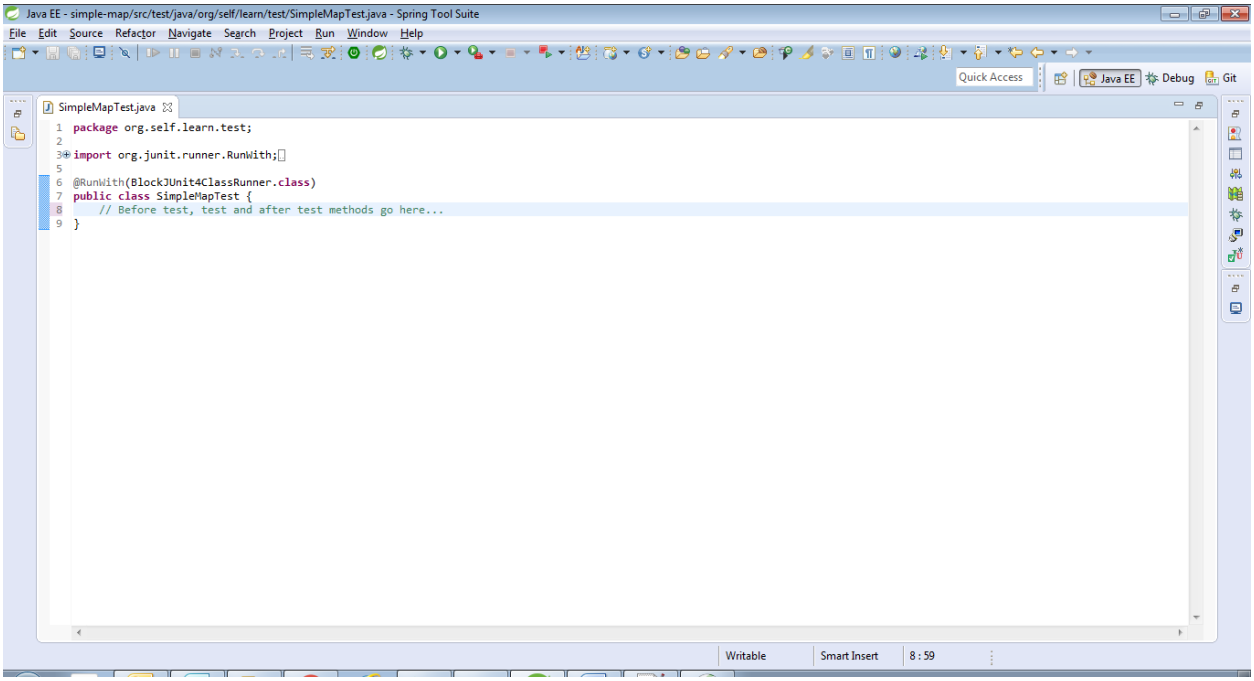
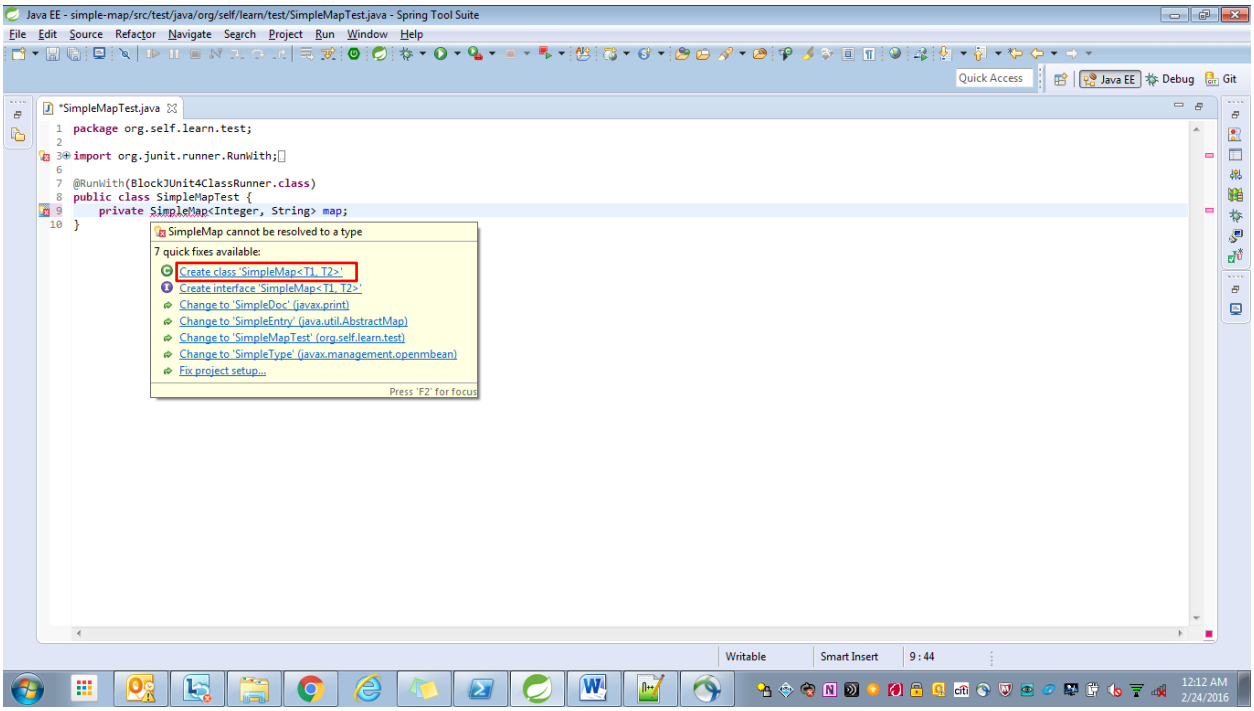
Simple Map without using Java Collections Framework (JCF)

1. Can have Null ‘Key’
2. Can have null ‘values’
3. Cannot contain duplicate ‘keys’
4. May not retain order in which “keys/values” were saved →**No Test Required**
5. Performance is NOT a concern i.e. we will not implement Hash Table like features e.g. bucket, hash values etc. Also redundant iterations are allowed →**No Test Required**
6. Manages size dynamically
7. Throws exception if no such key found
8. Throws exception if duplicate key found
9. Should be able to perform the following operations:
    a. Create Simple Parametrized Map (Generics)
    b. Save ‘key’ and ‘value’ pair
    c. Retrieve value given a ‘key’
    d. Remove a “key-value” pair given a ‘key’
    e. Retrieve size of the map i.e. number of “key-value” pairs
    f. Check if a given ‘key’ is present
    g. Check if a given ‘value’ is present

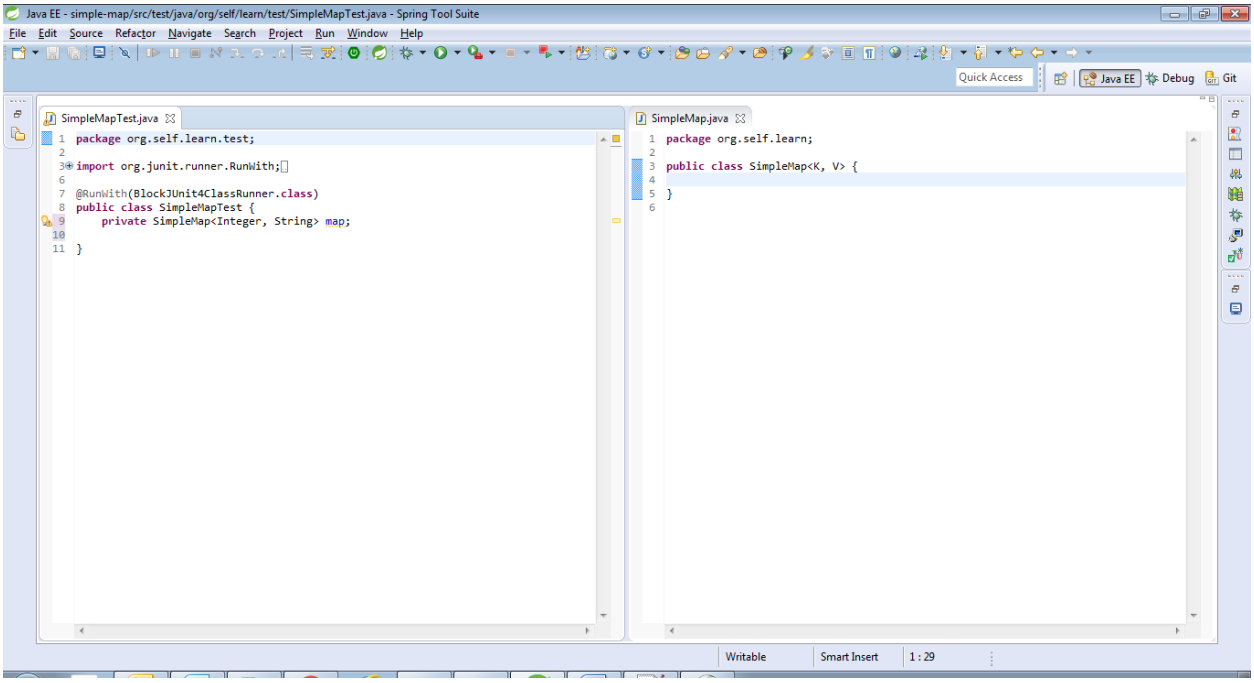
Fact that we are test driving development, test class is crated even before your implementation class. Here is what an empty implementation class looks like:



Let’s start by creating the test class and trying to instantiate the test object. I prefer to use the quick fix to crate the class/methods under test. This, although might sound a bit exaggerated, ensures that our test drives our implementation.

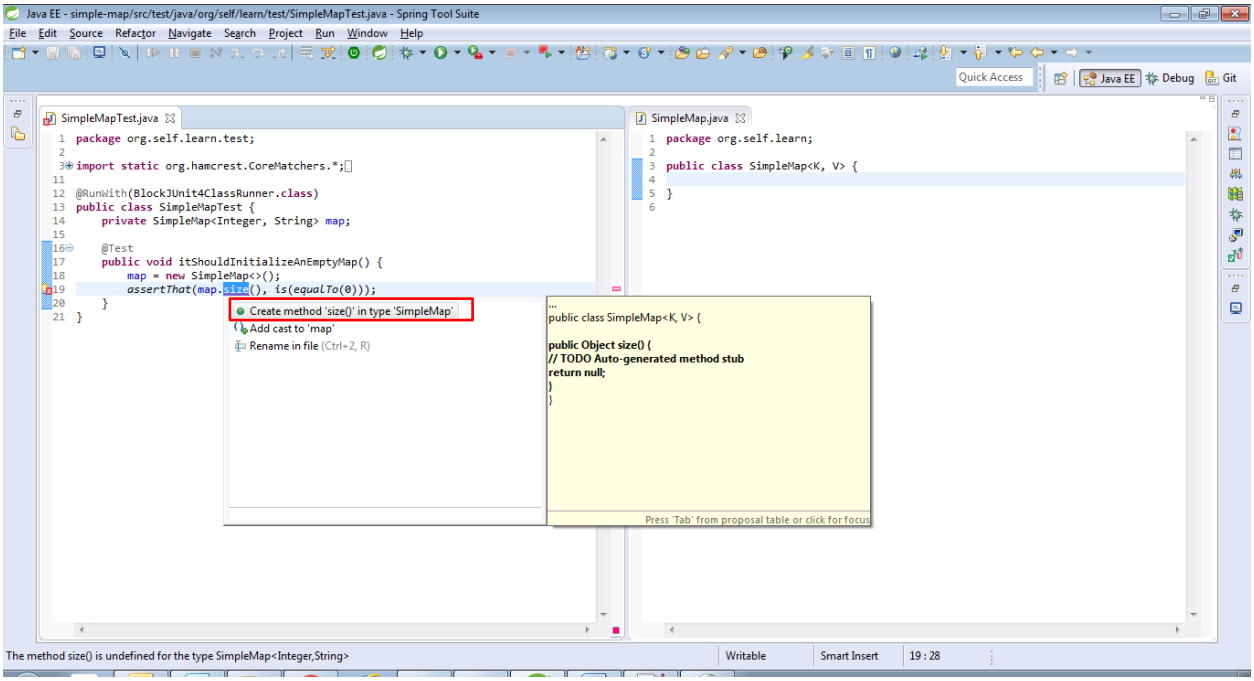


Make note of the eclipse window… this is our preferred style for TDD i.e. having test and implementation class side by side.

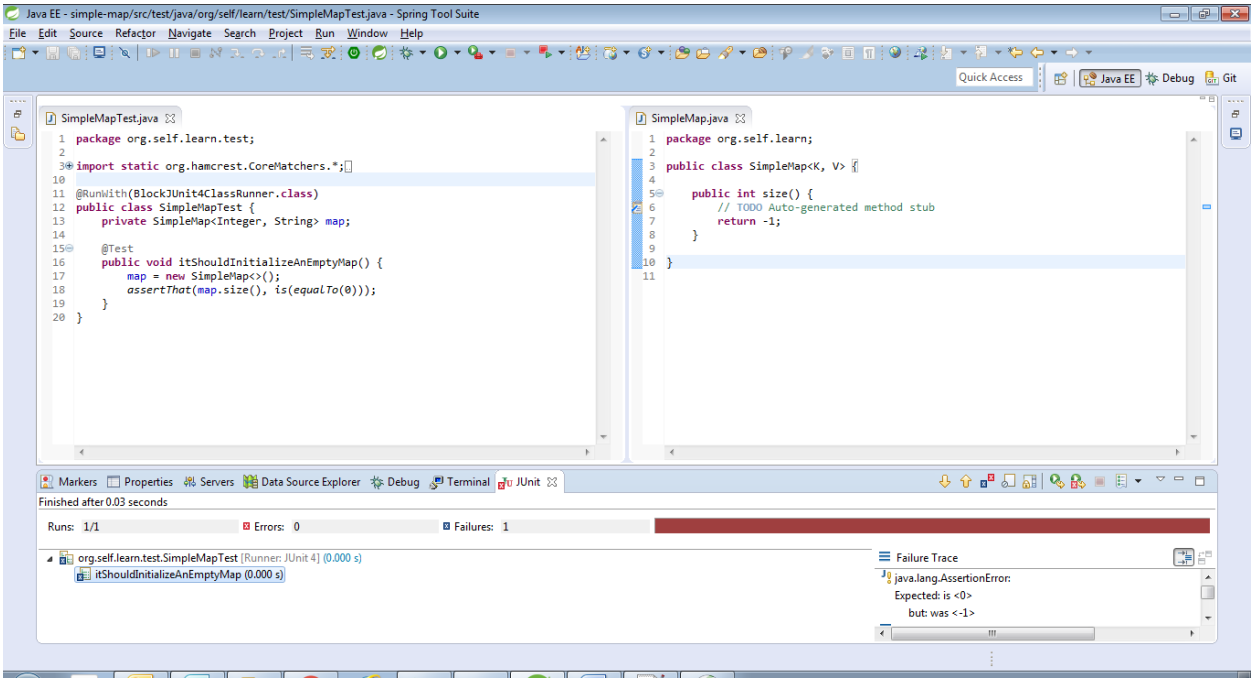


Key idea behind TDD is that you start by test driving that smallest, simplest and tiniest bit and build on top of it. Start with a blank mind without having any preconceived notion of the implementation. Think about what (**and not how!**) would you like your class to do.

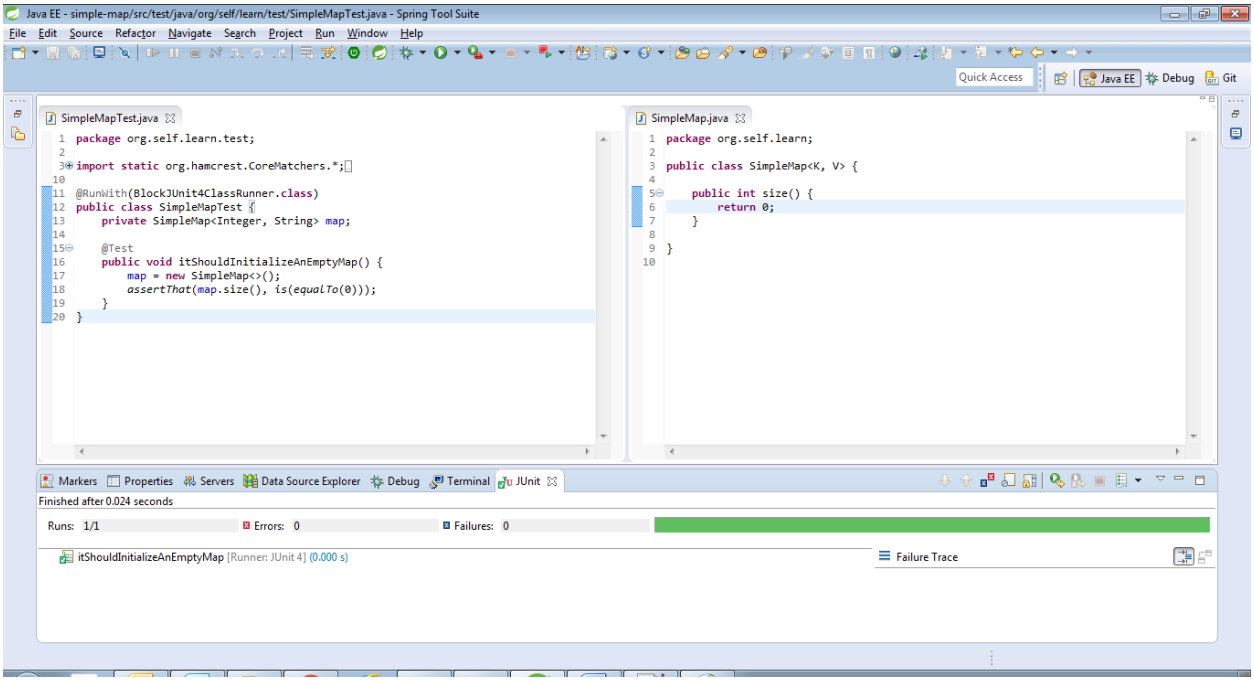
Let's write our first test - "itShouldBeAbleToInitializeAnEmptyMapWithSizeZero()"… Notice how your test tells what your implementation is.



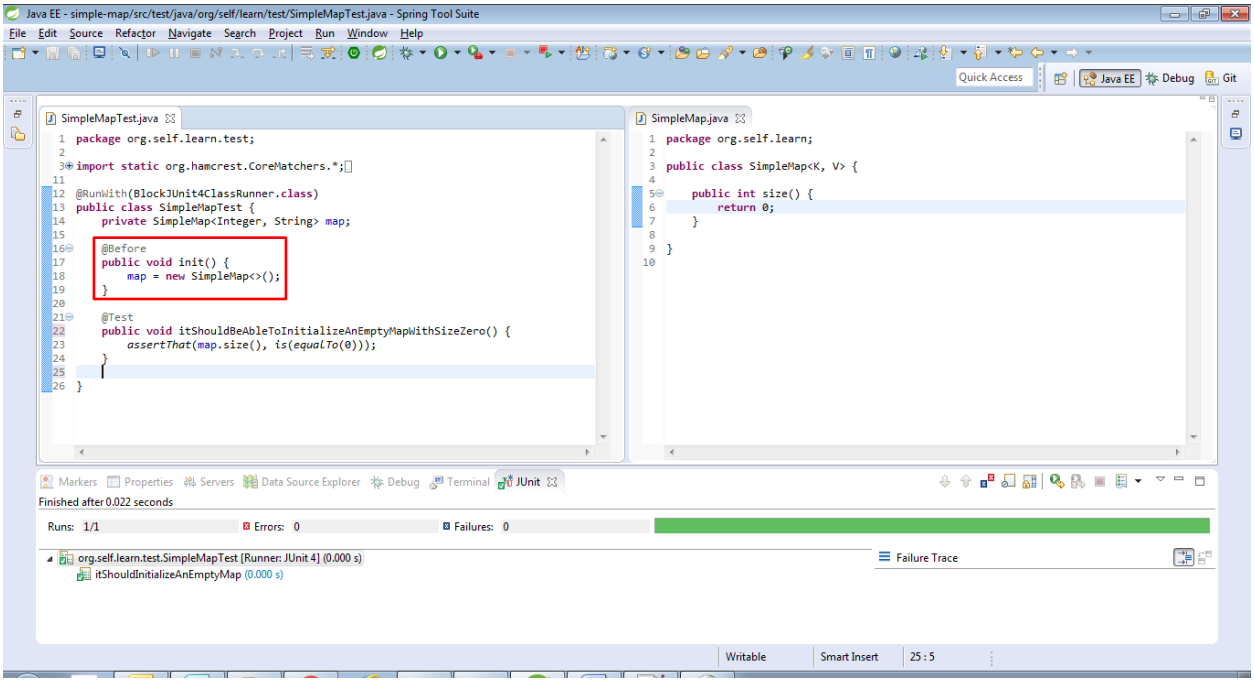
I used quick fix on my test class to crate my implementation again… With that, let's run the test which would obviously fail as we expect it to…



Returning zero takes care of that… (note that the test completes successfully). Idea is to only implement how much the tests force us to… It ensures that you have 100% code coverage.

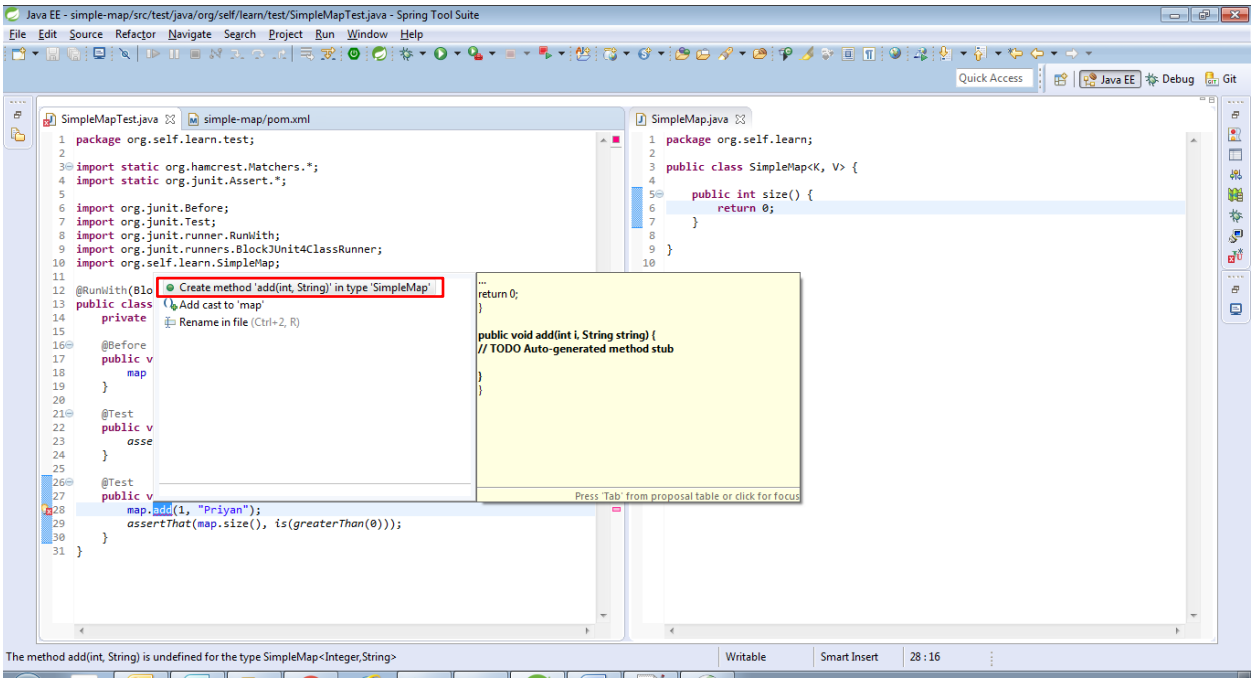


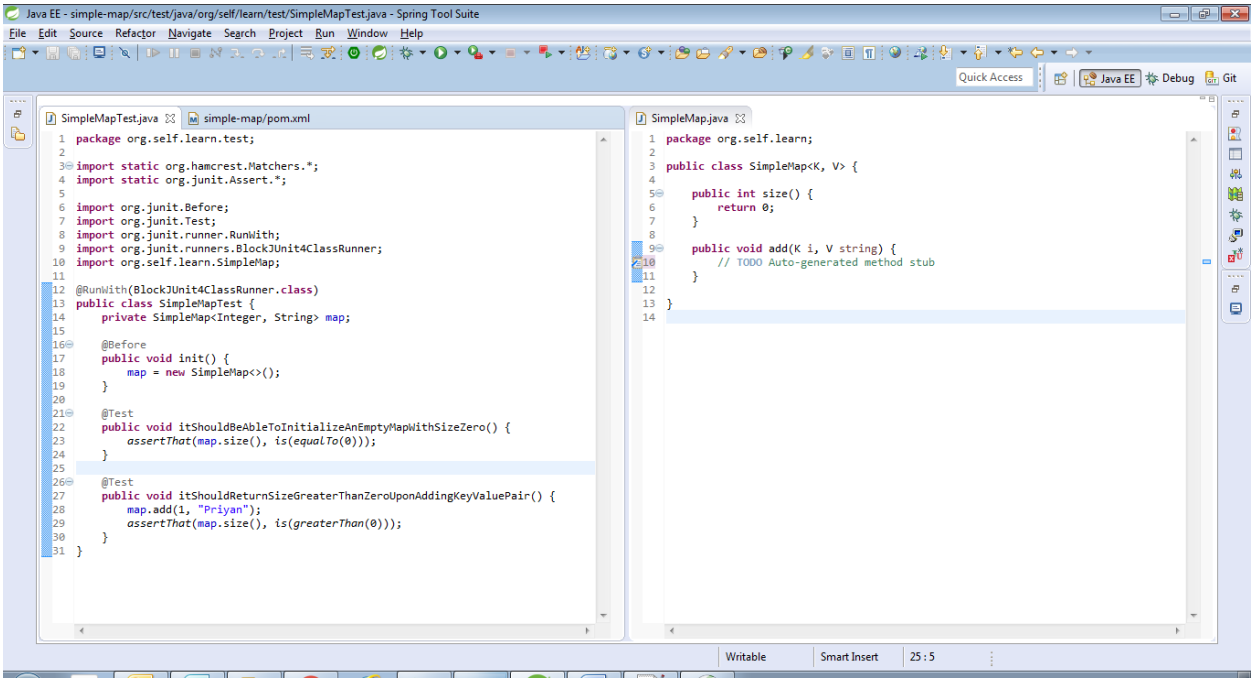
Next I separate the test set up (which in this case is just the test object initialization) into the @Before so the test runs it every time before it runs my tests. I have also refactored the name of my test to make it more explanatory.



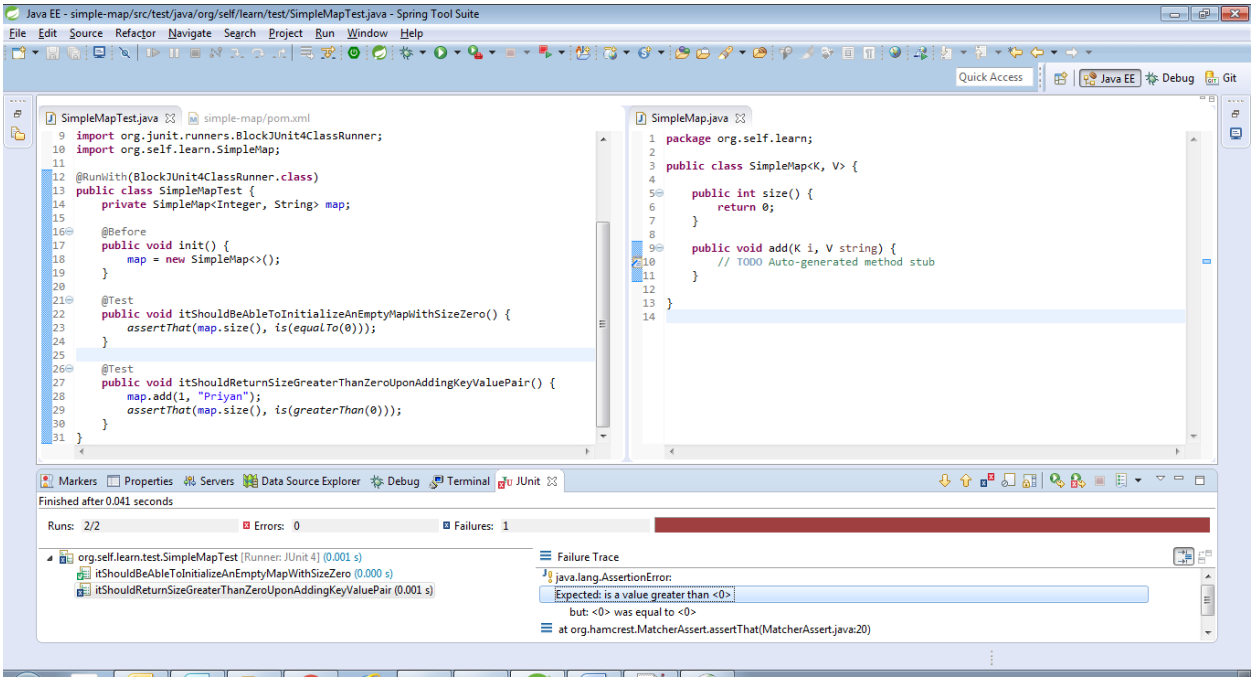
Let's get rolling with the subsequent tests based on a similar approach. The second test is "itShouldReturnSizeGreaterThanZeroUponAddingKeyValuePair()"…

Note again how the test demands implementation…

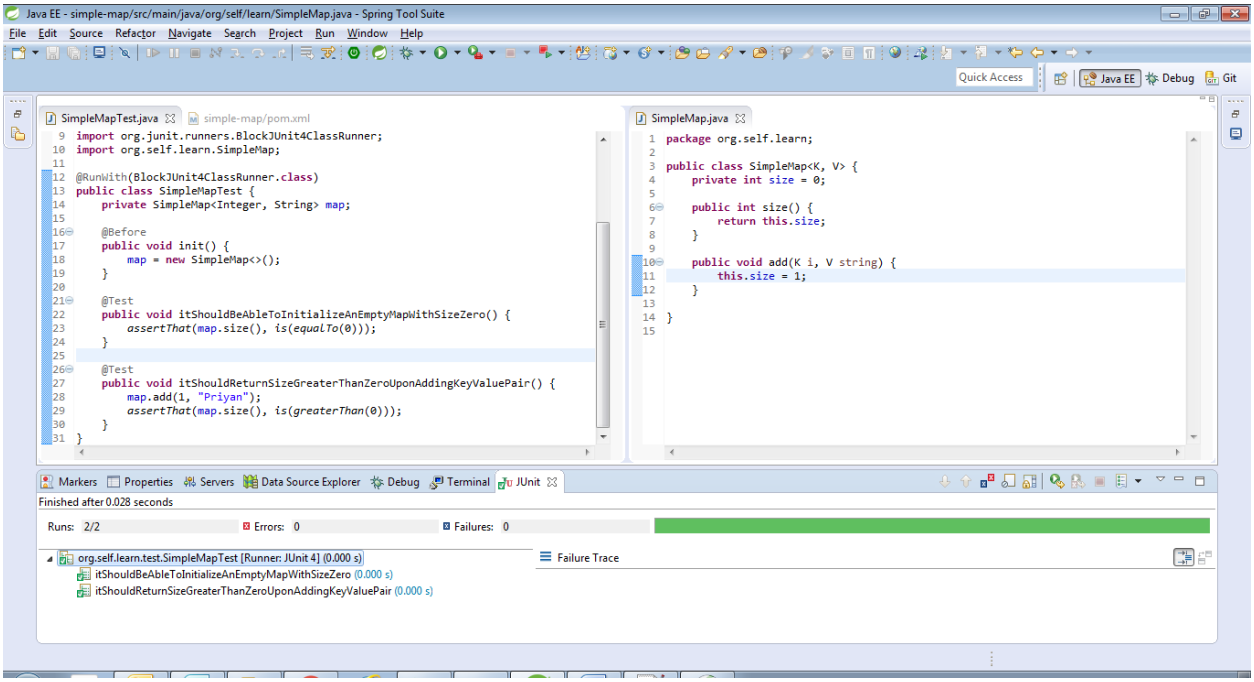




Test fails for obvious reason when ran…

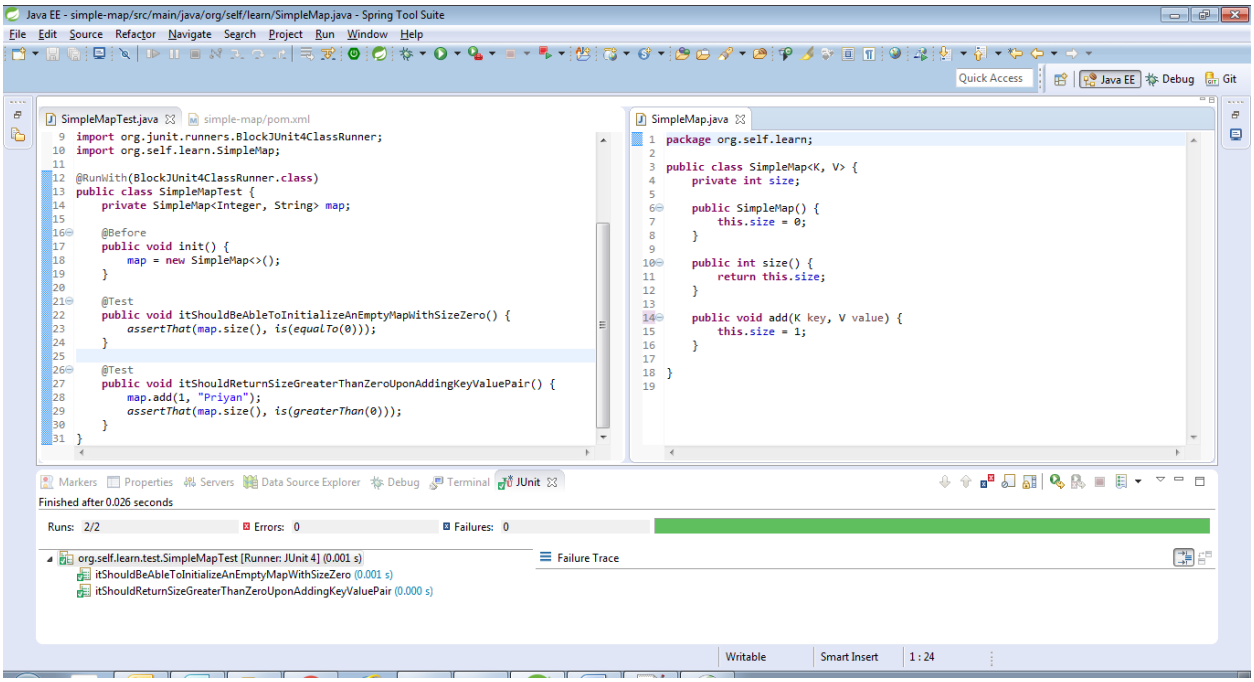


I again did the smallest change that would make my tests run successfully and they do…

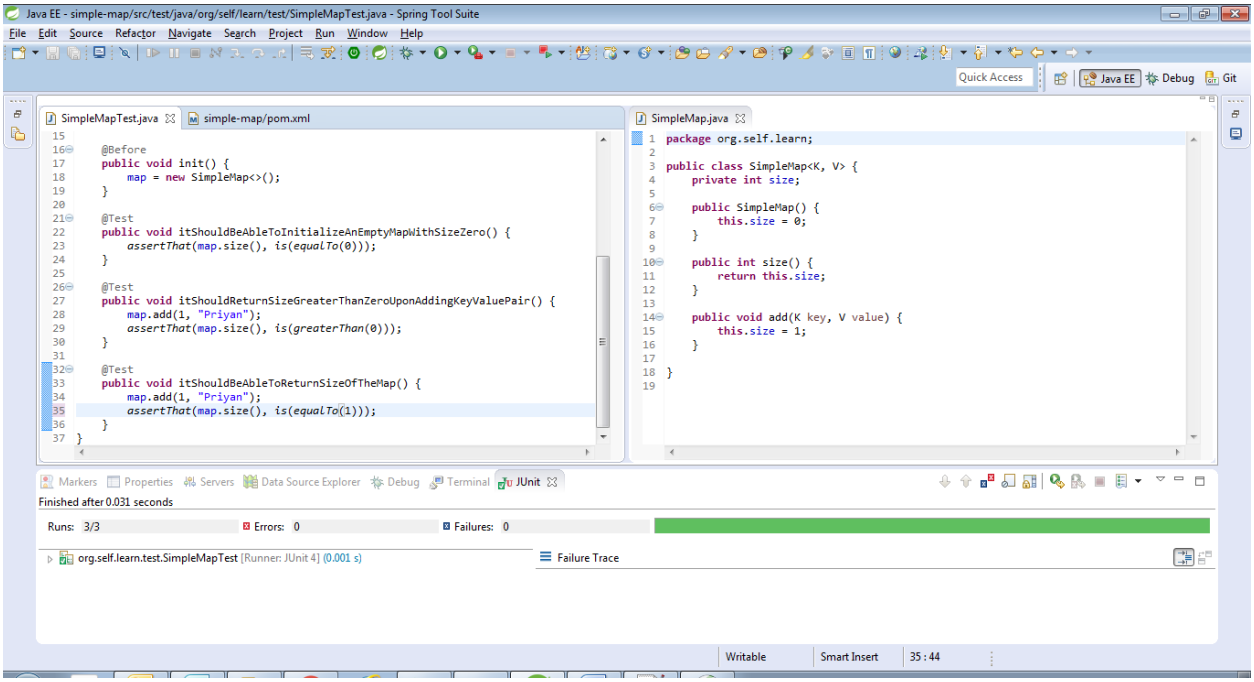


Quick refactor i.e. initialize “int size” inside the constructor. Tests still run fine.

Note: Refactors are anything that improve code quality while not change the behavior of my implementation

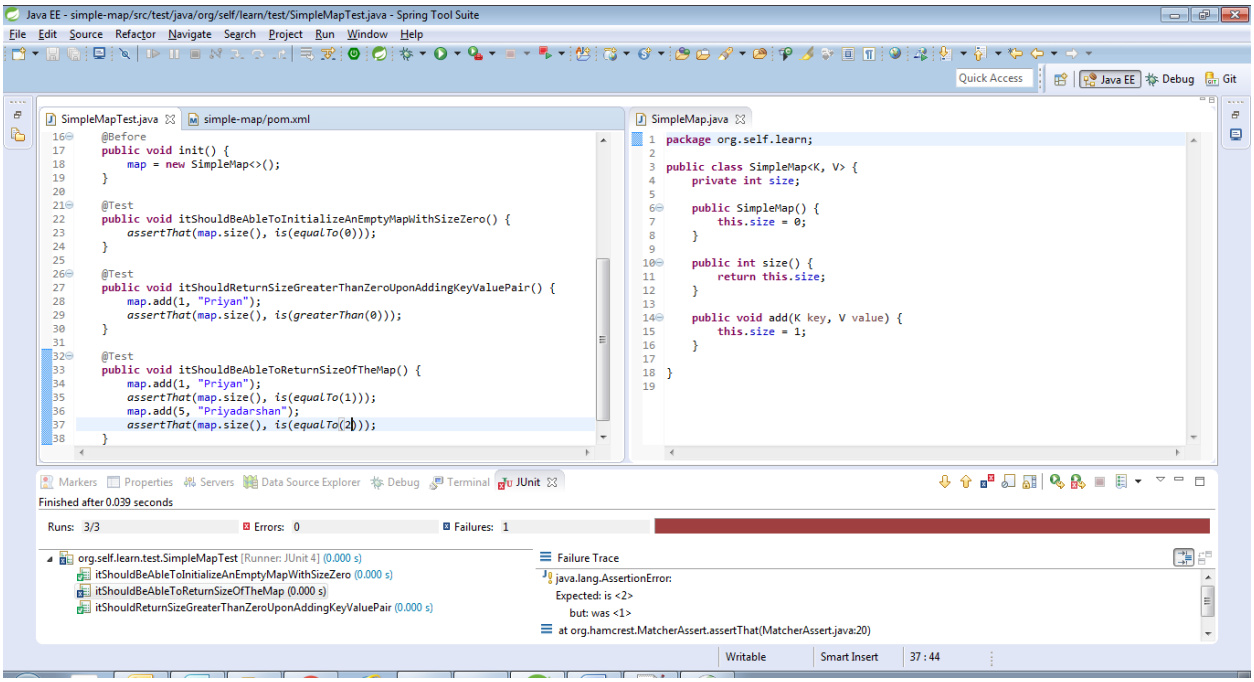


Another “itShouldBeAbleToReturnSizeOfTheMap()” test which the implementation has already taken care though…

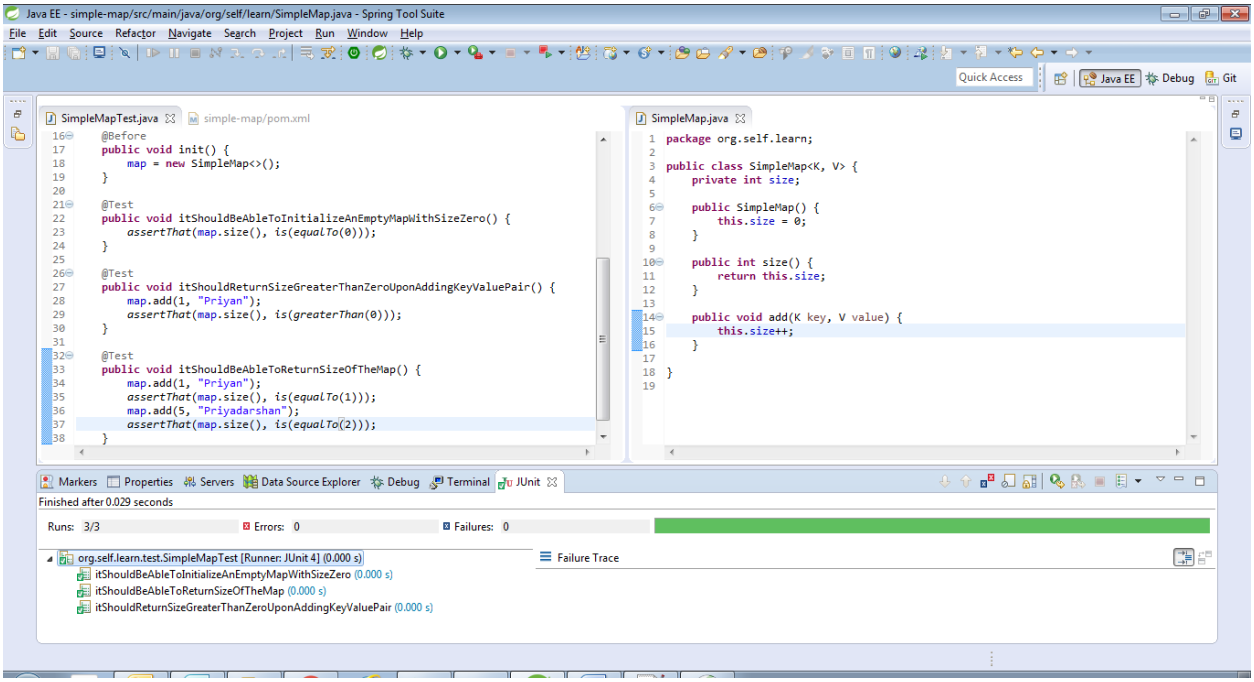


Well, the tests pass giving an illusion of completeness of the implementation. Let's find out if that's really the case though, by adding another test case…

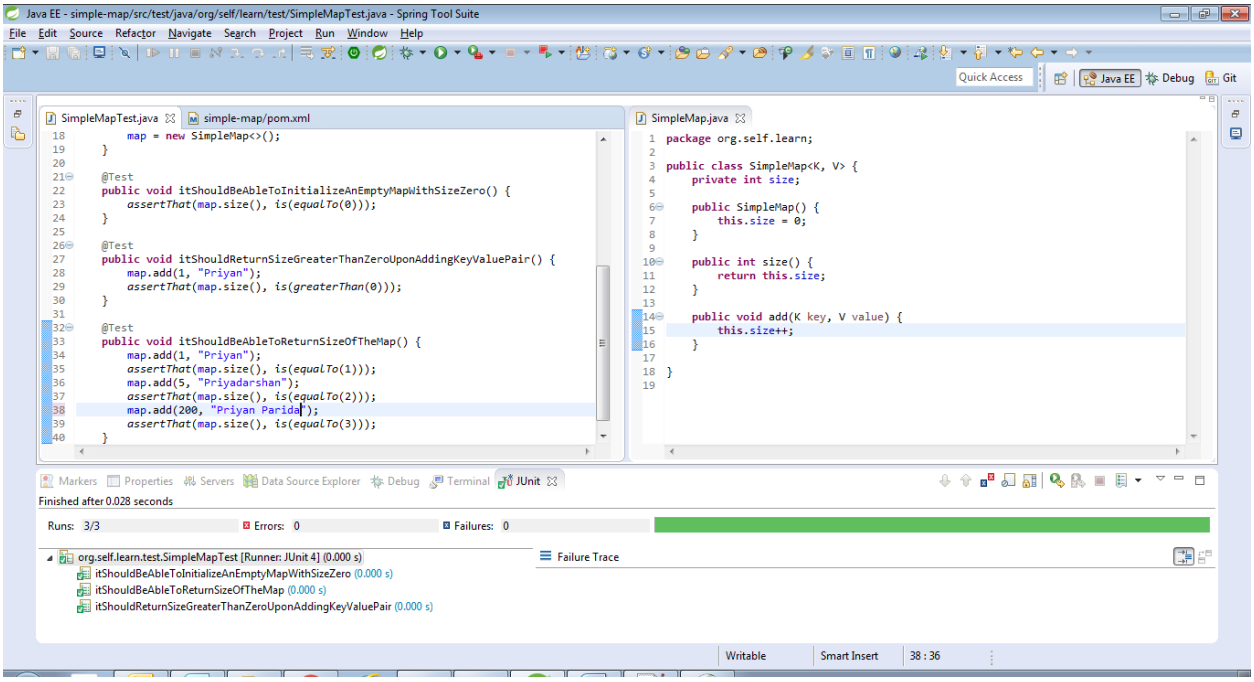
The test fails this time…



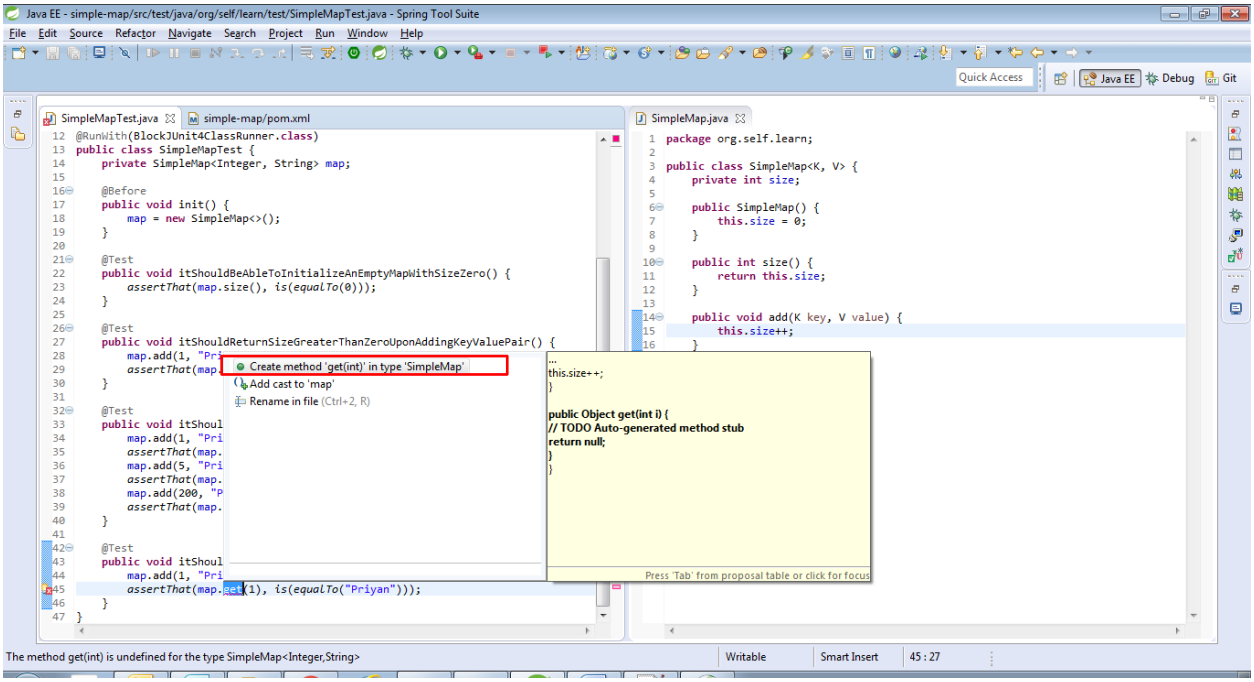
Let's take the next leap to address the test for size of the simple map. And fix does work, as the test confirms.



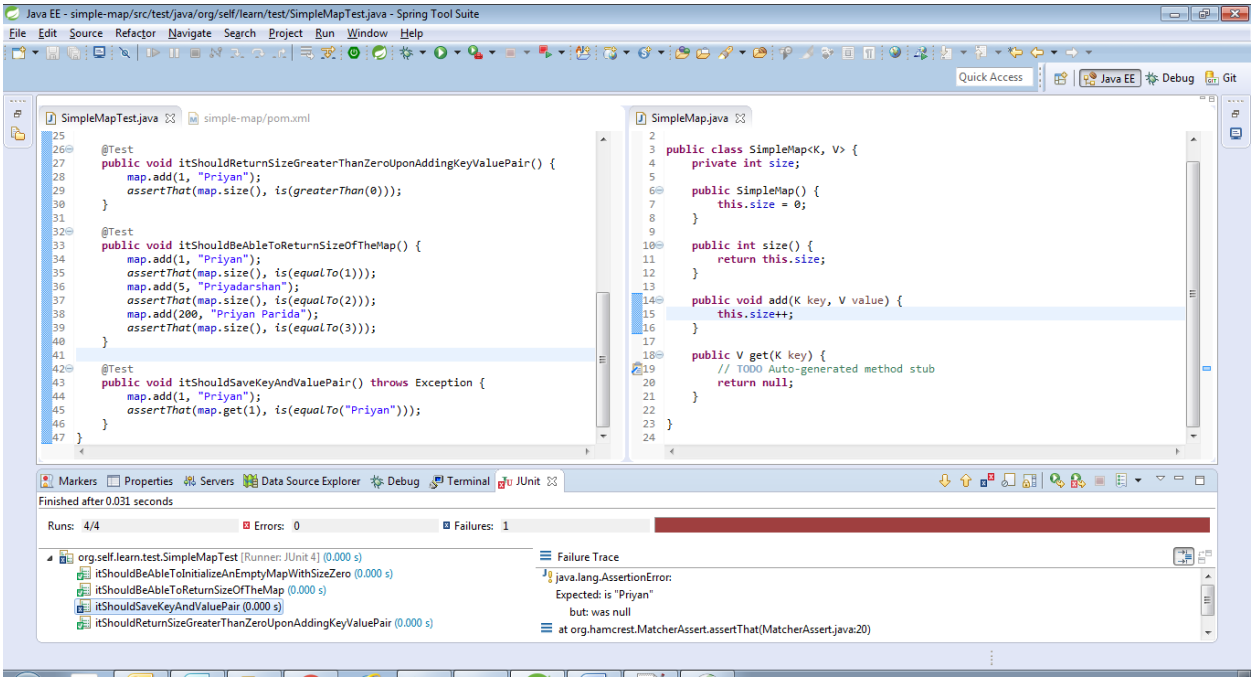
Let's go an extra mile to make sure that our implementation really works! And it does because tests are green.



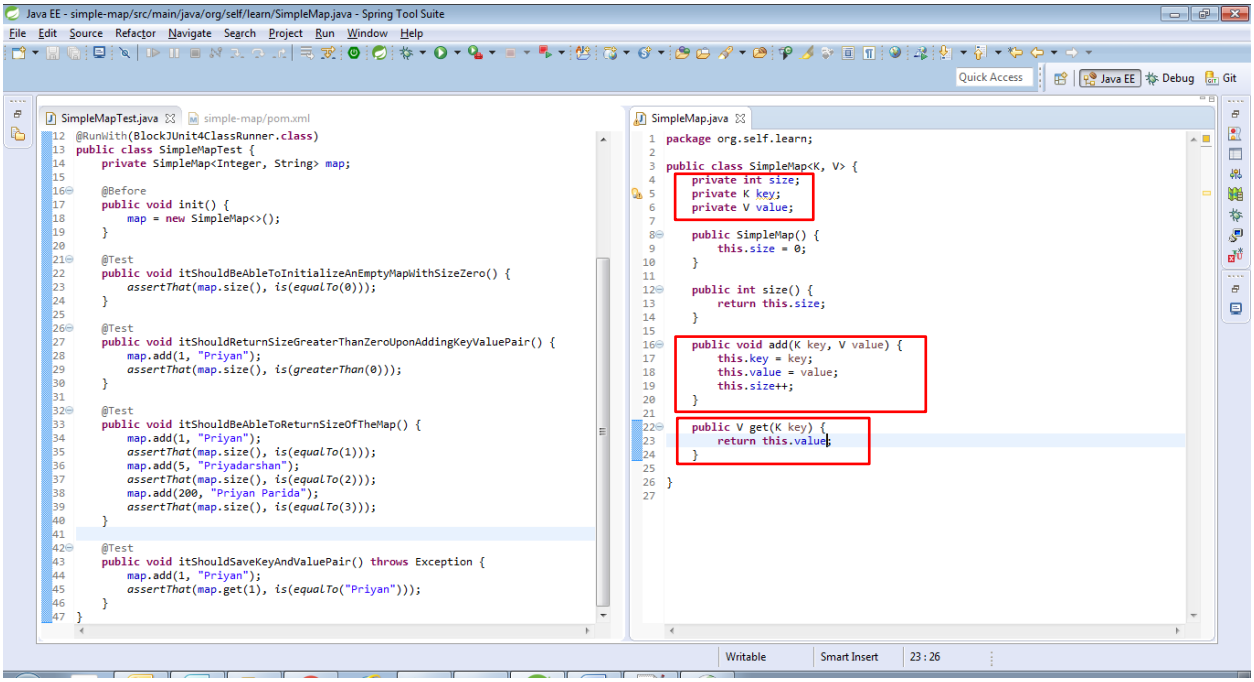
Let's add some meat into this by verifying whether our map is really saving keys/values…

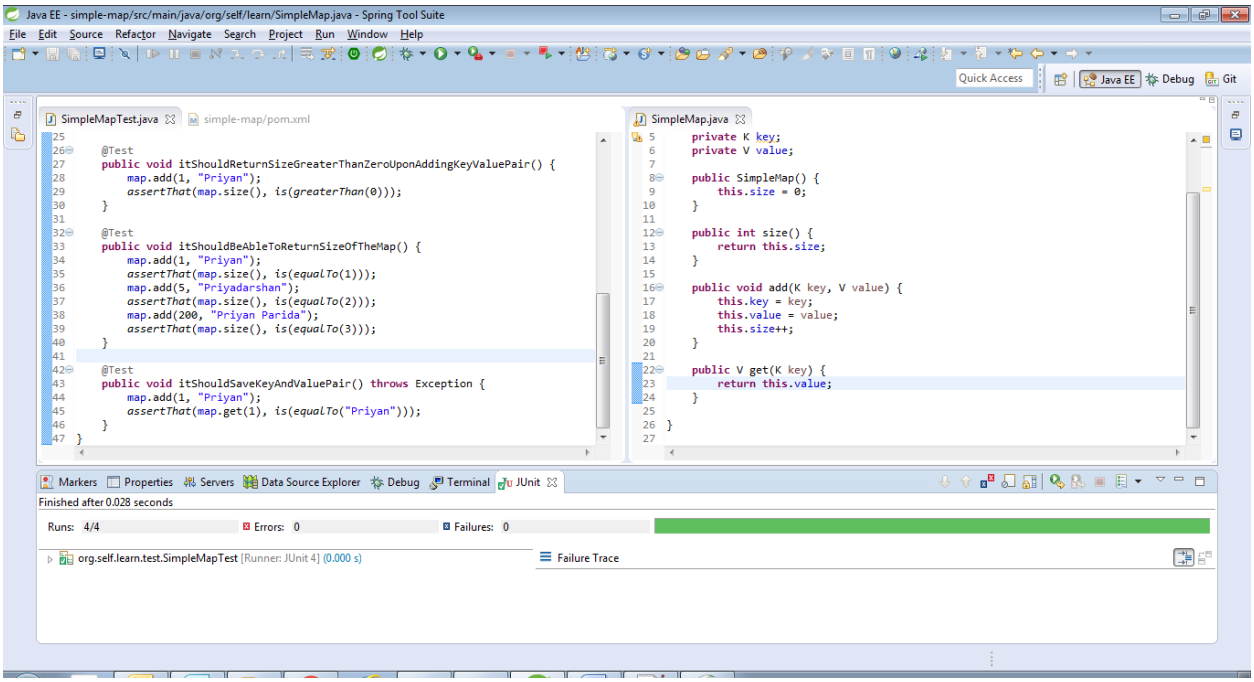


Here is what the completed test along with the auto generated method looks like. The test fails for obvious reasons.

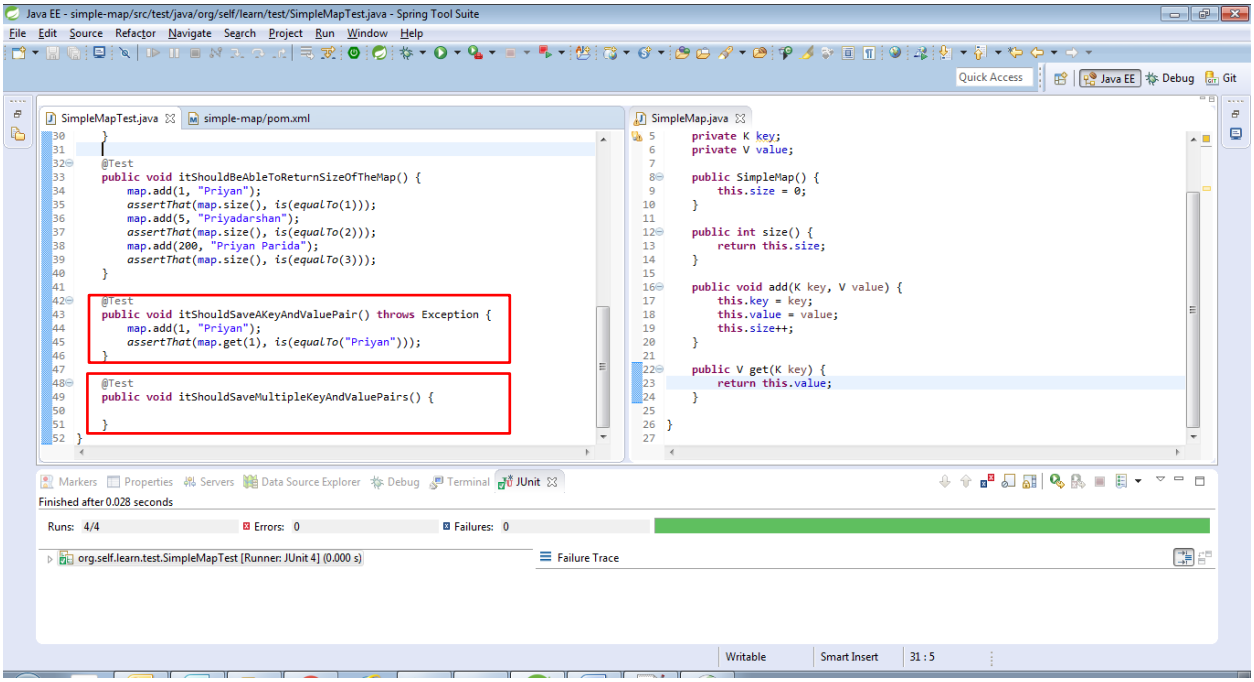


I have gone ahead and provided implementation to address this part. The tests show that the implementation is correct indeed!



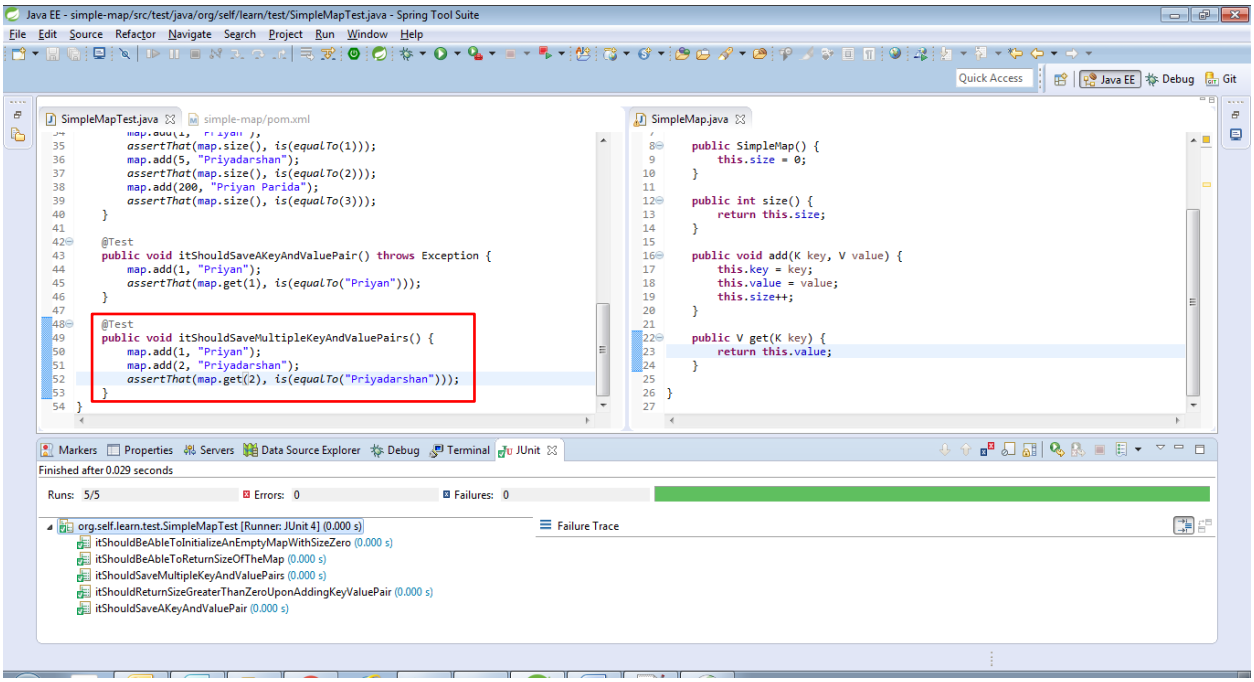


Let’s make the test for values being saved a bit more elaborate. I have renamed my test to better tell what it is actually testing…

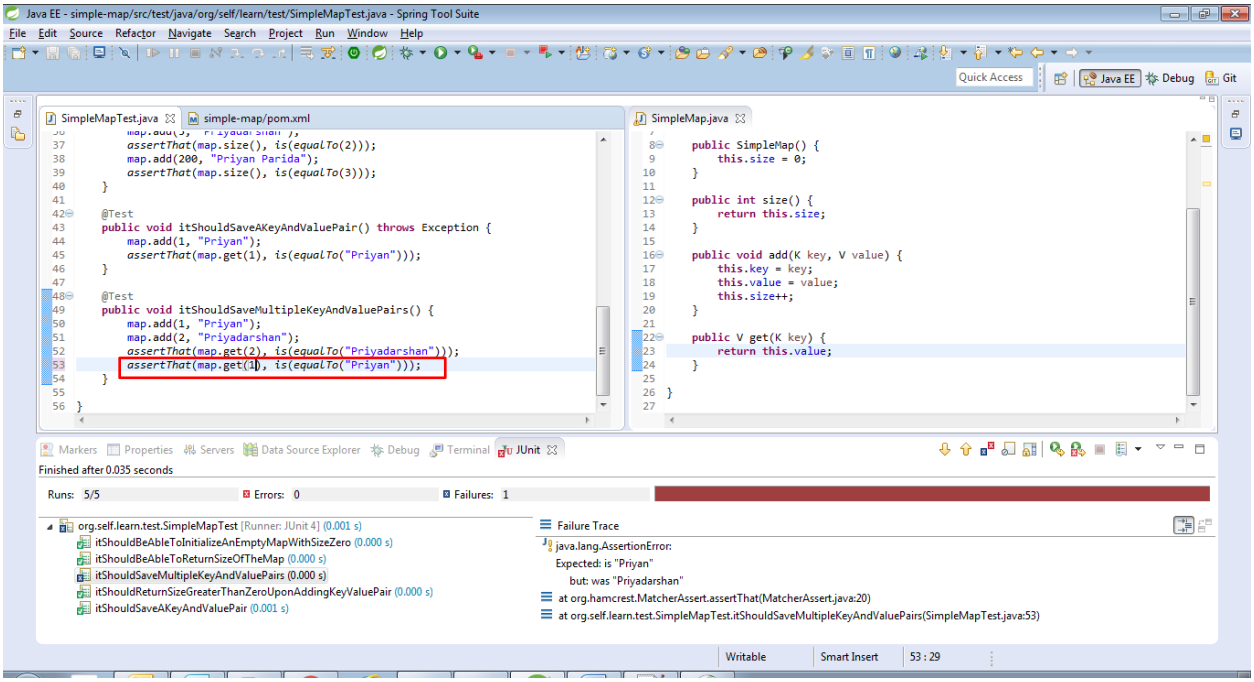


We want to test the map is able to save, not just a single key/value, but multiple keys/values. The test below gives an illusion that it does.

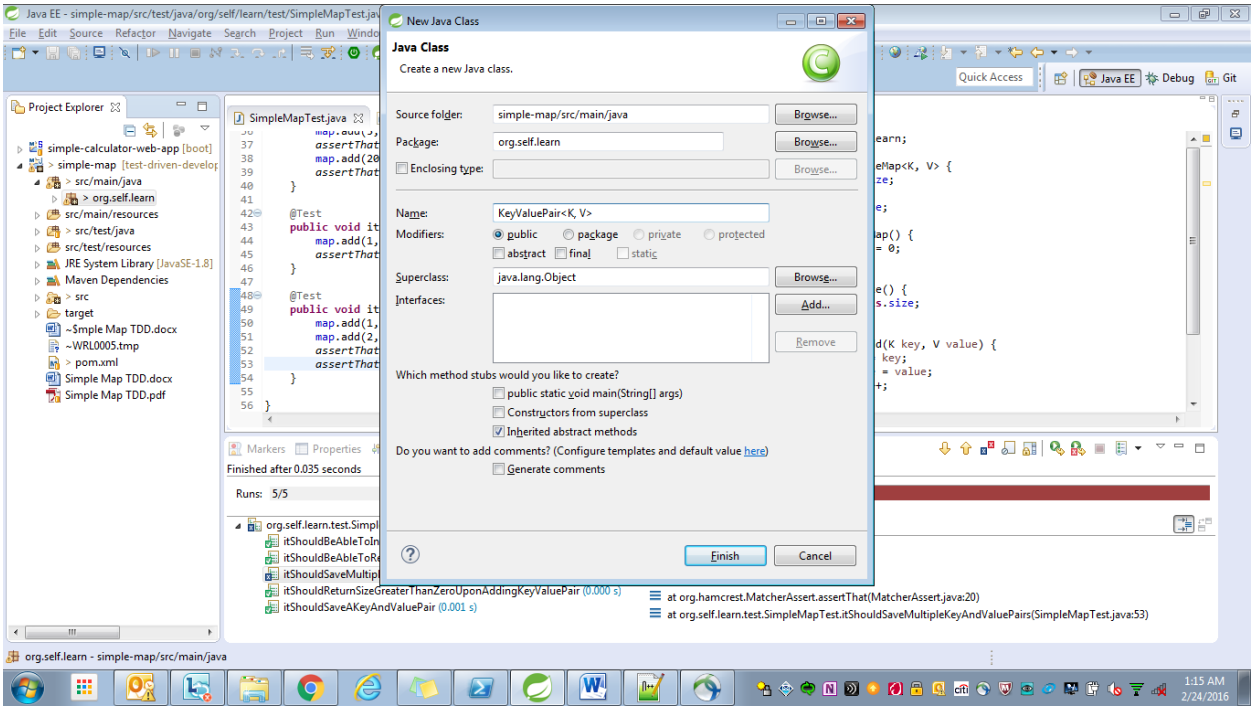
But does it really? Check the next test.



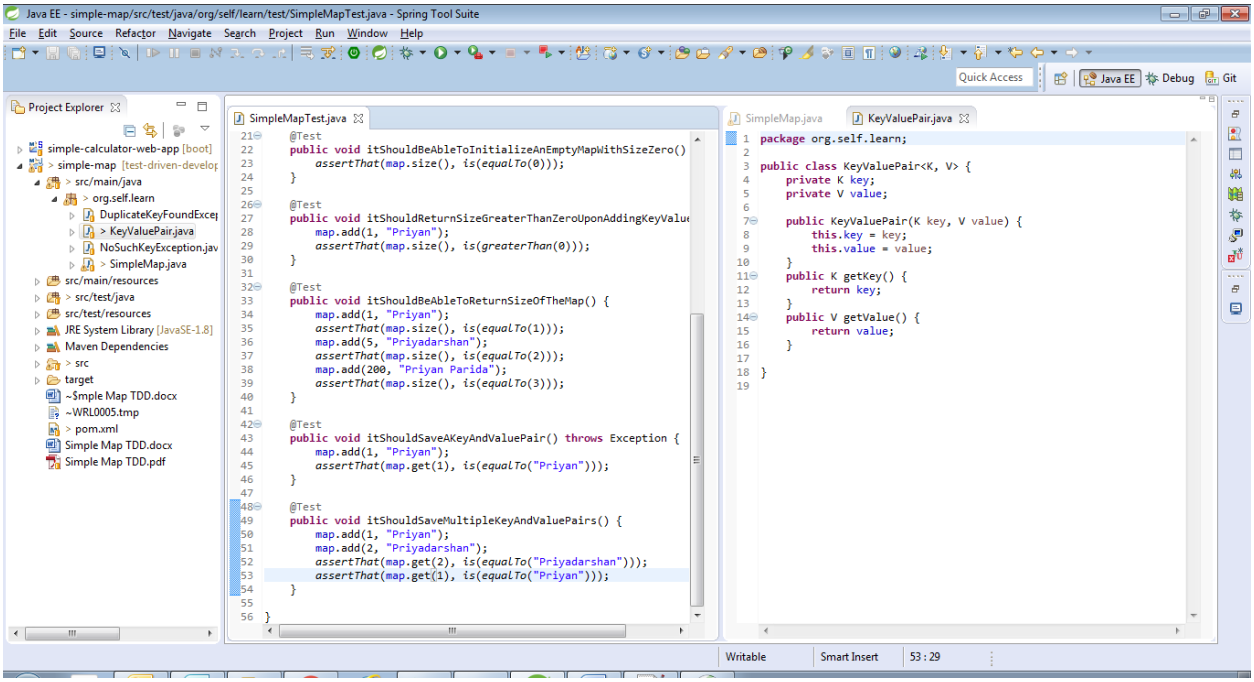
There we go… This test caught us. ☺ Expected “Priyan” but was “Priyadarshan”. Something is definitely wrong.



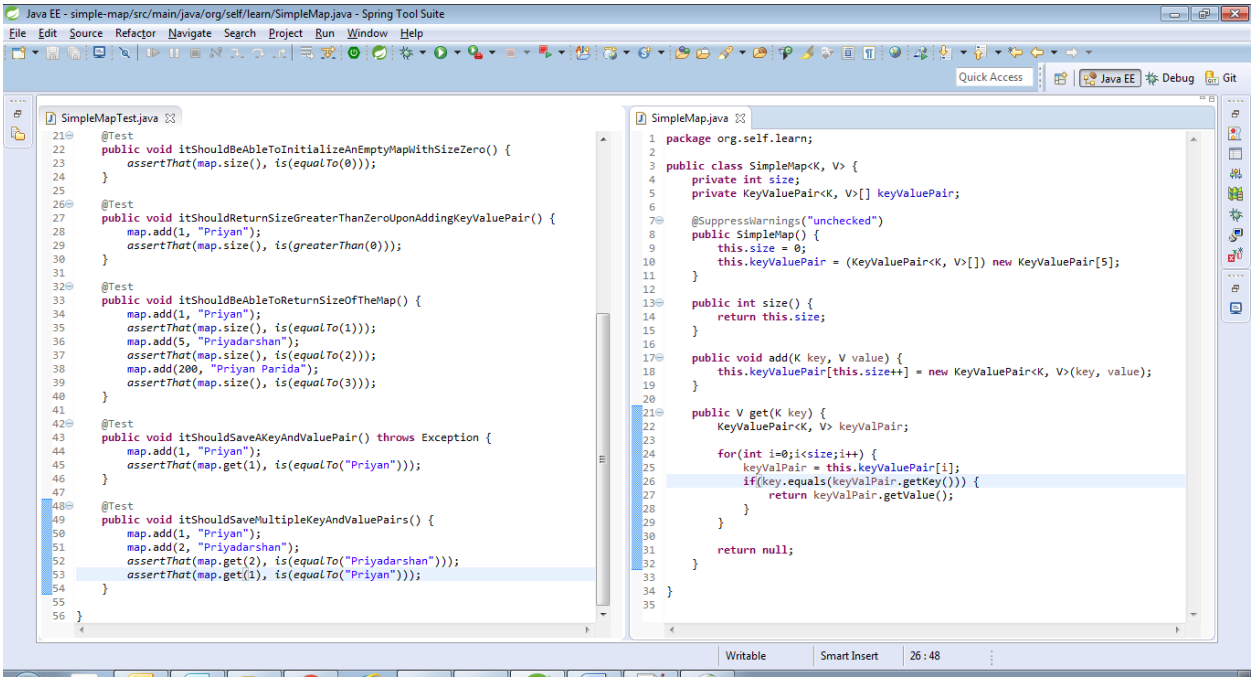
Let's go a few miles to actually store keys/values into the map.



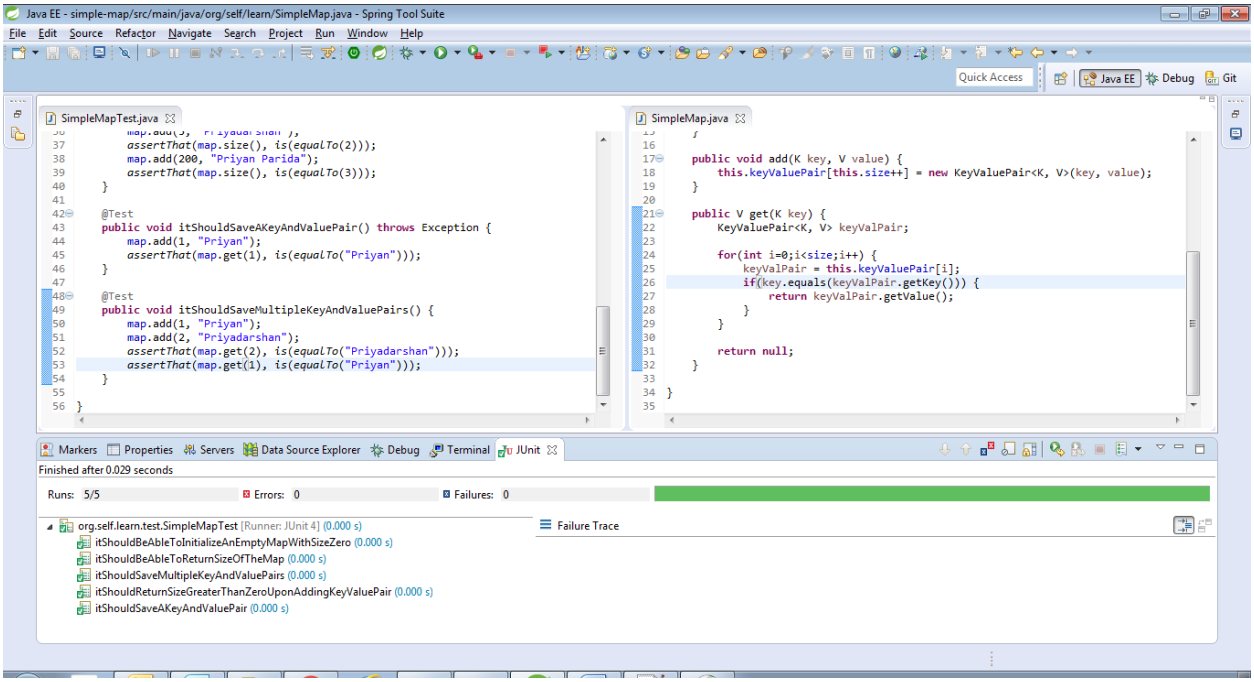
Here is what my Key Value Template looks like.



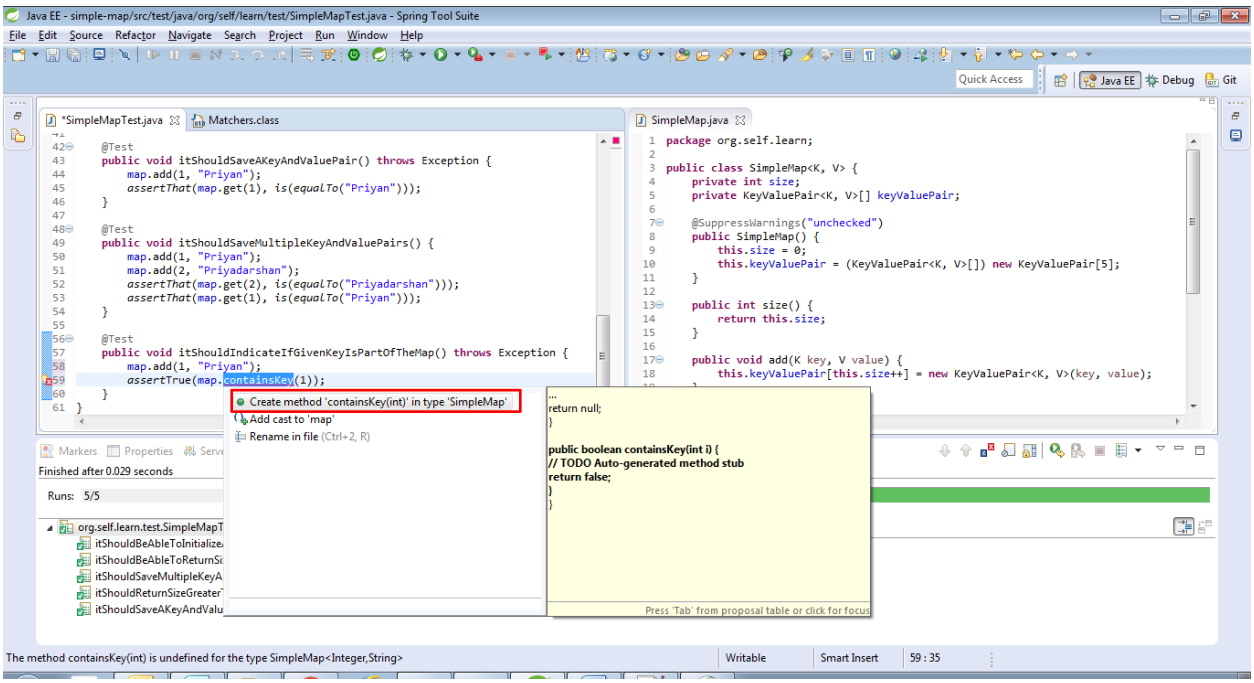
Added array of key value template as a member variable, saved keys/values into the array. The “get()” method now returns the value corresponding to the key.



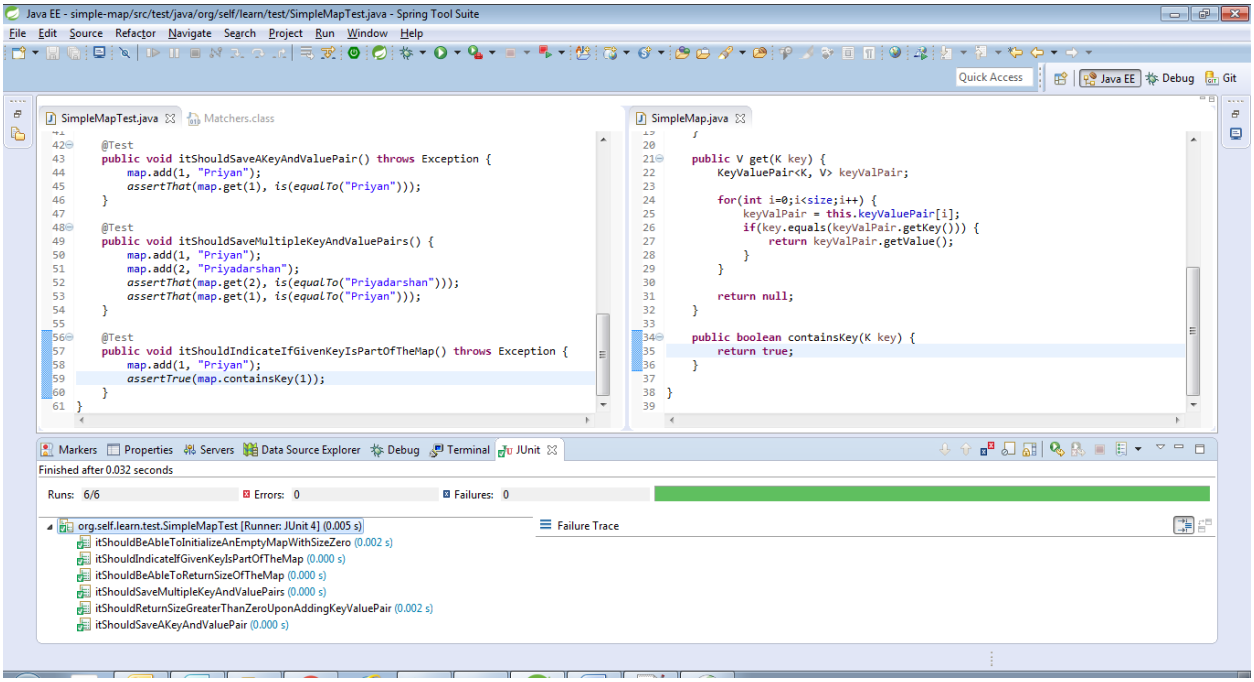
I run the tests and they complete successfully this time.



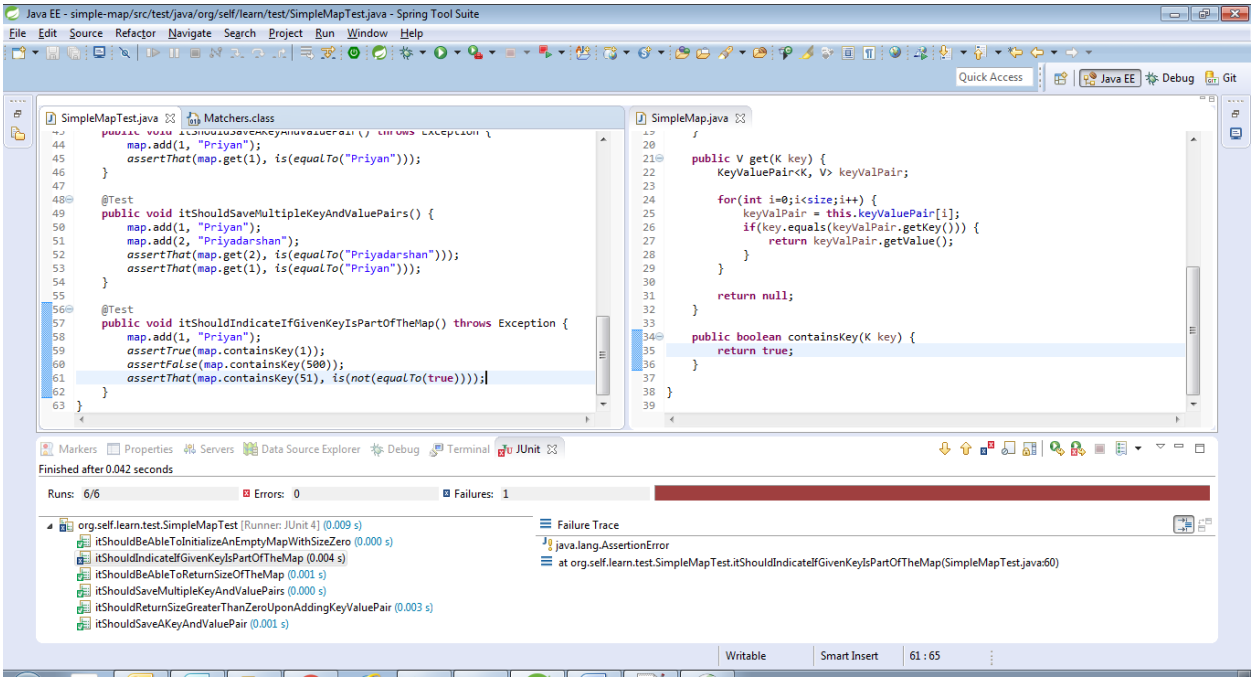
Next, let's write test cases for "containsKey"



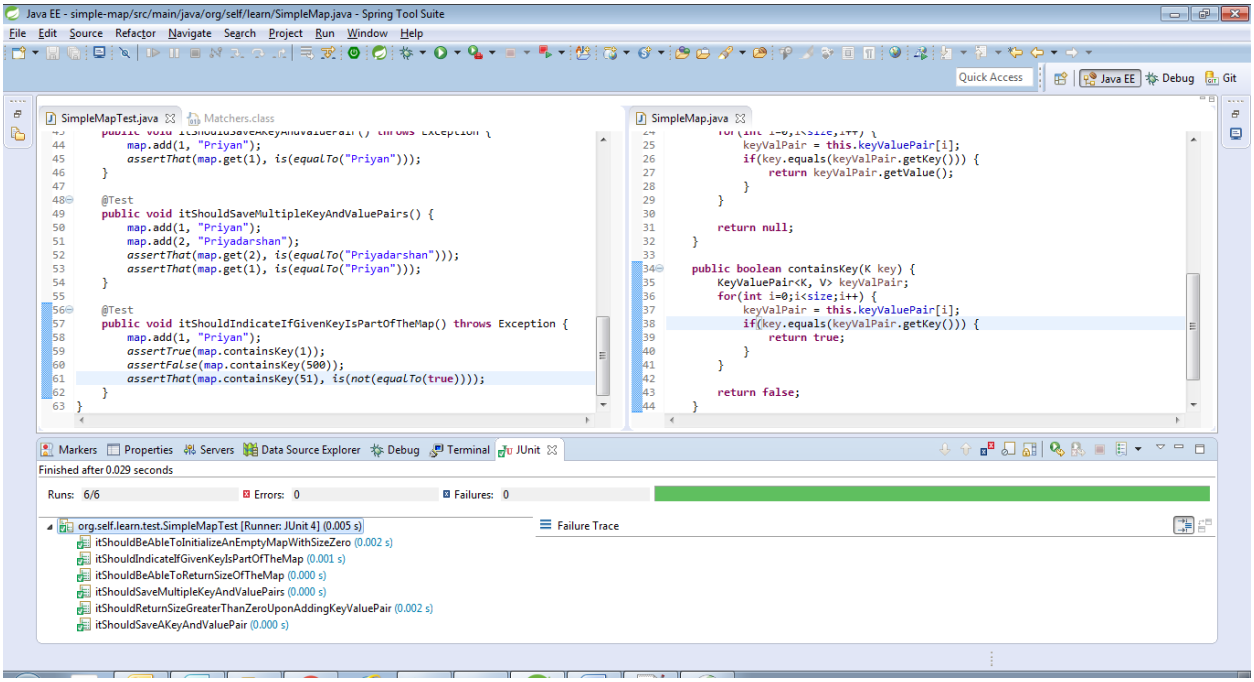
Successful completion of the test gives us an illusion that the implementation is complete. But, it is not. Let's add another test case to prove this.



The test fails when ran now. We know returning “true” always isn’t the right implementation. So let’s go fix it.

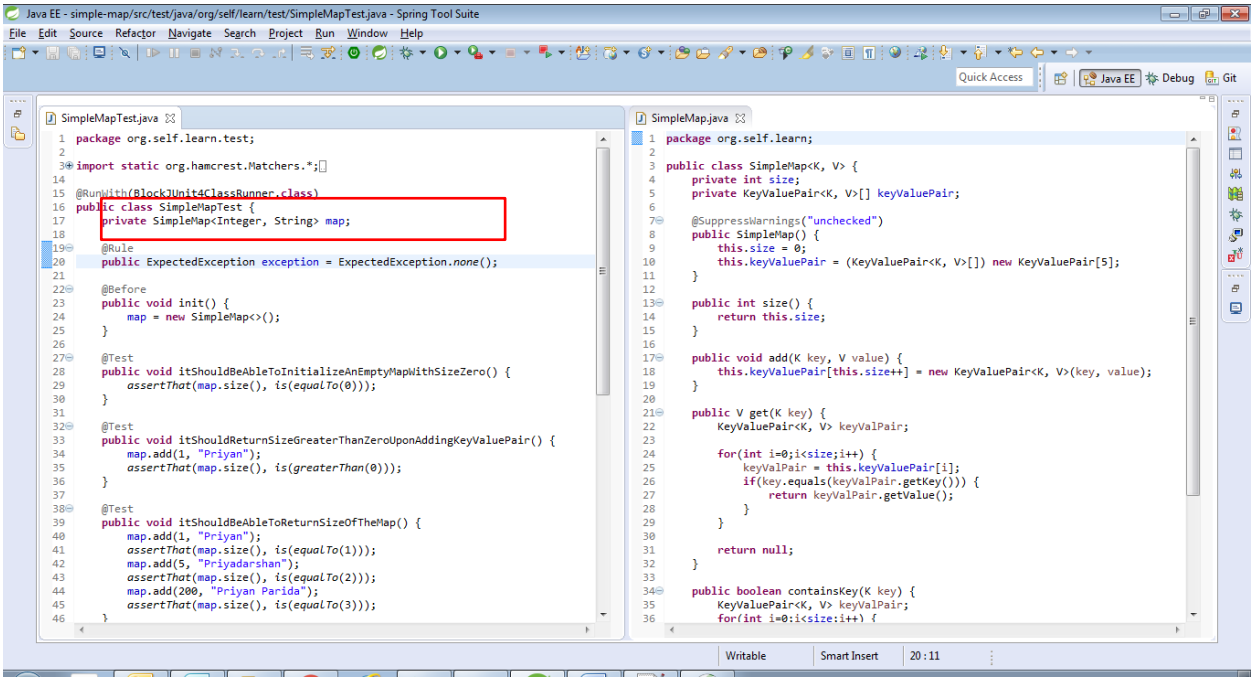


The test result indicates that the implementation is complete now.

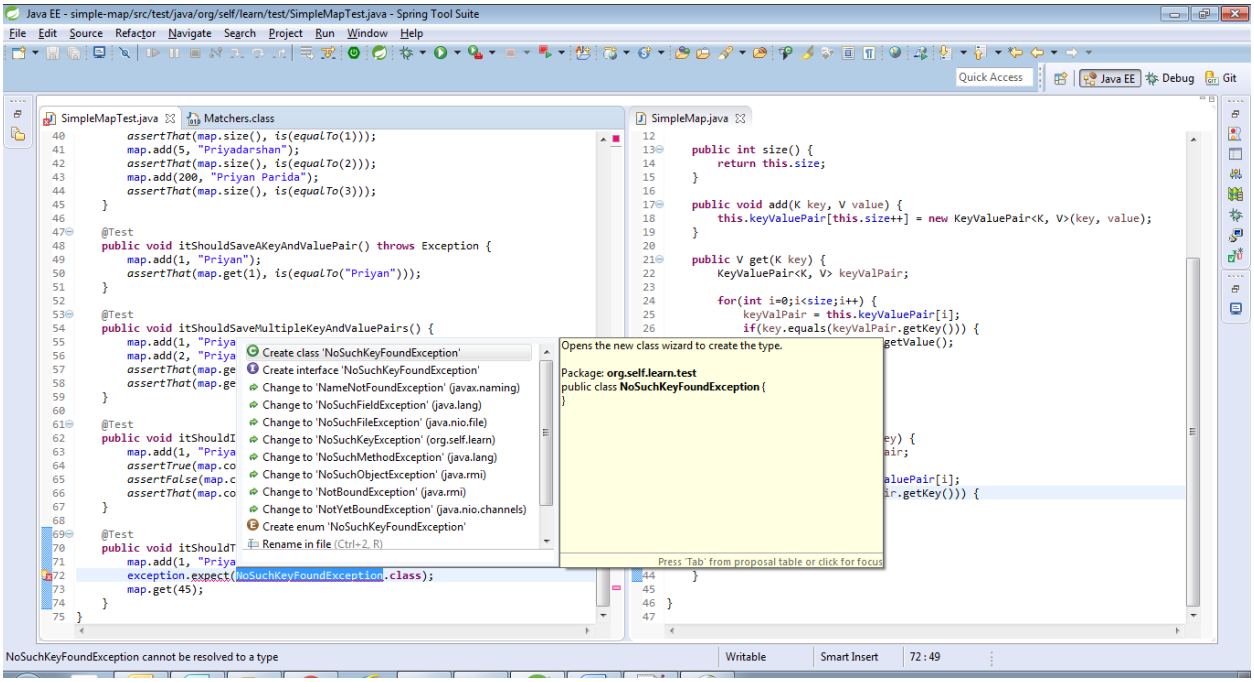


Next, let's test the edge cases w.r.t. getKey(). We want it to throw an exception when a key isn't found. Note how the tests drive the implementation.

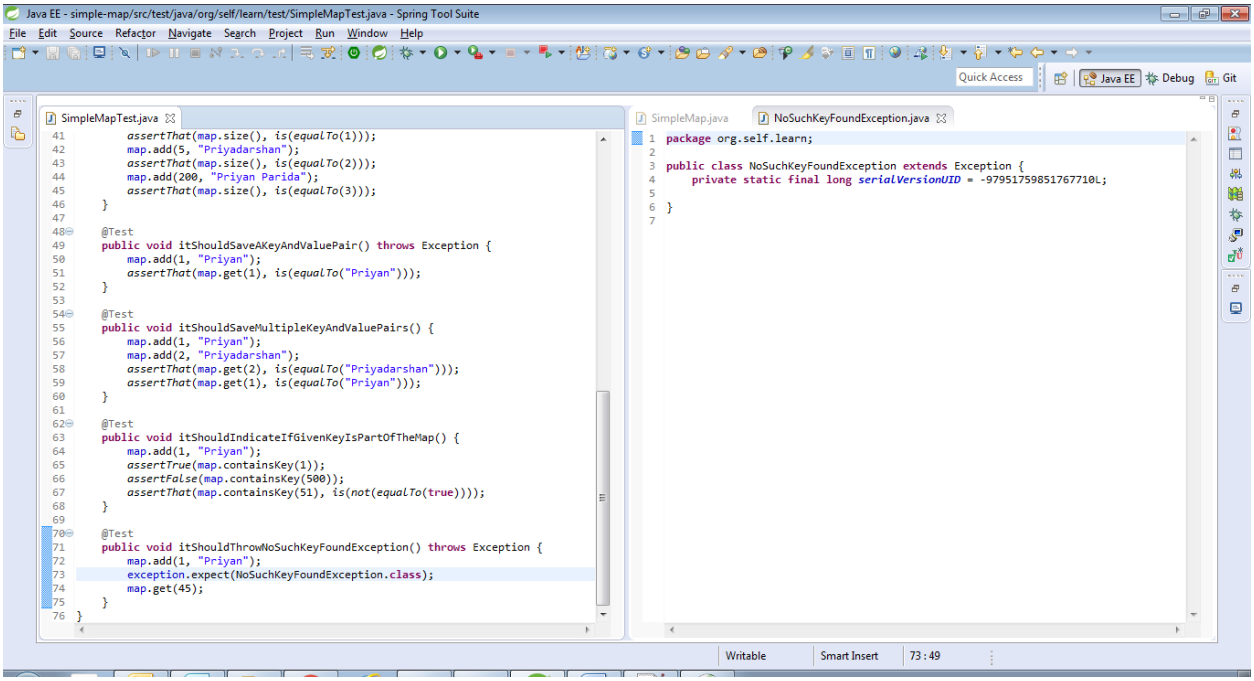
We verify exception using a Junit Rule. It's one of the several ways of testing this.



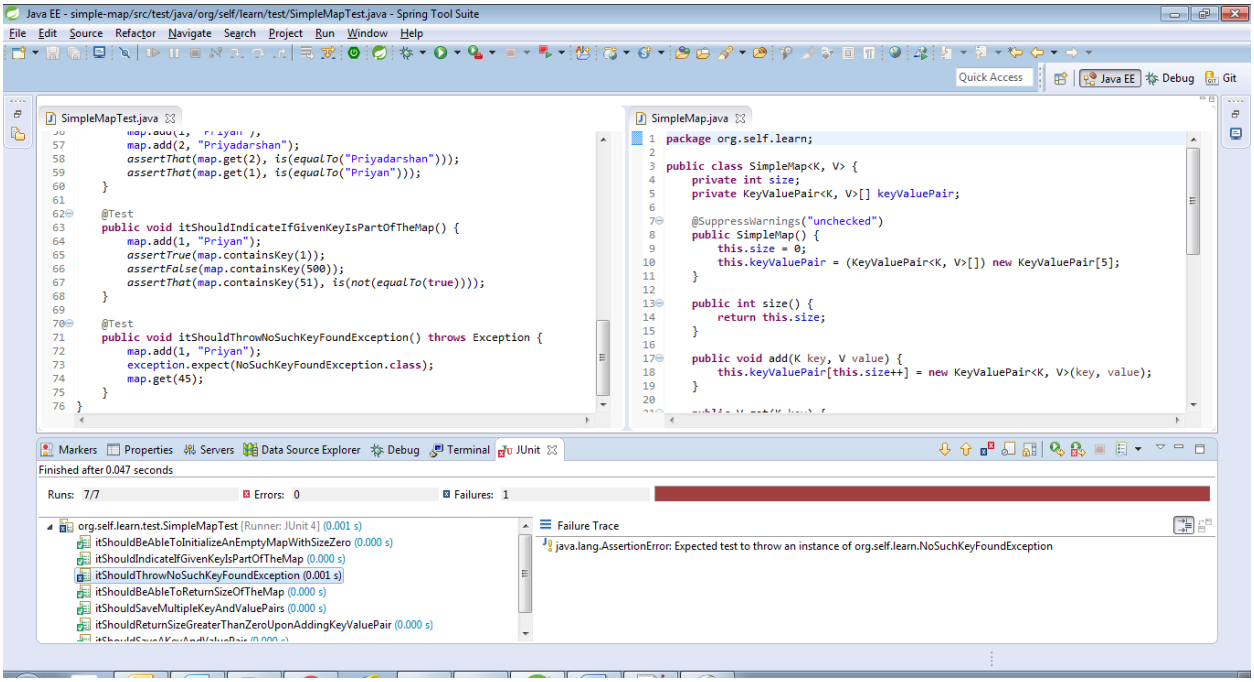
Next we write test and create the custom exception object..



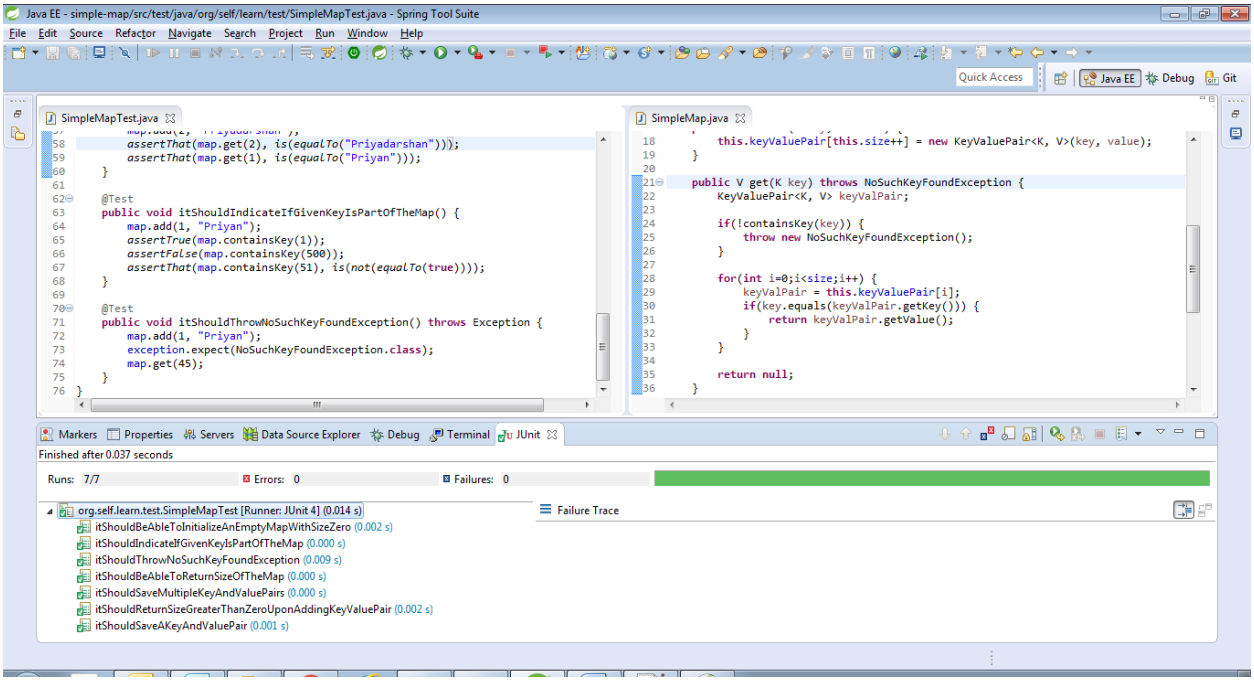
Compilation errors are gone…



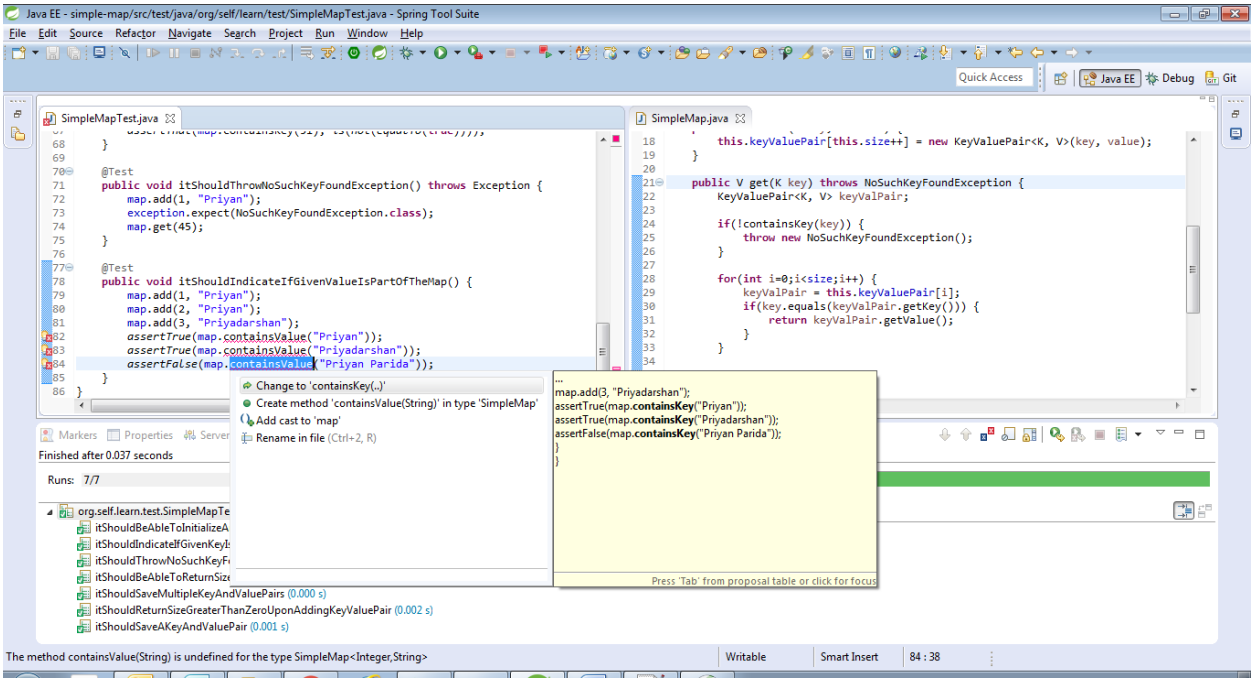
With the compilation issues resolved, we are ready to run the tests now. The test fails as we expected it to.



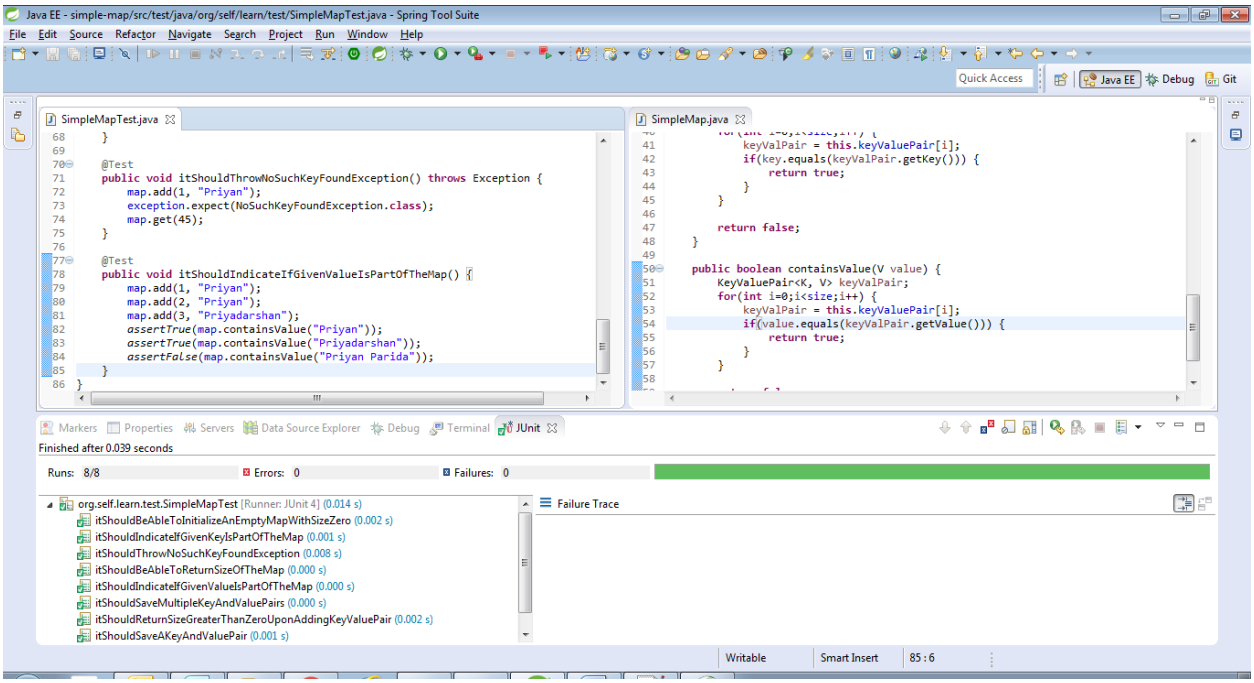
Let's quickly get this working… The test completes successfully.



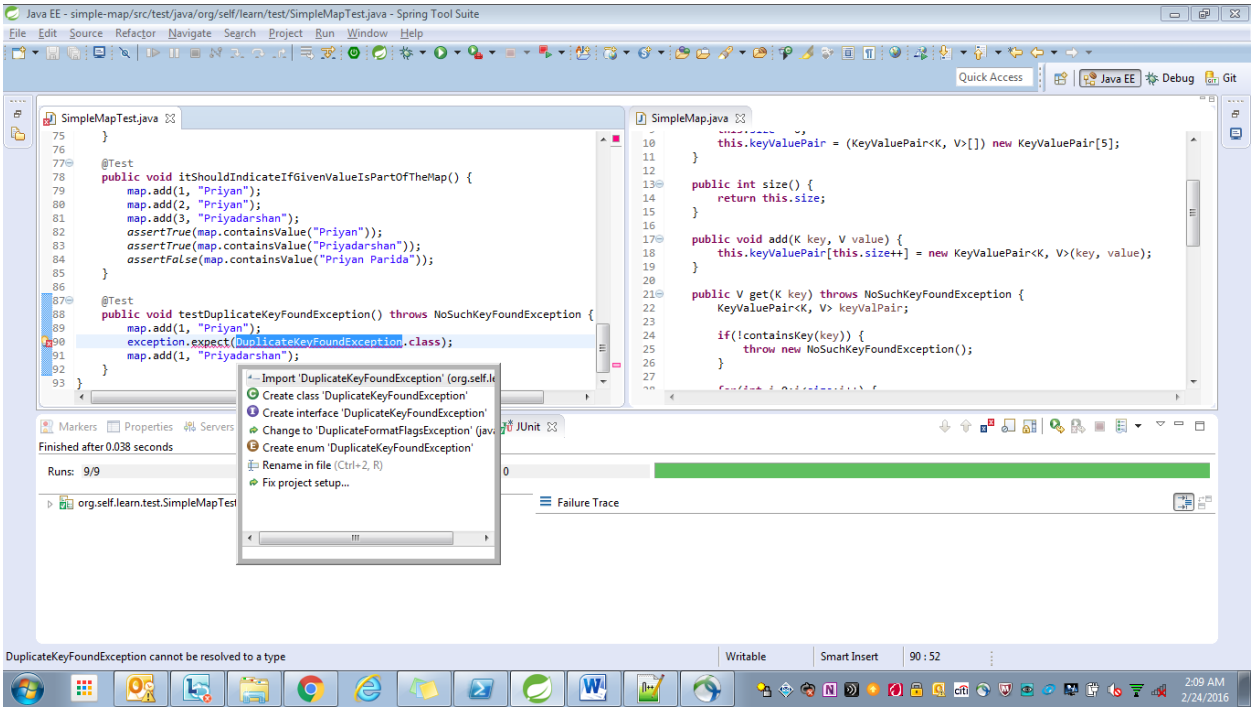
Next up is test/implementation for "containsValue()"



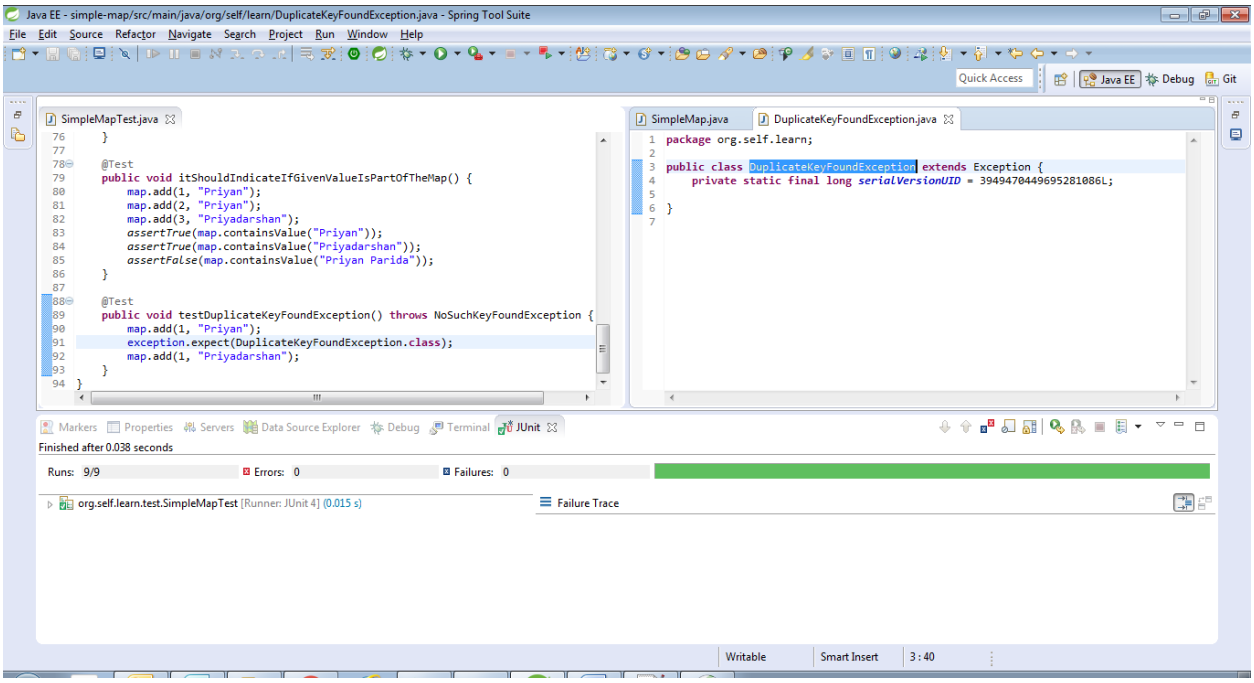
I have skipped a couple of steps here i.e. auto generate method, implement the smallest bit, run test and start over with the next bit of implementation… Here is what the final test and implementation looks like. That took care of it, the test results indicate that.



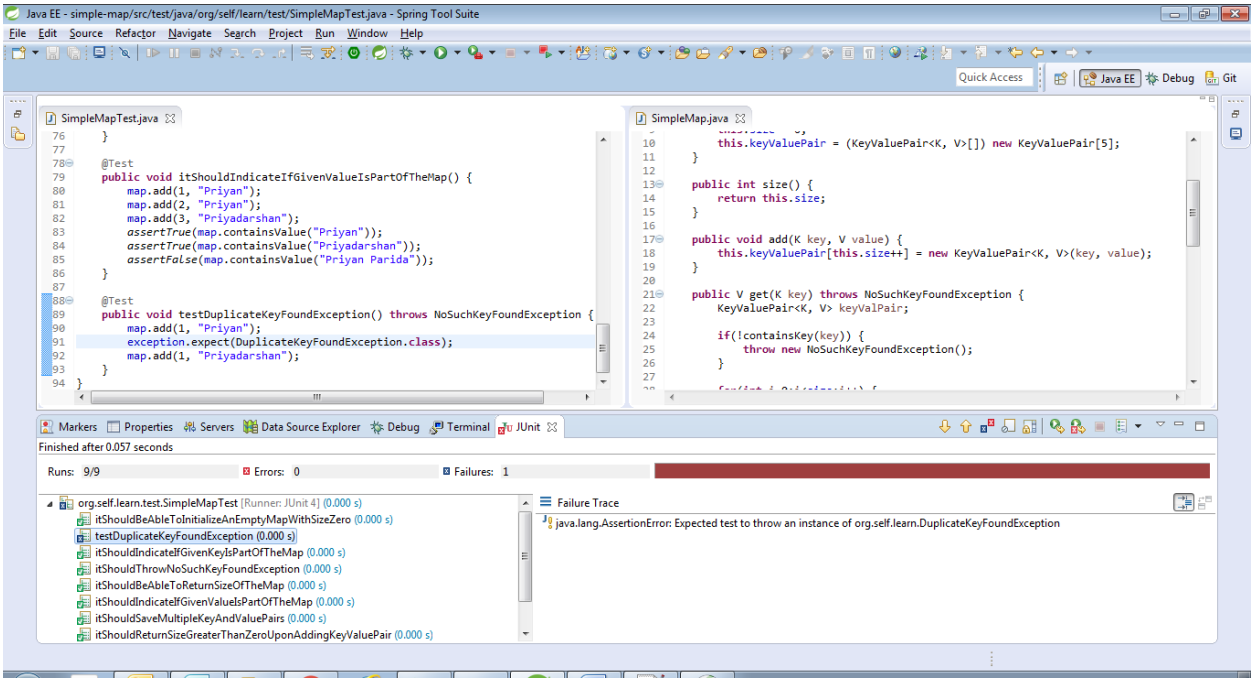
We do not want to allow our map to contain duplicate key and throw an exception when that is the case. Let's write a test for that.



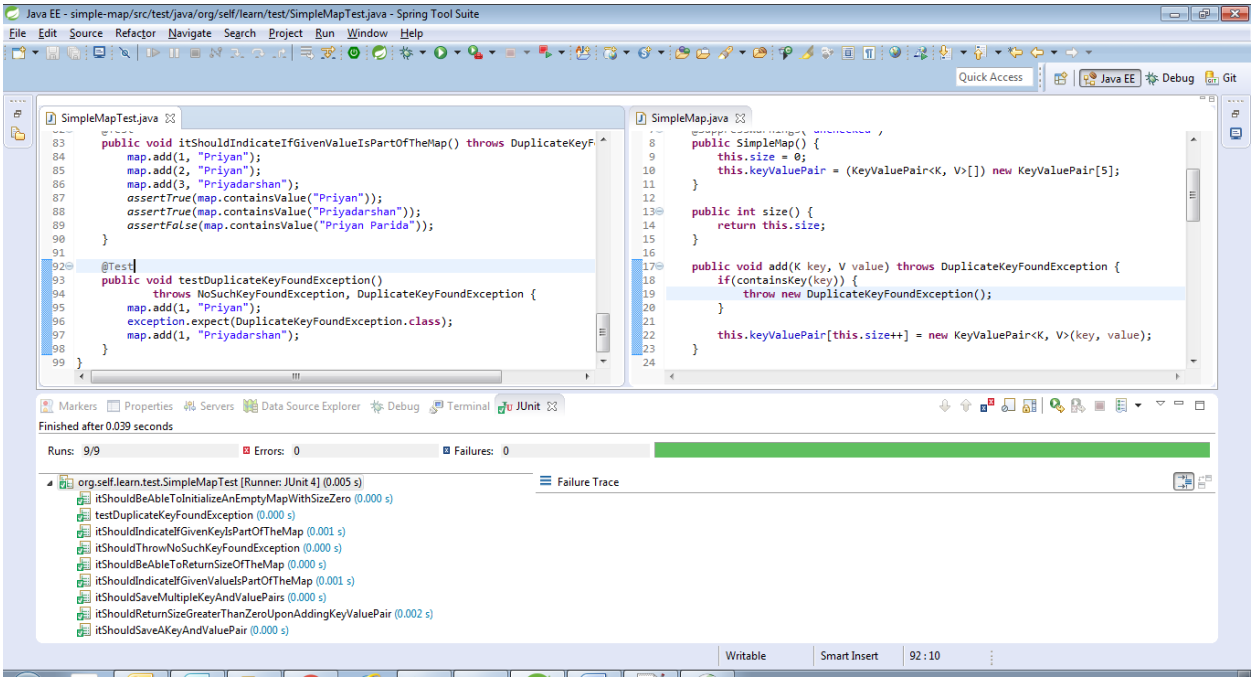
Create the custom exception class to resolve compilation error…



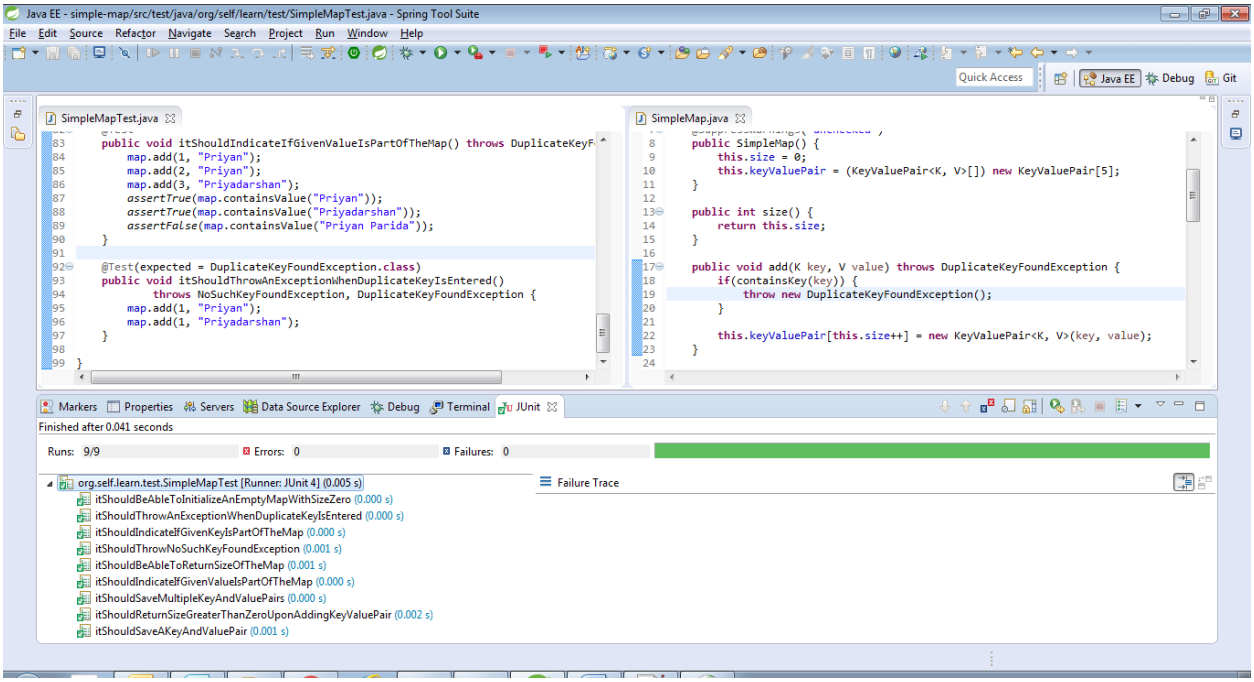
The test fails because we are not surfacing an exception yet.



That will fix it, the tests are all green now.



Btw, here is another way of asserting exceptions

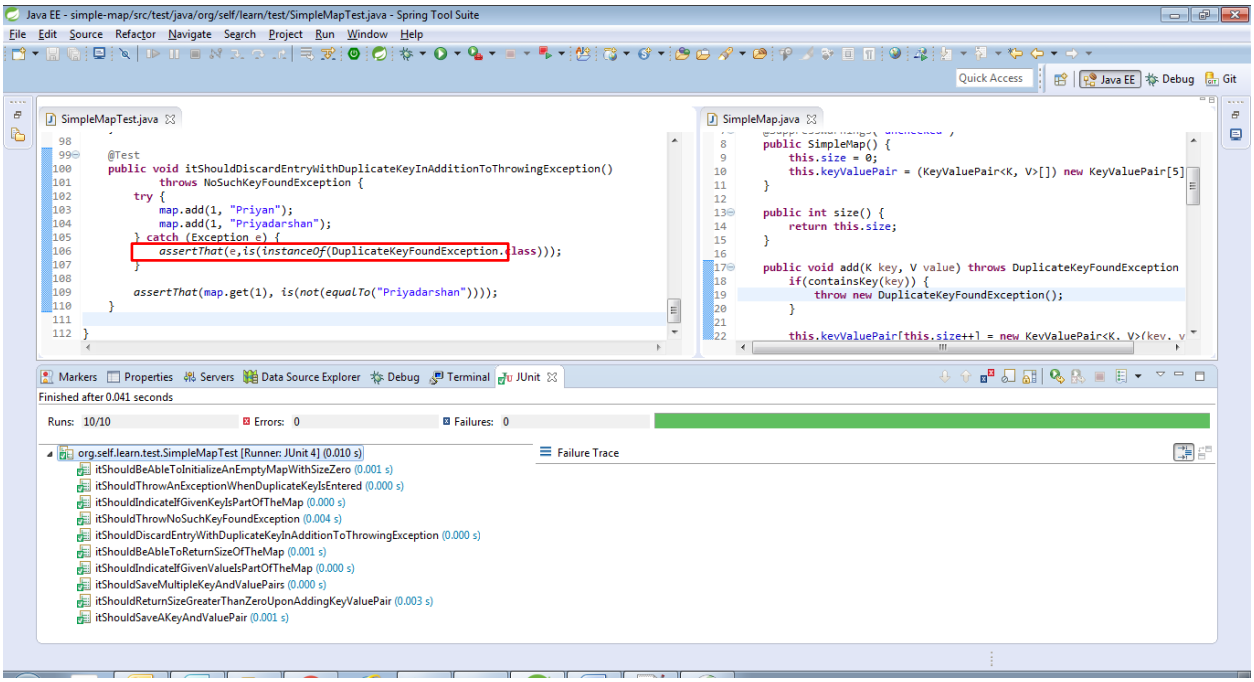


Along with asserting exception, I also want to ensure that the value inserted with duplicate key is discarded.

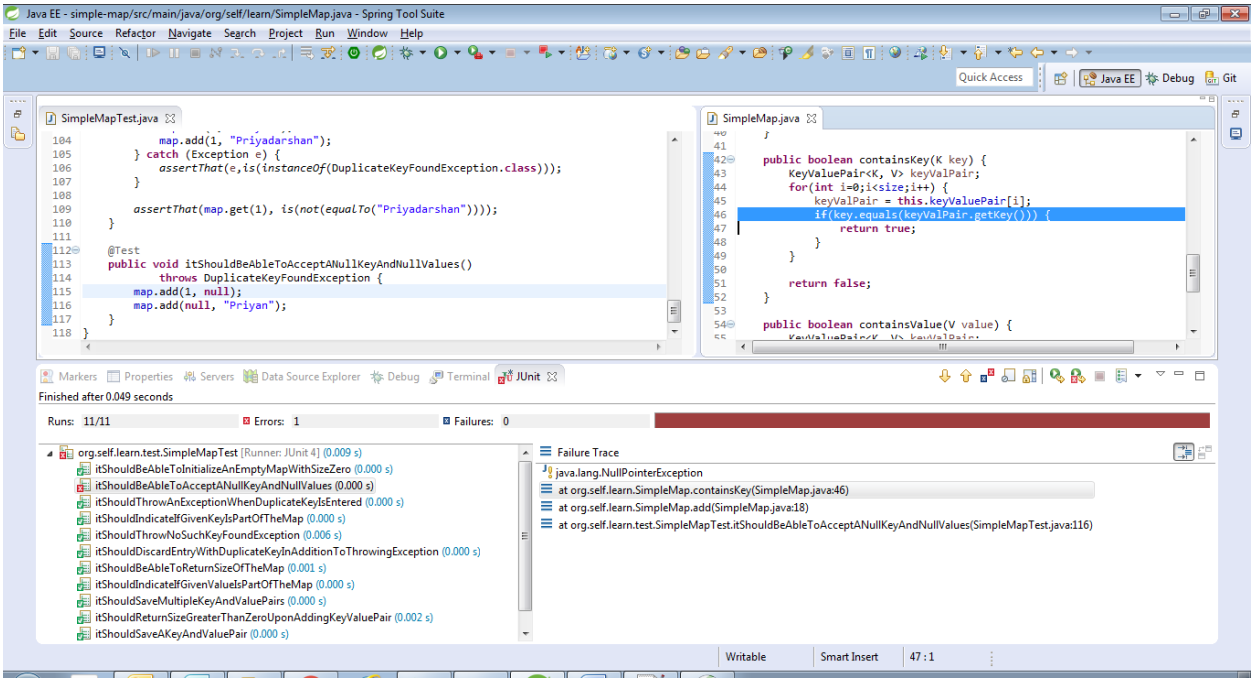
Make note of the assertion style we used in this case. Junit rule could not have been used in this case because anything after exception is thrown, won't be executed i.e. the following would not work:

```
map.save(1, "Priyan");
thrown.expect(DuplicateKeyFoundException.class);
map.save(1, "Priyadarshan");
assertEquals("Priyan", map.get(1)); →This assertion will never be executed.
Same is the case with @Test(expected = DuplicateKeyFoundException.class)
```

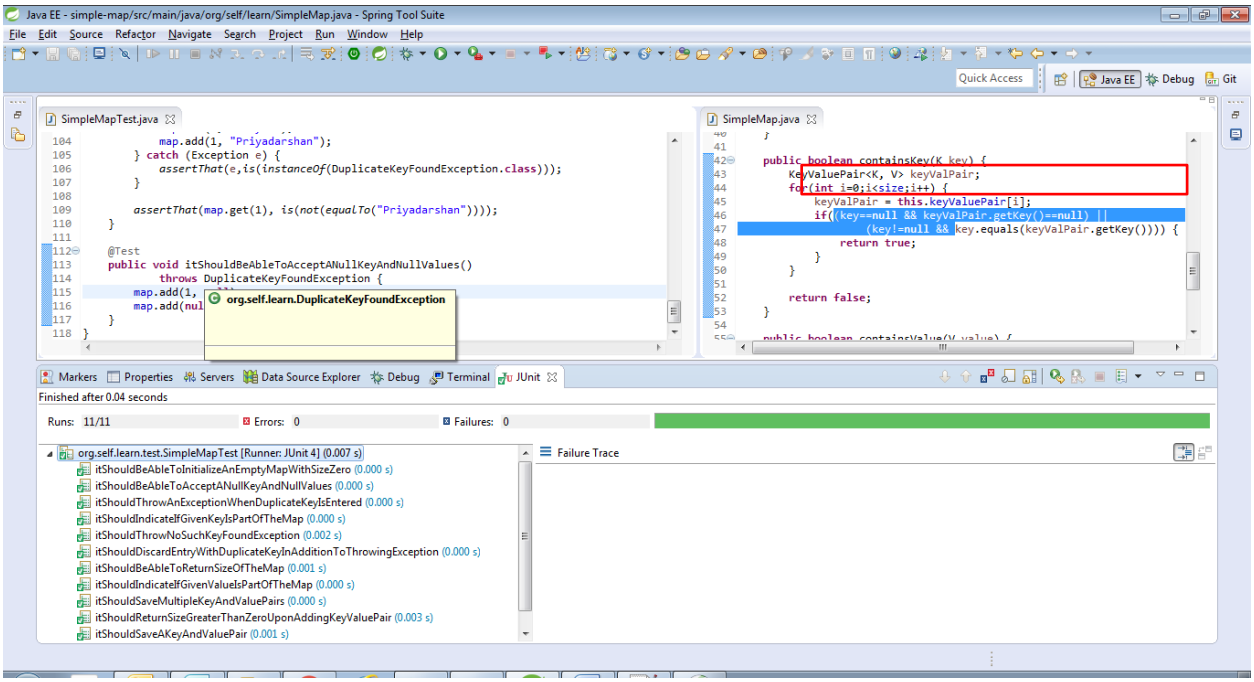
I have added another way of dealing with exception based assertions to write that test. The test did complete successfully.



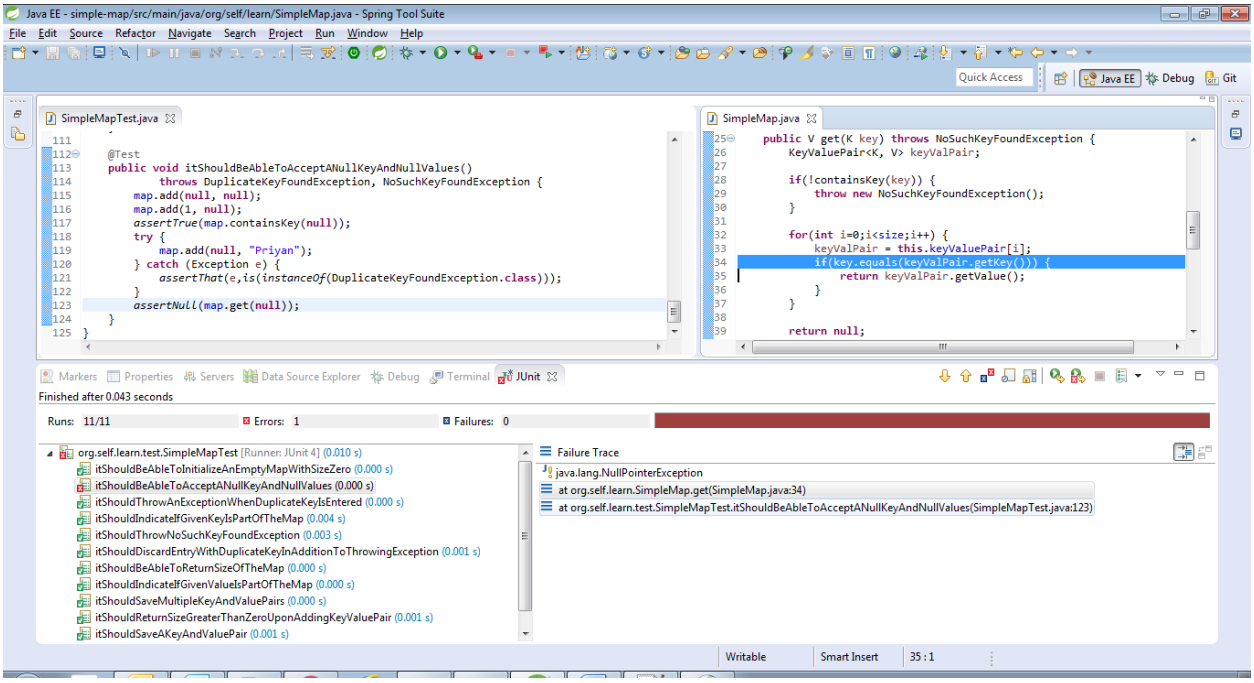
Let's test if the map is able to handle null key (and not keys) and values. I ran a premature test to find that the map does not handle that yet! Make note of the NullPointerException which leads to test failure.



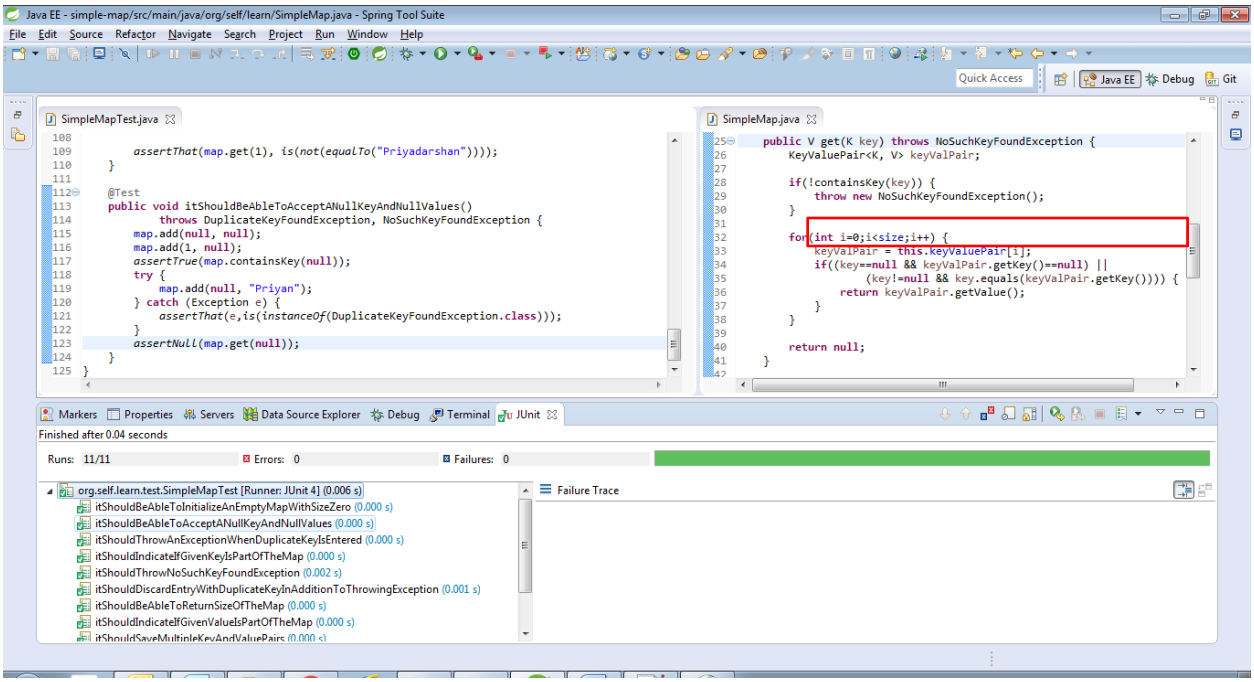
Refer to the fix highlighted in RED. Test results confirms that the fix took care of the issue.



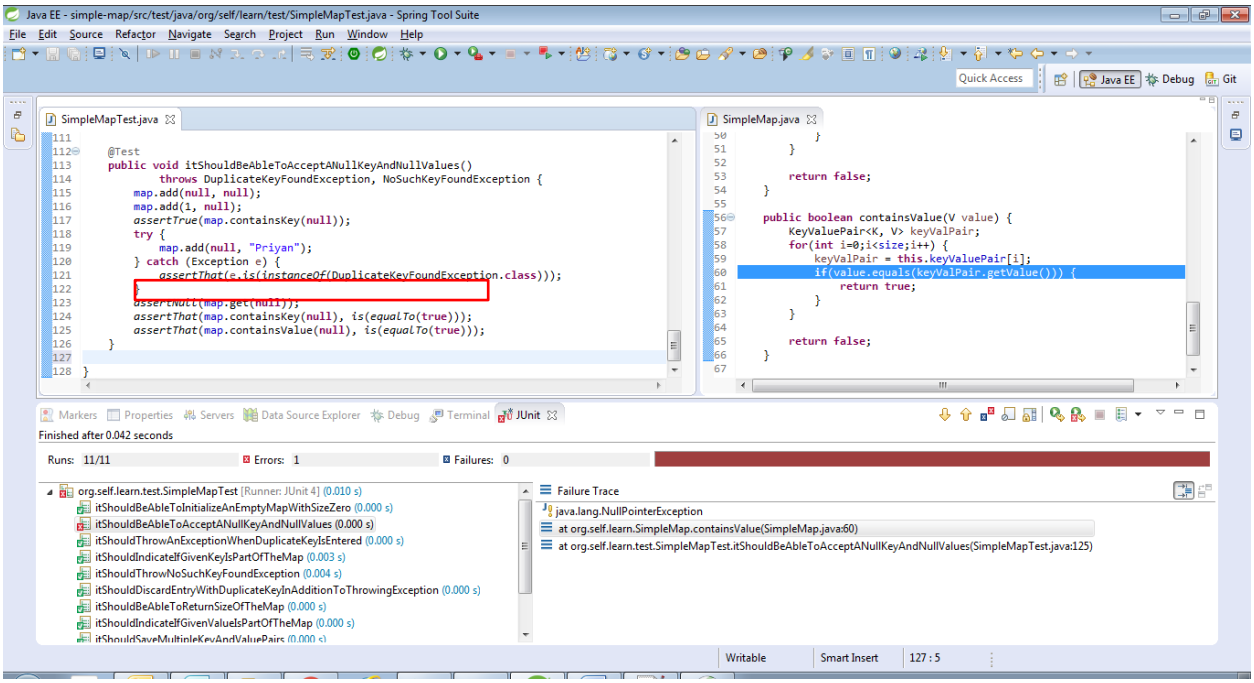
Here is an attempt to make the test more comprehensive and test result indicates that we haven't cared for all scenarios yet.



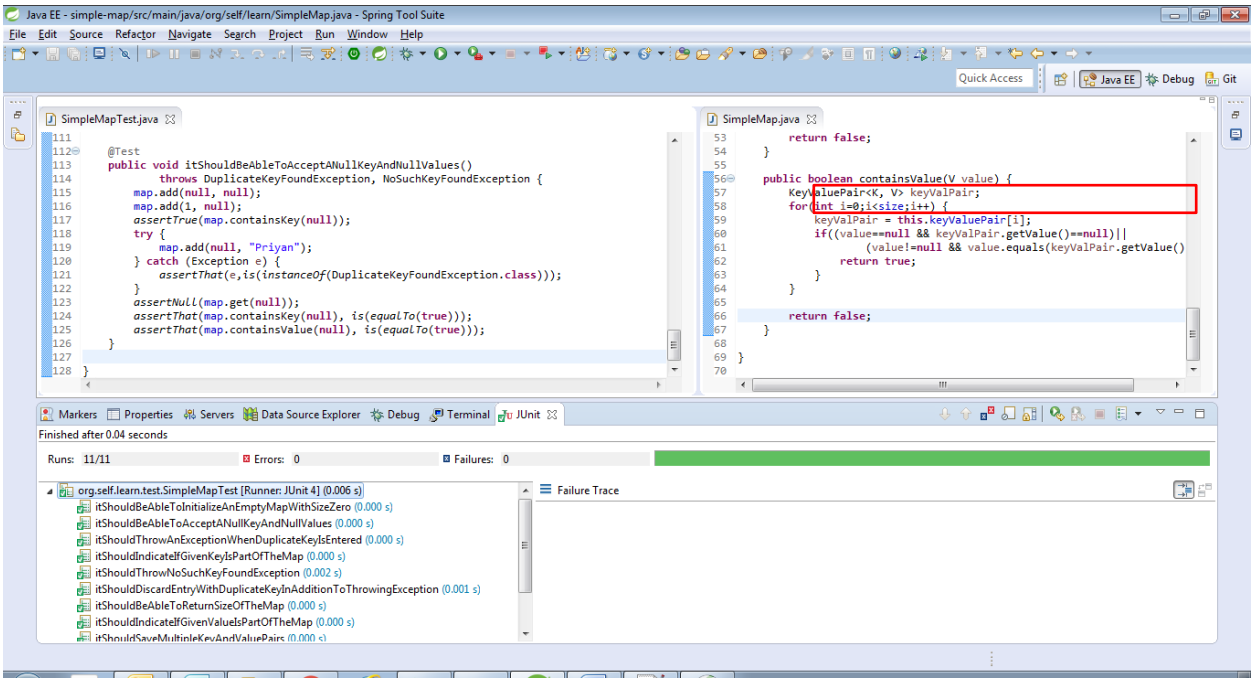
Refer to the highlighted code fix, tests look good with that.



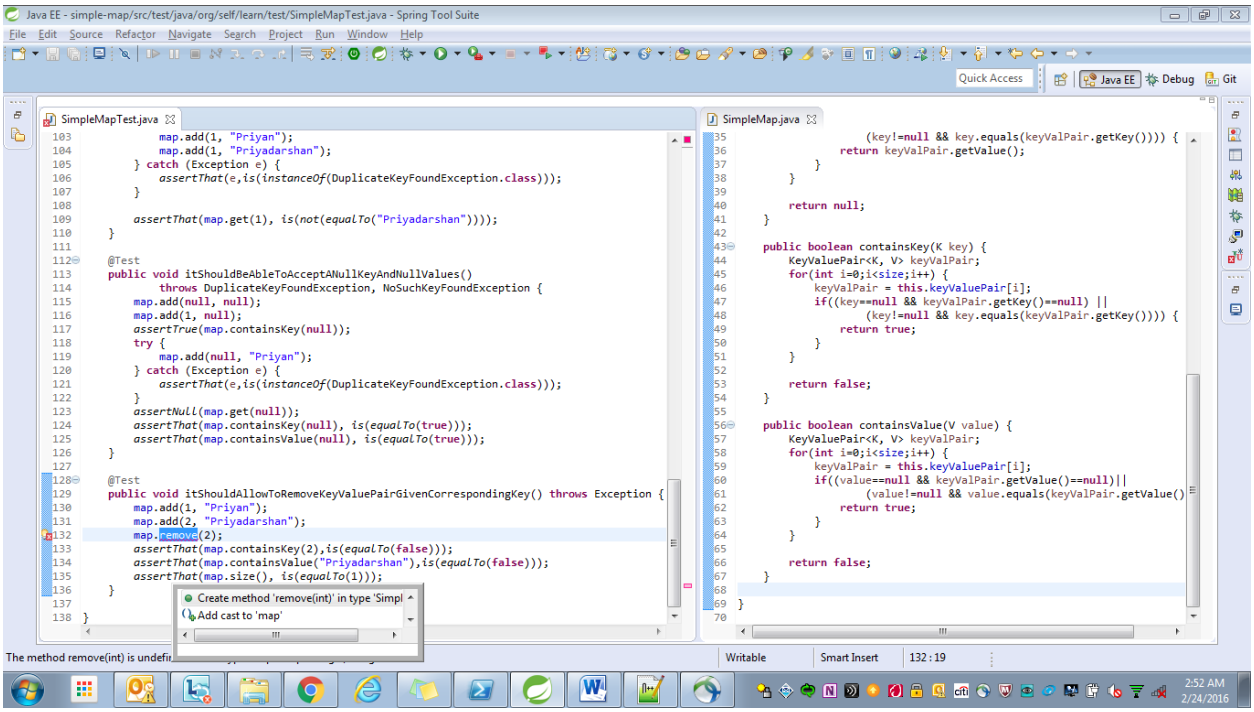
Following test-build iteration is to ensure that the same handled for containsValue() method as well.

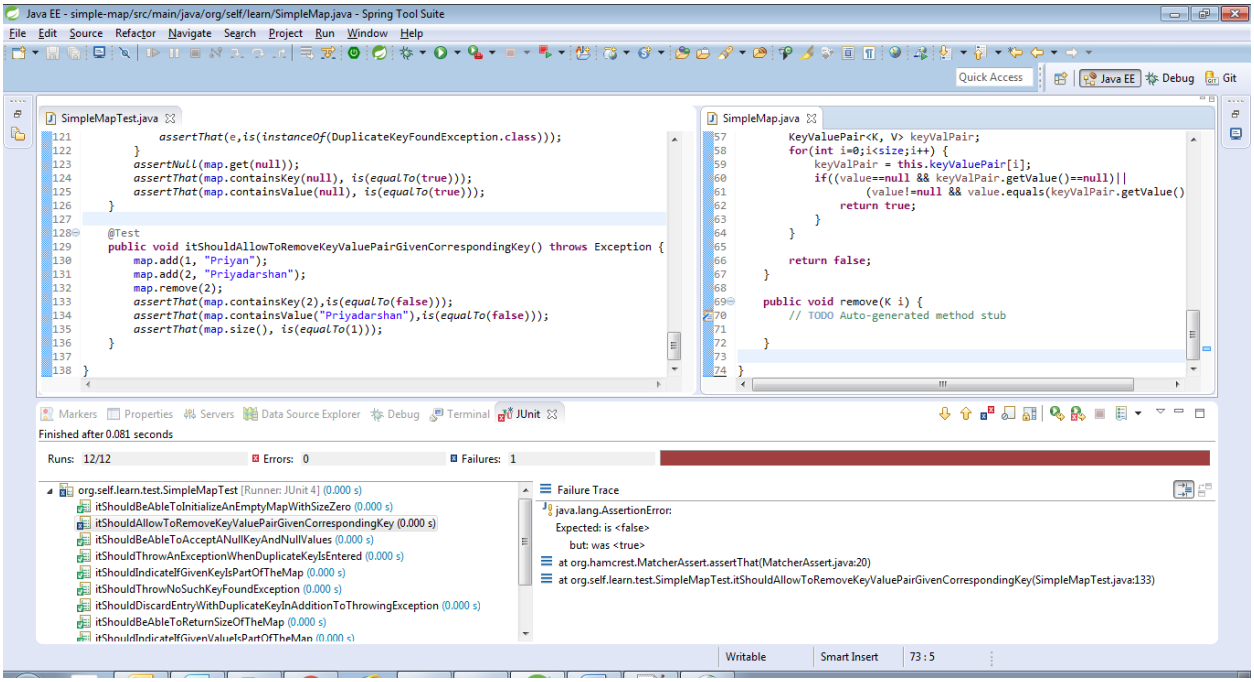


That will take care of it!

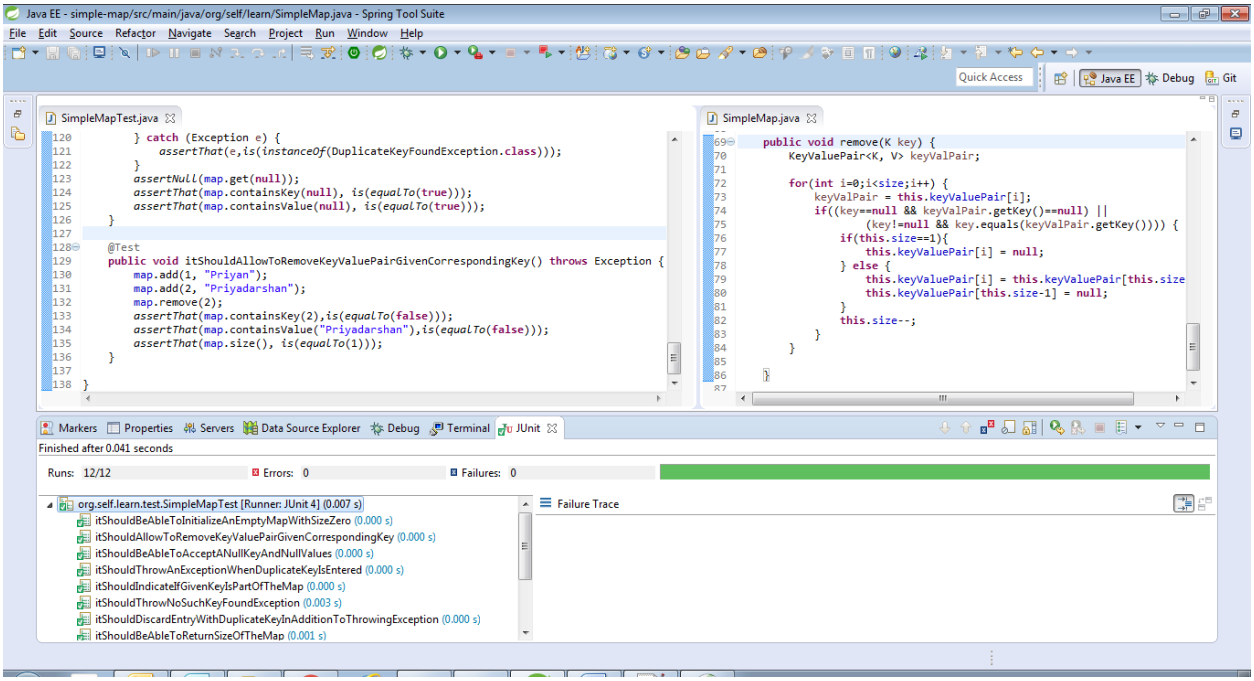


Next up is remove… Following few screenshots depict how we put its implementation together using TDD approach.

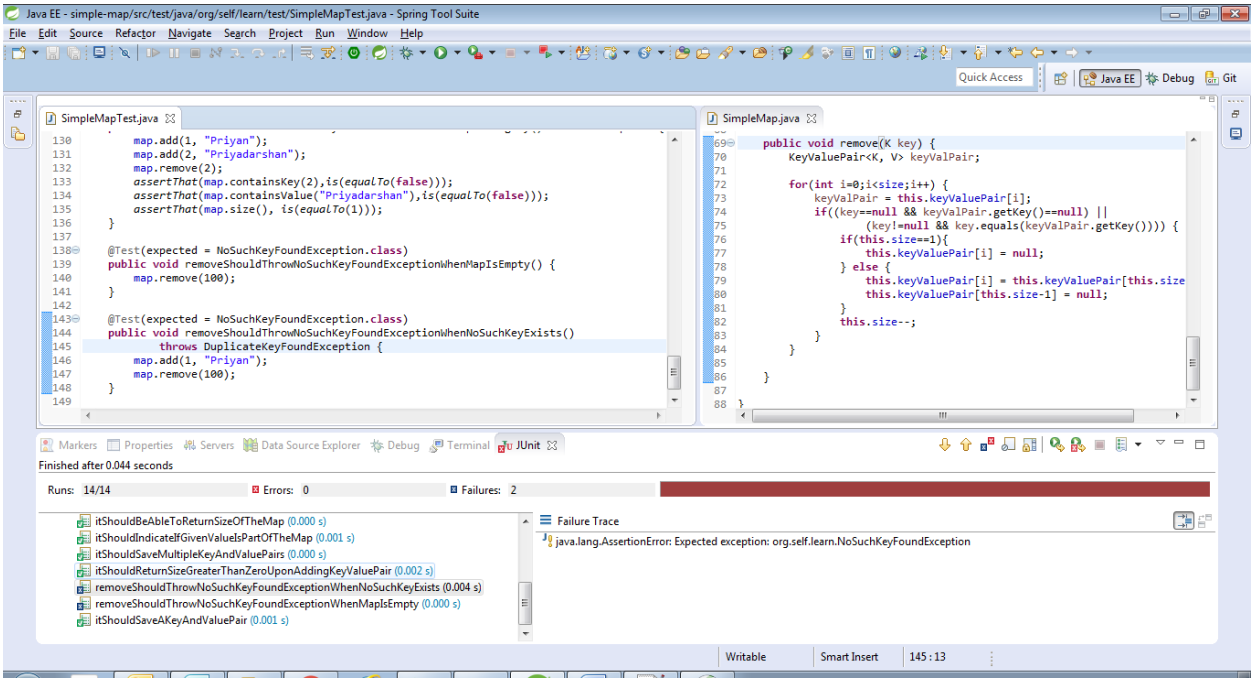




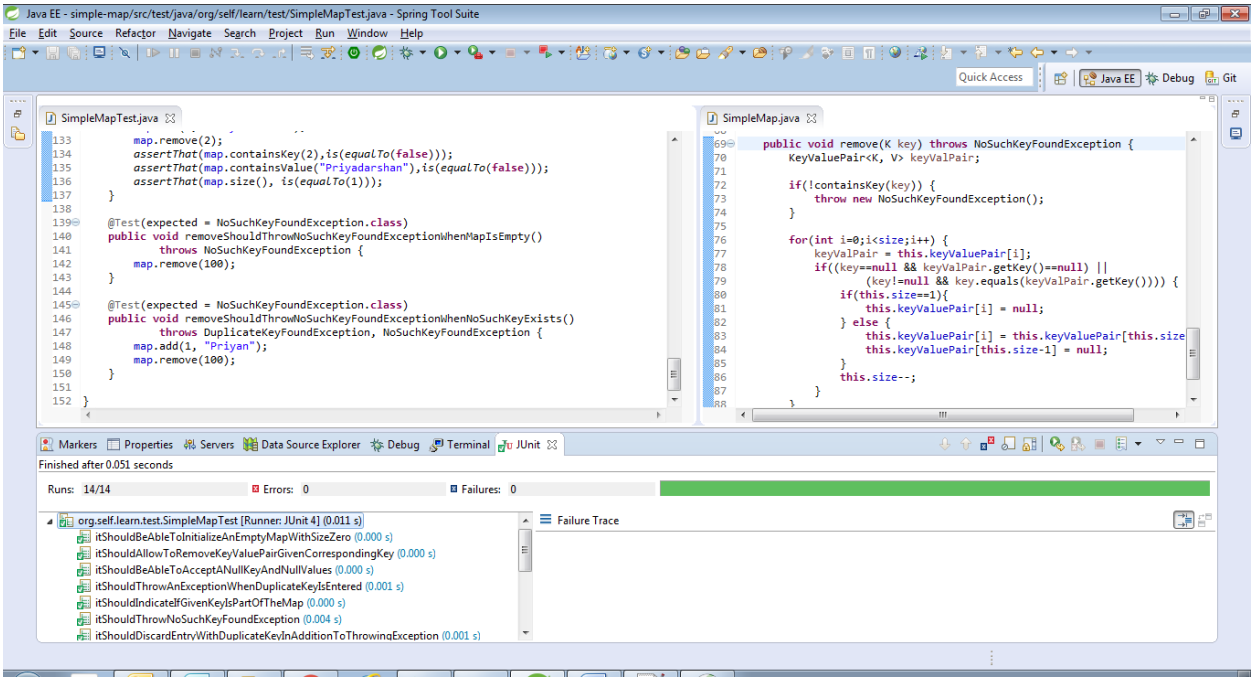
Let's implement the remove method



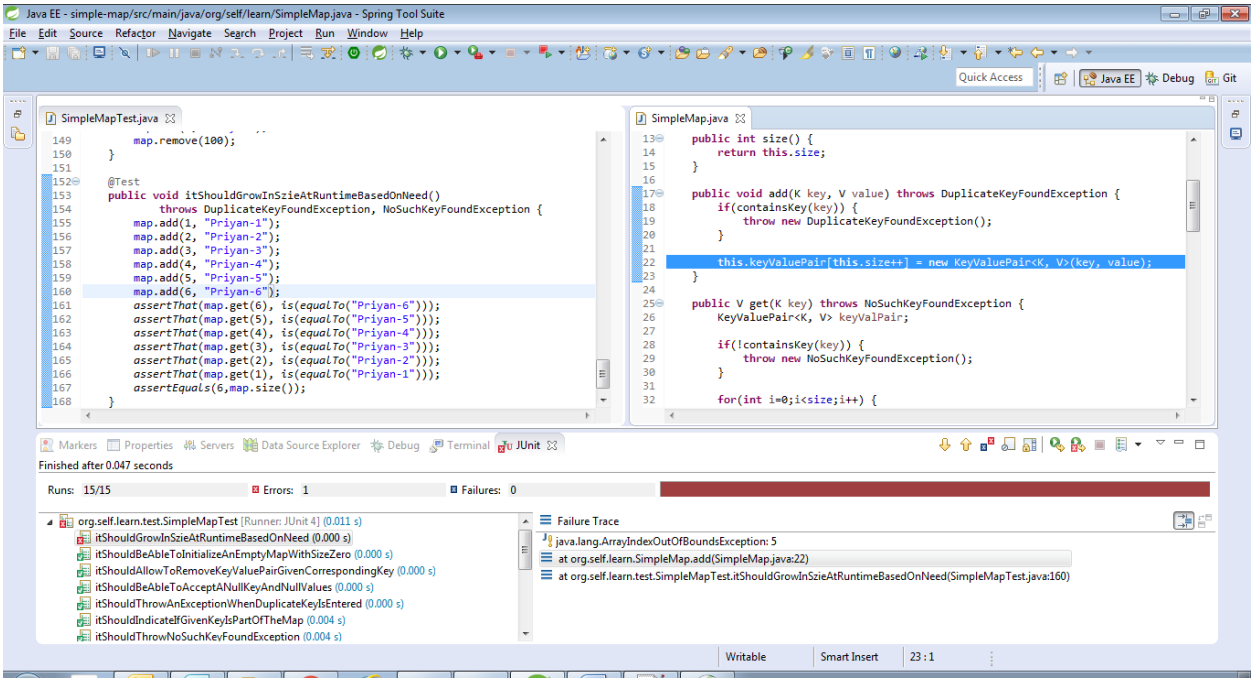
Added couple of more tests for surfacing expcetion when key is not found. Tests are failing at this point.



Here we go with the fix… and that should be it for remove.



The final one is support for dynamic size. I have added six key/value pairs to the map to end up with ArrayIndexOutOfBoundException which lead to test failure.



Following is the completed test and corresponding implementation. Note that all the tests complete successfully at this point. And we have built all desired functionality into our map as well. That concludes this exercise.