

Chapter 12

Testing with JUnit

Hello!

The purpose of this chapter is to explore unit testing with JUnit.

Unit Test Cases

Writing test cases is a vital part of software development. Whether we create whole-program tests, manually and interactively test the program, or write small test cases for the smaller pieces of code, we should always be actively seeking whether our code behaves correctly in all expected scenarios.

Unit tests are test cases that are trying to test as small a part of code as manageable. In Java, this tends to mean that one method's behavior is under scrutiny. We can still write different kinds of unit tests, for instance:

- testing the expected uses with various inputs and checking the outputs (black box testing)
- writing tests that should utilize specific lines of code from the implementation (whit box testing)

We also want to be able to re-run all our tests at will, so that if we make even the smallest change, we can re-check that we didn't break any of the functionality that any passing test cases had witnessed in the past. By writing our tests as actual code that can be executed in a batch, we are performing **regression testing**.

junit is a Java package that provides the framework for both writing unit test cases as well as a controlled means for automatically running all these tests (also as regression tests), recovering from exceptions as needed, and reporting the results of pass/fail for all tests.

In this lab, we will examine some existing code and JUnit test cases to learn more about how to use this particular package. If you find yourself writing code in other languages, the same opportunities should be available – writing small tests; performing regression testing; and quite likely, some other semi-standardized unit testing facility or libraries exist. So take the ideas of unit testing with you far beyond the end of this semester, and save yourself oodles of time – test as you go!

"Installing" JUnit

In order to write test cases using JUnit, we need to have the Java package that provides all of the junit testing pieces available on our classpath when we compile. Grab the file `junit-4.11.jar`, which

should be available next to our lab. (Or if you can download a later version online, that should be equally applicable).

We can always just manually add it to our classpath when we compile and run our tests (shown Mac-style, don't forget to use `;`'s on PC):

```
demo$ javac -cp ../Users/me/path/to/junit-4.11.jar MyTestsClass.java
demo$ java -cp ../Users/me/path/to/junit-4.11.jar MyTestClass
```

JUnit is a convenient-enough thing to always have around that you might think you want to make it automatically show up on your classpath. There are ways to extend your CLASSPATH environment variable to include some directory where you can place jar files like this one, so that they are always available for any use of Java. But there's a decent argument to be made that jar files should not be made globally accessible, as this could break the functionality of already-installed Java applications and other system uses of Java. (The alternative would then be that packages are copied per-project, or managed through some other heavier-weight approach such as Ant or Maven).

So we will just keep using the `-cp` approach. You can copy `junit-4.11.jar` to each project and then have a simpler invocation if you'd like, such as `javac -cp ../junit-04.11.jar *.java`.

Getting Started: Some Sample Code to Test

In the supplied code, there are two files: `SomeCode.java`, and `TestSomeCode.java`. Load them both up and inspect them. `SomeCode.java` includes various method definitions that are perhaps correct (no. They are all wrong.).

`TestSomeCode.java` includes `@Test` methods that will convince us whether the methods of `SomeCode.java` are written correctly or not.

Testing the `isPrime` Method

In `SomeCode.java`, the `isPrime` method checks whether its `int` parameter is prime or not, and returns a boolean. We see a few unit tests over in `TestSomeCode`:

```
@Test
public void isPrime_1() {
    assertEquals(false, SomeCode.isPrime(21));
}

@Test
public void isPrime_2() {
    assertEquals(true, SomeCode.isPrime(29));
}
```

The first two tests show the basics of writing a junit test case: return type should be void, no parameters accepted, and the `@Test` tag just before all the modifiers. With those features in place, it's just a matter of using various `assert*` methods to say "require this equality/boolean/condition to be true". We can also indicate failure with `fail` methods. Below are some of the definitions found in `org.junit.Assert`:

```

public static void assertEquals(int[] xs, int[] ys); //similarly for other array types
public static void assertEquals(Object a, Object b);
public static void assertEquals(int a, int b); // similarly for all other primitive types

public static void assertTrue(boolean expr);
public static void assertFalse(boolean expr);

public static void assertNull(Object obj);
public static void assertNotNull(Object obj);

public static void fail();
public static void fail(String reason);

```

Whenever an `assertTrue` method is called, if its argument evaluates to false the entire `@Test` method is marked as failure. Similarly, whenever an `assertEquals(..)` method is called, if its two arguments don't either `==` each other (for primitive types) or `a.equals(b)` (for object types), then the entire `@Test` method fails. If `fail()` or `fail("some reason")` are ever evaluated, that `@Test` would also be marked as failure. We would of course want to hide these fail-calls behind if-statement logic or something.

The exact mechanics of failure is that an `AssertionError` (an exception) is thrown, which the junit test-bench code then catches, so that it can move on to the next test.

Combining multiple assertions

If you want to include multiple assertions in one `@Test`, you can. But be forewarned, once a single assertion is failed, the entire test is aborted and the later test cases won't even be checked. Similarly, if two different assertions exist along different paths (such as in different branches of an if-else), then by definition they can't both be tested. It would be better to put them in separate `@Test` methods, and then you'll always get feedback on each individual assertion.

```

@Test
public void isPrime_3(){
    // many assertions in one test. not always a good idea, but often convenient!
    assertEquals(true, SomeCode.isPrime(2));
    assertEquals(true, 17);
    assertEquals(true, SomeCode.isPrime(113));
    //assertTrue is simpler for these tests here. Here are more examples.
    assertTrue(SomeCode.isPrime(5));
    assertTrue( ! SomeCode.isPrime(4));
    assertTrue( ! SomeCode.isPrime(12));
    assertTrue( ! SomeCode.isPrime(15));
}

```

Running the Tests

If you are working in DrJava, you can easily run the tests. Just have at least `TestSomeCode.java` open in DrJava, and after compiling click the Test button. You will see the results in the "Test Output" pane at the bottom.

If you are running the tests from the command line, it would look like the following (except that Windows users should use `;`'s, not `:`'s):

```

demo$ javac -cp .:junit-4.11.jar *.java
demo$ java -cp .:junit-4.11.jar TestSomeCode
JUnit version 4.11

```

```

..E.E
Time: 0.01
There were 2 failures:
1) isPrime_2(TestSomeCode)
java.lang.AssertionError: expected:<true> but was:<false>
....<rest of trace clipped>
2) isPrime_3(TestSomeCode)
java.lang.AssertionError: expected:<true> but was:<false>
....<rest of trace clipped>
FAILURES!!!
Tests run: 3, Failures: 2

```

Notice that we needed to make sure junit-4.11.jar was part of our class path. It was in the current directory for this example, but we still had to explicitly add the jar file to the path. Consider the jar file like a folder, and this makes sense – `javac/java` never look in nested folders for us. As always, running the `java` executable on a class means calling its main method; take a look at `TestSomeCode.java`'s main method:

```

public static void main(String args[]){
    org.junit.runner.JUnitCore.main("TestSomeCode");
}

```

It invokes some of the JUnit code on the class named "TestSomeCode". If it had been part of a package, we'd have seen `org.junit.runner.JUnitCore.main("package.sub.sub2.TestSomeCode")`, or something similar. All we really need to know is that the exact contents of the String works as the actual class and its actual entire fully qualified package name, and the rest of this main method is just "boilerplate code" (code that always has a very regular shape, but still must be written each time we want to use some programming idiom or feature).

Your Turn!

- add a `@Test`, checking that negative values are all composite (we'll go with that definition).
- Seeing the error messages, go ahead and fix the definition itself – don't change any test cases (they should all be correct already), just keep fixing `SomeCode` until all tests pass.
→ does this mean our code is always correct? No. But it means it's at least as correct as our test cases are smart enough to consider.

Testing the `fibonacci_iterative` Method

The fibonacci sequence looks like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... It begins with the first two values being 1, and then each later value is the sum of the two previous fibonacci numbers. If we consider the values all stored in some endless array, with the usual zero-based indexing, then we can ask for the n th fibonacci number. `fibonacci_iterative` is a method that calculates the n th fibonacci value using iteration (loops) instead of recursion. (Fibonacci number calculation is a very popular example of when naïve use of recursion is a bad idea).

There is at least one major flaw in this method (can you see it?). Add this `@Test` to your testing code:

```

@Test
public void fibonacci_iterative(){
    assertEquals(34, SomeCode.fibonacci_iterative(8));
}

```

Now run the tests again. Uh-oh, it's stuck in an endless loop! Either hit "reset" in DrJava, or ctrl-C on the command line to quit. How can we still write test cases when we're worried about code never ending? If we know about how long it should take (or just want to give ample time), we can place a timeout on any individual test:

```
@Test (timeout = 1000)
public void fibonacci_iterative() {
    assertEquals(34, SomeCode.fibonacci_iterative(8));
}
```

The timeout number is measured in milliseconds (thousandths of a second). Often any small test case will easily be done before a second, so 1000 or 2000 milliseconds is usually sufficient. In any case, the testing results will indicate that this specific test case timed out, and it is counted as failure.

Your Turn!

- fix the endless loop issue by remembering that *n* must go down by one each iteration. See how the timeout no longer happens. Does the code work yet?
- Add some extra @Tests for `fibonacci_iterative(...)`. What would be good cases? 0, 1, 2? negatives? Can we test them all?
- Convince yourself that you've entirely fixed the code. If that means adding more unit tests, that's fine.

Testing indexOf and dayOfYear

The next two methods are inter-related. `dayOfYear` uses `indexOf` as part of its calculations. Whenever you notice this dependency, it's a good idea to spend effort on the independent one (e.g. `indexOf`), and later try implementing the dependent one (e.g., `dayOfYear`). These are also buggy implementations. Let's first try to make tests, and then we can fix the code itself.

Black Box Tests

Tests that don't consider the particular implementation choices can be called "black box tests", because they treat the method like an opaque, uninspectable thing where all we can do is feed it inputs and see if we get the expected outputs. Let's write some black box tests for `indexOf`.

`indexOf` has the behavior we'd expect: given an array of Strings and a key value, it's supposed to find the *first* occurrence of the key and report back that index-location. If it's not found, it should return -1. We'll start you off with two test cases before you write your own:

```
@Test
public void indexOf_1(){
    String[] words = {"a", "b", "c", "d", "e"};
    assertEquals(3, SomeCode.indexOf(words, "d"));
}

@Test
public void indexOf_2(){
    String[] words = {"a", "b", "c", "d", "e"};
    assertEquals(-1, SomeCode.indexOf(words, "z"));
}
```

One test passes, the other fails. Why might that be? Before we go inspecting the code more closely, try making your own test cases. What are some examples you'd like to try?

Your Turn!

- Think about expected uses, namely when the value exists and we search for and find it. Try creating another example of this style of test.
- Some times there are corner cases; if we have multiple matches, does it find the first one? Make one or more test cases to inspect this.
- Sometimes the sought value won't be present, and it should return -1. Write one or more tests like this.

→ these are all black-box tests; we're not looking at the specifics of the code, just the outsider's view of what `indexOf` is supposed to do.

White Box Tests

Sometimes we choose to write test cases that are directly attempting to test parts of one method. If we take a closer look at `indexOf`, we see that it uses a loop. We should always consider whether it's possible for the loop to run zero times, one time, or many times perhaps. Similarly, there's an if-statement inside that loop. What if the if-statement's guard is true on the very first iteration? Some interior iteration? None of the iterations?

As these examples are trying to show, white box tests are to help us achieve **code coverage**. We want to make sure that every part of the code has been inspected by our tests, so we actually look at the shape of the implementation and try to come up with enough test cases to make sure all possible control-flow paths through the method are considered.

Your Turn!

- Write test cases that focus on the loop of `indexOf`. These are white box tests.
- Write test cases that focus on the if-statement of `indexOf`. These are also white box tests.
- Lastly, target the `return -1` at the end of the method. If you already have such a case, then you can just mentally check this off your list of places to focus upon; but it never hurts to have extra test cases.

Once you're comfortable that `indexOf(..)` is entirely correct and tested, it's time to tackle `dayOfYear(..)`. The assumption is that we could call it for April Fool's Day 2013 as `dayOfYear(2013, 4, 1)`. Notice that January is assumed to be 1, not 0.

Your Turn!

- Come up with test cases for `dayOfYear`. Try to have some black box examples (expected usage), as well as white box examples (code coverage tests).
- Do you have test cases that cover all the corner cases? What would be good corner cases here – negative values, different leap-year scenarios, zero-values, what?
- Run your tests; find the bugs; fix the bugs; ???; profit!
- If you find yourself wanting to add more test cases as you go, that's a good thing – do it.

A note on duplicated setup

You might have noticed that the two black box test cases above had the same `String[]` defined in them. If the setup turns out to be significant overlap between multiple test cases, we can actually push them into one special method that gets called once before all the test cases are run, using the `@Before` tag:

```
String[] words;

@Before
public void setup() {
    String[] words = {"a", "b", "c", "d", "e"};
}
```

We can add instance variables like `words`. Then in our `@Before` method, it gets its value. No matter how complex a setup we want to do, we can just put that method code here. After all `@Before` methods have been run, JUnit then runs all the `@Test` methods. As a mirror to `@Before`, we can also write `@After` methods, which can perform "teardown" operations like closing `PrintWriters` and `Scanners`, writing to log files, or whatever else you want to do after testing.

To be fair, we could also have just written :

```
String[] words = {"a", "b", "c", "d", "e"};
```

And then not needed the `@Before` method at all; it's just a small example of `@Before`, so it's a bit contrived.

If there are multiple `@Before` methods, which one is run first? We don't get any guarantee. If

you're worried about the ordering, just have one `@Before` method that calls your other methods to complete your setup. Not every method in `TestSomeCode` has to have some `@` tag, so using a single `@Before` tag to give you the chance to control the dispatch order is a fine idea.

Case Study: Greatest Common Denominator and Least Common Multiple

Familiarize yourself with the greatest common denominator and least common multiple. Wikipedia is a good resource for this.

http://en.wikipedia.org/wiki/Greatest_common_denominator

http://en.wikipedia.org/wiki/Least_common_multiple

In brief, from Wikipedia

- $\text{GCD}(a,b)$ is the largest positive integer that divides both a and b without remainder.
- $\text{LCM}(a, b)$, is the smallest positive integer that is divisible by both a and b .

There are two more methods in `SomeCode.java` which implement GCD and LCM:

```
public static int gcd (int a, int b) {...}  
public static int lcm (int a, int b) {...}
```

Your Turn!

Use your newly acquired unit testing skillz to:

- Create an effective set of test cases which would convince you that the code is correct if they all pass. What corner cases are there, and what behavior is expected? For instance, what if $a > b$, or $a < b$, or an input is negative/zero?
- Run these tests. Witness some errors.
- Even run the debugger a bit to try to enrich the bug-hunting process more!
- Do you want to add any more tests? Go ahead, especially if you find more corner cases that you didn't test before.
- Once all test cases are passing and you don't think you need any more, stop one more time and think: is this code entirely correct? Are there any other behaviors I should consider, or am I truly convinced that the code is correct?

Old Materials: installing Eclipse, JUnit, tests based on regular expressions.

Installing Eclipse

This page is provided in case you want to install Eclipse. We will be using JUnit directly, whether you use it at the command line, with DrJava, or Eclipse (or any other compatible IDE). First, go to this link <http://www.eclipse.org/downloads/> and download "Eclipse IDE for Java Developers". Follow the installation instructions.

Eclipse is written in Java, and has support for writing code in many languages. It has many extra features (common to most IDEs), such as code completion, type checking in the background to immediately bring up errors or warnings, and management of your packages and files.

One reason we haven't explicitly focused on using Eclipse at first is that it also creates extra files and folders, and can make it confusing what files and folder structures are actually required to be writing Java code. It also blurs the distinction between editing, compiling, and executing, because you get a nice, shiny "play" button that will compile and run your code all at once, if it can. Some future programming situations will explicitly require you to write code command-line style, so we wanted to cement comfortability with that mode of programming first.

When you open Eclipse, you'll need to specify your workspace. This is a directory where Eclipse can create "projects". It's important that you manually "close project" whenever you go between projects; the play button sometimes is trying to run the other project. When it's time to work on another project and you want to close any other projects, in the "package explorer" panel, right-click the project and "close project", so that just the one you want to create is open.

We think of each assignment or program as a "project". We can create a new project via File → New → Java Project. We similarly add classes with File → New → Class. Eclipse tries to be helpful with all sorts of checkboxes and things, but you can always just type things in manually later on, in case you forget to check "create method stubs for inherited abstract methods", or something similar. Eclipse will create a folder for the project in its workspace directory, and inside of that, all your code will go by default into a "src" directory (source).

Installing JUnit in Eclipse

Let's install JUnit. Go here, <http://www.junit.org/>, to download the jar file named junit-4.11.jar. (If this lab gets outdated and there's a later edition, grab that one instead! ☺)

We need to add this jar file to your project's class path. In Eclipse, go to:

- File → Preferences (or, Eclipse → Preferences on Mac)
- select the Java menu on the left
- open the build path submenu
- edit the classpath variables submenu

- "Add", and select your file. It's a good idea to put the file somewhere like a bin directory: ~/bin, ~/local/bin, maybe /users/shared/bin, ... anywhere is actually okay, as long as it's added to your classpath and you don't go moving it later on.

An alternate way to add JUnit to a project: go to Project → Properties → Java BuildPath left-menu → Libraries tab-header → Add External JARs → Follow the dialog and add your .jar file.

Testing with JUnit

In order to perform testing with JUnit, we need something to test. Let's create a single class that has multiple static methods that main can call, each one trying to utilize a regular expression.

Let's start out with the following code:

```
import java.util.*;

public class RE {

    public static void main(String[]args){

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter a phone number: ");
        String input = sc.nextLine();

        boolean wasPhoneNum = checkPhoneNumber(input);

        System.out.println("\nThat was"
                           +(wasPhoneNum?"":"n't")
                           +" a phone number.");
    }

    public static boolean checkPhoneNumber(String s) {
        return s.matches ("(\\d{3}) \\d{3} - \\d{4}");
    }
}
```

In Eclipse, select File → New → Java Project, name it (perhaps Lab8), and then hit OK. Next, we want to add a class: File → New → Class. Name it (RE was used above), hit OK. You can now paste or retype the class definition provided above.

First off, fix the compilation error. It relates to the issues of actually representing our regular expressions inside a Java String. What does \\d mean to Java when we're creating a String? Nothing intelligible; we meant to have a backslash, then a d, inside the String, so we have to represent each individually. The answer is in the footnote 1.

1 represent it as "\\d", so that "\\" means backslash, and "d" means d.

In general, write your regular expression on paper (which might contain backslashes), and then symbol-by-symbol, represent it as a Java String. You'll also have to escape double-quote characters if they are a part of your regular expression.

Initial coding: writing something worth testing.

Okay, back on task. Let's test this regular expression with the following inputs:

```
(123)123-1234  
(123) 456 - 7890
```

Why aren't these working? What's wrong with our regex? Let's try another:

```
123 123 - 1234
```

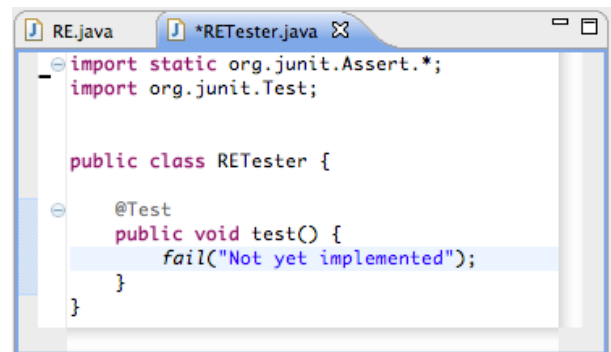
Ah, parentheses didn't mean what we thought. We forgot they are special symbols. Let's modify the regular expression (escape the parentheses), and test again:

```
(123)123-1234  
(123) 456 - 7890
```

Okay, we're getting closer. We'd like to allow whitespace anywhere except between adjacent number characters. Try fixing the regular expression to your heart's content, and then let's move on to testing it with JUnit.

Testing our method with JUnit

Let's use Eclipse's handy menus to help us create a JUnit test class: navigate to File → New → JUnit Test Case. We are actually creating a regular Java class, where the testing code will live. Name it something like RETester, and hit OK. If asked if you should add junit to the build path, say yes! Otherwise, we had instructions at the top of this section to add the jar file to a project.



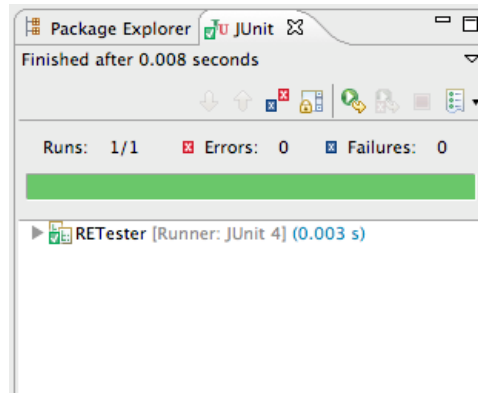
let's replace the *fail* line with the following:

```
assertTrue(RE.checkPhoneNumber("(123) 456 - 7890"));
```

We now have our first JUnit test. Its purpose is to check that when we call RE's checkPhoneNumber method on particular inputs, it returns true.

Near the Package Explorer tab, there should be a JUnit tab. (If not, go to Window → Show View → Other → Java → JUnit). We want to "run as JUnit Test". Either use the narrow expander next to the

green "play" arrow, or use the menus (Run → Run as → JUnit Test). When we run, we should see something like this:

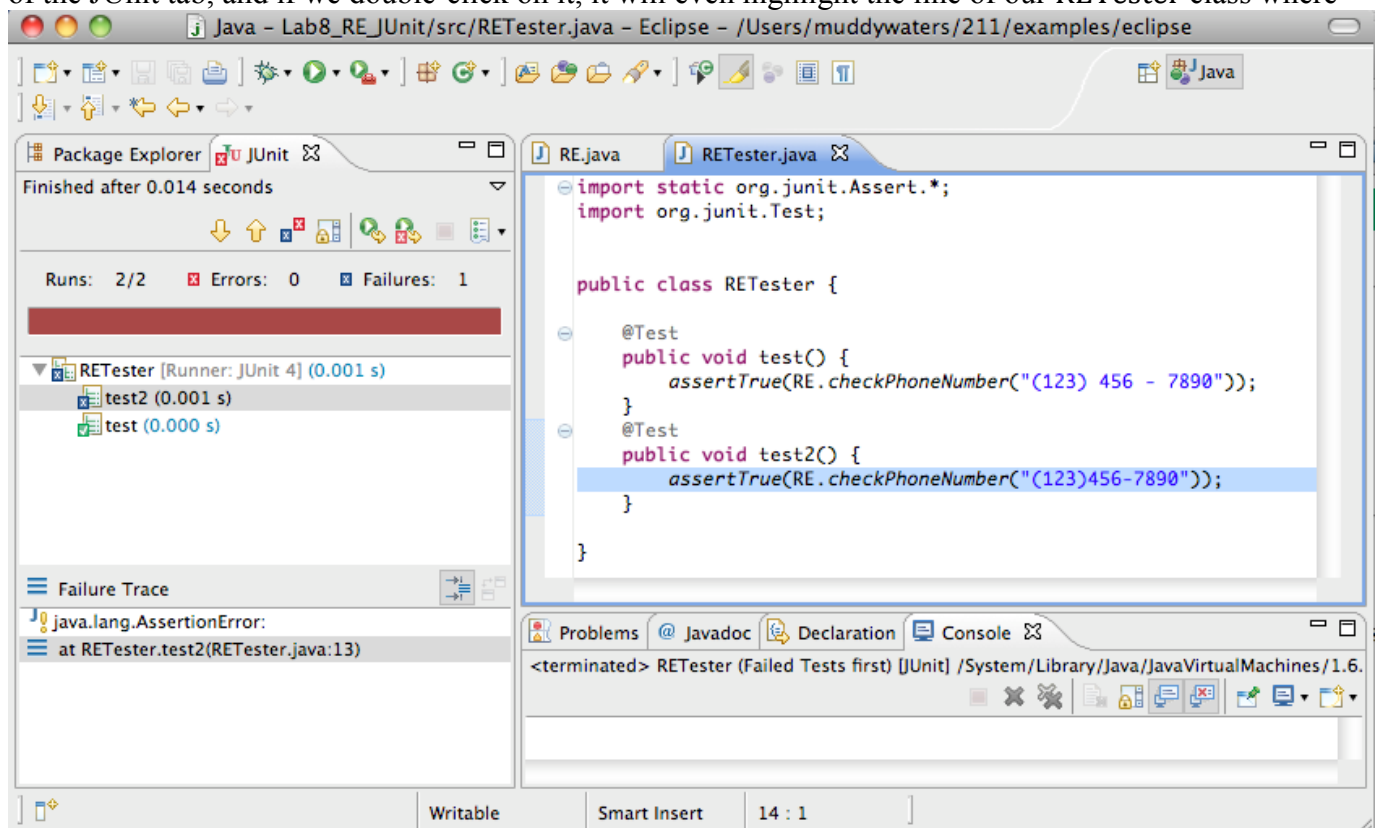


Hooray! All one of our tests passed. Let's create another test case. In order to do so, we just create another `@Test` method in our class. We can find many alternatives to the `assertTrue` method in the documentation here: <http://junit.org/apidocs/org/junit/Assert.html> For now, let's just test more inputs by :

```
@Test
public void test2() {
    assertTrue(RE.checkPhoneNumber("(123)456-7890"));
}
```

Next, we run our tests again. The play button in Eclipse will remember that we last ran our JUnit tests, and will still do that instead of running our main method (until we change back with another "run as" selection).

We see a lot of information this time: we see a big red bar, indicating failure; we see one test with a checkmark next to it, and our new test (test2) with an x on it. We can also see the "failure trace" portion of the JUnit tab, and if we double-click on it, it will even highlight the line of our `RETester` class where



the assertion failed.

Overall, we're seeing that we can make lots of test cases, and then re-run them all conveniently. Now, let's try to add in whitespace wherever it's convenient: let's add "\\s+" thus:

```
return s.matches ("\\((\\s+\\d{3}\\s+\\)\\s+\\d{3}\\s+-\\s+\\d{4})");
```

We can now conveniently re-run all our test cases, and see how we fared. We got the same results: test passed, test2 failed. Let's make things less restrictive, and use "\\s*" instead of "\\s+". Yay! Now our test2 case passes.

Your Turn!

- Add more test cases: specifically, add cases that should **not** be valid phone numbers. Remember, we should always ask ourselves two questions:
 - "do I accept enough strings with my regular expression?"
 - "do I accept too many strings with my regular expression?"
- Try using more methods from **org.junit.Assert**. For instance, assertFalse and assertNotNull both might be useful starting points in your testing (in general, not just or specifically for this lab tutorial).

Your Turn!

- Create another static method in your main class, named validateEmailAddress. It should work like checkPhoneNumber did, using a regular expression. Then, create multiple JUnit tests to see if you think it accepts enough strings, as well as not too many, by relying on the results of your JUnit tests.
 - Notice at this point we don't even have to call the method from the main method; we can just happily test away on a small, isolated part of our code! Testing at its finest...

Another Case Study

Let's look at another case for testing. We will have a Triangle class that contains a calculateArea method. We want to test that method; it gets the math wrong for Heron's formula.

```
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class Triangle {
    public int side1, side2, side3;
    public Triangle(int side1, int side2, int side3) {
        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
    }

    public double calculateArea () {
        //Heron's Formula for area of a triangle
        double s = (side1 + side2 + side3)*0.5;
        System.out.println("\ts="+s);
        double result = Math.sqrt(s * (side1-s) * (side2-s) * (side3-s));
        System.out.println("\tresult="+result);
    }
}
```

```
        return result;
    }
}
```

Add the above class definition to a package in Eclipse. Again, let's create a new JUnit Test Case. Fill it in with the following code.

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.*;

public class TriangleTest2 {

    public Triangle t1, t2, t3;

    @Before
    public void setupStuff() {
        t1 = new Triangle (3,4,5);
        t2 = new Triangle (5,4,3);
        t3 = new Triangle (8,5,5);
    }

    @Test
    public void test() {
        assertTrue(t1.calculateArea() == t2.calculateArea());
    }

    @Test
    public void test2() {
        assertTrue(t3.calculateArea()==12);
    }
}
```

We are using another feature of JUnit: a method annotated with the `@Before` annotation will be called, once, before then calling each `@Test` method once. In this way, we were able to just use `t1`, `t2`, and `t3` in our `@Test` methods without having to manually declare and instantiate them inside each method. Note that we still had to declare `t1`, `t2`, and `t3` as instance variables of the entire `TriangleTest` class. This allows us to share the setup work that prepares us for running the actual assertion checks that our tests want to perform.

We could have actually put both assertions in the same method:

```
@Test
public void test() {
    assertTrue(t1.calculateArea() == t2.calculateArea());
    assertTrue(t3.calculateArea()==12);
}
```

But when the first one fails, we don't get to even attempt the second assertion; it's more disciplined to have separate methods for each test case, though it's perhaps more convenient to have multiple assertions in the same test method.

Your Turn!

- Use your understanding of JUnit to debug the issues plaguing our implementation of Heron's formula. Feel free to look up the original formula as part of your investigations.
- Make sure you have both "positive" and "negative" test cases: ones that show the code serves its intended usage, and others that show it can't be misused.
 - What will happen if the user creates a triangle with new Triangle(3,4,100) ? Is this something you should or can test in JUnit? How might you do so? It's possible to add any normal bit of code in our testing class. Hopefully the testing class doesn't get so complicated that we need to test our testing code.