures, you probably missed an import on a prior step so go back and do that now.

## Step 7 – Playing with some new features

JUnit 5 has a number of new features. Let’s try some of them out. This section intentionally gives you less detail about what to type. Feel free to go back to the previous sections if you don’t remember the syntax. Also, there are multiple correct solutions. One of them is in the solution project.

### Step 7.1 – Using repeating tests

Sometimes it is useful to run the same exact test multiple times. For example, when testing multithreaded code, you want to make sure the same result occurs each team. We don’t have multi-threaded code here, but let’s write such a test anyway.

1. Create a test named ThirtyNinthAnniversaryEventsRepeatingTest
2. Create an empty method with the signature:

@RepeatedTest(value=100)

**void** tooEarly() {

}

1. This will run an empty test 100 times. Try it out and watch the number of tests go up.
2. Now implement the method so that it calls ThirtyNinthAnniversaryEvents.*isCelebrationDay*(date) with a random date in the year 2016 once per repetition and asserts that the result is false.
3. This is not a good use case for a repeated test because there is a better way. Do you know why?
4. Create another method called stillTooEarly() but this time use a @MethodSource. (if you still don’t know which type of test would be better, search this file for MethodSource)
5. Run the tests in this class again. Confirm there are 200 tests in the output and that the names of the tests from the method source are clearly.

### Step 7.2 – Displaying names

You can display any string instead of the class or method name. Let’s give this a shot.

1. Open ThirtyNinthAnniversaryEventsRepeatingTest
2. Right before the class definition add:

@DisplayName("Repeating vs Parameterized")

1. Right before the Repeating test, add an annotation with another name.
2. Run the test and see the nice English names.
3. If you speak another language, try adding a character from that language.
4. For the parameterized test try using this String. (You might get prompted to save in a different encoding. Say ok if you do). What gets output?

"π is fun. Even \uD83D\uDE38 like π."

### Step 7.3 – Bonus if time: write a test from scratch (putting it together)

We’ve converted a lot of tests, but now let’s write a test from scratch. My solution is in TestFromScratch

Can you write a test class in JUnit 5 that:

1. Before the any of the tests are run, code an assumption the test is running on whatever operating system you are using. For example, if you are on a Mac, check that it is running on a Mac. Hint: System.getProperty(). Ensure this code is only run once.
2. Has a test that confirms first 20 primes starting with 5 are odd.
   1. If you feel comfortable with streams, do not use a loop anywhere in the program.
   2. If you do not feel comfortable with streams, make sure the test names are the prime numbers. For example, the first test should display as “5”, the second should display as “7”, etc.
3. Has a test that confirms Thread.sleep(1) does not take more than a half second to run and confirms this fact 20 times. (Do not use a loop)
4. Have a test that has an empty body and a display name that uses the String: “Good job, now go have a \uD83C\uDF7A”.
5. If you did this right, you have 41 tests and a fun display message.