# JUnit 5 Hands on Lab

Reference 2

Step 1 Baseline 2

Step 1.1 – pull latest code and instructions 2

Step 1.2 – run project 2

Step 2 – Convert to JUnit 5 3

Step 2.1 – Update pom.xml 3

Step 2.2 – Run Maven build 4

Step 2.3 – Run in IDE 4

Step 3 – Converting basic tests to JUnit 5 5

Step 3.1 – Updating the FishermansWharfTest 5

Step 3.1 – Updating the SeaLionTest 6

Step 4 – Migrating @Test annotation parameters 7

## 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 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 – pull latest code and instructions

Clearly, the first thing we have to do is pull the latest code for the project! Your teammate (me) helped by placing an older version of the project to test connectivity and make it as easy as possible.

1. Double click Git Bash on desktop



1. cd git
2. cd J\*

It should look like:



1. git pull

Wait a minute and you’ll get any updates to this project since the VM was packaged for Oracle to provision to Hands On Lab machines.

There were likely changes to this document after it was created so please use the latest. You can get to it from Windows explorer (in the directory you are currently in) or from <https://github.com/boyarsky/JavaOne2017-HOL-JUnit5>.

### 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. Double click Eclipse on the desktop



1. Eclipse is configured to open into a default workspace with the project checked out.
2. Right click the project “san-francisco-tourism” and choose Run as > Maven Install
3. After it runs, check the output has “BUILD SUCCESS” near the end

**IntelliJ**

1. Double click IntelliJ on the desktop



1. IntelliJ is configured with the project checked out
2. In menu bar, select Run > Run… > Maven
3. After it runs, check the output has “BUILD SUCCESS” near the end

**Maven**

1. Open DOS, PowerShell or Git Bash
2. cd git\Java\*\san-fransico-tourism



1. mvn install
2. 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-RC3</junit.jupiter.version>

<junit.platform.version>1.0.0-RC3</junit.platform.version>

<junit.vintage.version>4.12.0-RC3</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-platformlauncher</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 – Converting 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 – Updating 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. 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 – Updating 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-to-junit5*](https://github.com/boyarsky/convert-junit4-to-to-junit5)

### Step 3.3 – Migrating 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 one that it is easy to understand why that didn’t happen here. Which takes us to…

## Step 4 – Migrating Parameterized Tests

### Step 4.1 – Migrating 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 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 – Migrating ThirtyNinthAnniveraryEventsTest

Now we have a test in which the data was externalized so users could provide it. 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 add 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. 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. Next add the two parameters to the test method

LocalDate date, **boolean** expectedResult

1. Finally, delete all the unnecessary code: the @RunWith annotation and the entire static method. Feels good to remove that ugly code, doesn’t it.
2. Try to run the test. You’ll likely get an error about there not being runnable methods. (I see this inconsitently in both Maven and Eclipse). No worries. Just add a dummy method and try again:

@Test

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

// no op

}

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

Let’s supposed 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. And 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 – Refactoring 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.

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 X – Migrating @Test annotation parameters

TODO write

## Step X – Removing 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.