* **Creating Test Methods**

Now that we’ve got our classes configured, let’s get to making some tests. In a typical project, we’d have some tasks that we knew the program would need to be done. Since we’re not working toward any particular goal, let’s instead look at the kinds of things we can test, and make up some methods that allow us to make use of these.

A good way to write tests is to “manage expectations” and “validate results.” What this means is that we may want to do a little prep before calling our methods, and then check the results after our call, to be sure expected things have happened. Let’s make some simple examples of prep and validate testing.

Let’s suppose we have a need to have a method to verify that a string meant for a password meets a few criteria: it must be longer than six characters (to be hard to crack), shorter than 20 characters (since that’s all the room we have to store them in our database), and must not contain spaces (just to have something negative to check for), must contain at least each of a one capital letter, lower-case letter, number, and “special” character (such as periods, exclamation, question, octothorpe…most of the other things we can easily type). Let’s use a test-driven methodology, as it lets us focus on writing the tests first.

So that we don’t have a compile failure, let’s add a stub for our method to our main class. We’re validating a password, so it’ll just take a String and return a boolean indicating its adherence to our rules. In our base class, let’s create the method validatePassword() as something like this:

public class Foo {

public static boolean validatePassword(final String password){

return false;

}

}

Note that it does return false, meaning everything we give it will be deemed unacceptable. After we write some tests, we’ll fill in our method to be sure they all pass. The first round, we’ll do as we write the test, but then we’ll fly through some tests and satisfy them all at once.

Also, I’ve cheated and made this method static, just to make it easier to test. Of course, in a more fully-functional class, there’d probably be instance variables and private or protected worker methods.

For a first test, let’s verify that the password must have a value. We can test this by passing it a null parameter and an empty string. In our test class, let’s create a method to test null passwords. The convention for test methods is to start the name with the method being tested, and then extend it with a description of the test. Since we’re validating the validatePassword() method with null passwords, we’ll create a method validatePasswordNull(). To tell JUnit that this is a test method, we’ll annotate the method with @Test (adding the necessary import as well). Since we expect null passwords to fail, we’ll simply test our assumption using the assertFalse() test. This gives us the following test class:

import static junit.framework.Assert.assertFalse;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.JUnit4;

@RunWith(JUnit4.class)

public class FooTest {

@Test

public void validatePasswordNull() {

assertFalse(Foo.validatePassword(null));

}

}

This we can run using the trick mentioned above (Run menu, Run As, JUnit Test). Instead of a failure, we’ll see that we pass our test! This is a little inconclusive, however, as the validatePasword() method will always return false, but it’s a good start and gives us a little code to defend. In a moment, we’ll see that we’ll need to add a check for null to our method, as this test will cause a failure if we try to act on our null password! But let’s not give too much away.

Let’s add the second test we’ve already mentioned, for an empty string.

import static junit.framework.Assert.assertFalse;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.JUnit4;

@RunWith(JUnit4.class)

public class FooTest {

@Test

public void validatePasswordEmptyString() {

assertFalse(Foo.validatePassword(""));

}

@Test

public void validatePasswordNull() {

assertFalse(Foo.validatePassword(null));

}

}

Again, running the test will show that the tests both pass, as expected. The astute reader will note that these two tests were not strictly in our requirements. One can argue that a null or empty string fall outside of our length parameters, but I really wanted both a quick start and a couple of tests that will cause trouble, as I purposefully expose some potential pitfalls as we progress. With that, let’s move onto our requirements, and with this, we’ll start adding to our base class to make them work.

Let’s make a method that makes sure our password is longer than 6 characters. We’ll simply create a method that checks strings of shorter lengths ensure they’re not valid. Here’s where programmer’s preferences start to kick in. We can make a number of separate test classes, each passing in strings of different lengths. We can make one method making several validatePassword() calls. We can put this in a loop. Some may see the importance of the value of the string and want to mix that up, too. Since the rule is that a password must be longer than 6 characters, the content doesn’t matter, and the method of calling doesn’t matter, so I’m going to be concise and make one method that simply sends a string too short to pass by passing 5 characters. Since this is similar to the rule for too long, I’m going to make a simple method that tests for a string too long by passing 21 characters.

@Test

public void validatePasswordTooLong() {

assertFalse(Foo.validatePassword("123456789012345678901"));

}

@Test

public void validatePasswordTooShort() {

assertFalse(Foo.validatePassword("12345"));

}

To keep the post from getting way too long, I’ve just included the test methods here; inject them in the test class. Running the test still shows the validatePassword() method works as expected. So far, so good. Of course, we haven’t yet had to alter our base class beyond adding the stub. What we’ve asserted is only that our set of bad passwords will not be validated. Let’s quickly throw the rest of the expected failures in otherwise correct-length passwords:

@Test

public void validatePasswordNoCaps() {

assertFalse(Foo.validatePassword("!10therwisevalid"));

}

@Test

public void validatePasswordNoLowers() {

assertFalse(Foo.validatePassword("!10THERWISEVALID"));

}

@Test

public void validatePasswordNoSpaces() {

assertFalse(Foo.validatePassword("!1Otherwise Valid"));

}

@Test

public void validatePasswordNoSpecial() {

assertFalse(Foo.validatePassword("M1ss1ngSp3c14l"));

}

So now we’ve got a test class with eight methods that validate all of the cases we would expect to fail. We’ve got no string and empty string. We’ve got too-short and too-long strings. We’ve got a set of strings that are the right lengths, but are each missing critical parts. We’ve got a string with the right length, and right parts, but with a space in it, which is not allowed. Running the test will pass, and we can see that all of the invalid passwords are correctly failing the validation. Of course, everything fails as our method only returns false.

Let’s then add one final test method that should pass, which we know will fail, which will then allow us to complete our base class.

@Test

public void validatePassword() {

assertTrue(Foo.validatePassword("Good1!"));

assertTrue(Foo.validatePassword("Expect4Success!"));

assertTrue(Foo.validatePassword("1LongEnough?YesItIs!"));

}

This method has everything we expect. The first string is six characters, which is our minimum. The middle string is 15 characters, which is more than 6 and fewer than 20. The last string is 20 characters, which is our maximum. Each contains no spaces. Each has at least one each of lower-case and upper-case letters, numbers, and special characters. Note the assertion has been changed to True (and we’ll have to include the correct import). Below is the set of full test cases that we have to ensure our validatePassword() method works.

import static junit.framework.Assert.assertFalse;

import static junit.framework.Assert.assertTrue;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.JUnit4;

@RunWith(JUnit4.class)

public class FooTest {

@Test

public void validatePassword() {

assertTrue(Foo.validatePassword("Good1!"));

assertTrue(Foo.validatePassword("Expect4Success!"));

assertTrue(Foo.validatePassword("1LongEnough?YesItIs!"));

}

@Test

public void validatePasswordEmptyString() {

assertFalse(Foo.validatePassword(""));

}

@Test

public void validatePasswordNoCaps() {

assertFalse(Foo.validatePassword("!10therwisevalid"));

}

@Test

public void validatePasswordNoLowers() {

assertFalse(Foo.validatePassword("!10THERWISEVALID"));

}

@Test

public void validatePasswordNoSpaces() {

assertFalse(Foo.validatePassword("!1Otherwise Valid"));

}

@Test

public void validatePasswordNoSpecial() {

assertFalse(Foo.validatePassword("M1ss1ngSp3c14l"));

}

@Test

public void validatePasswordNull() {

assertFalse(Foo.validatePassword(null));

}

@Test

public void validatePasswordTooLong() {

assertFalse(Foo.validatePassword("123456789012345678901"));

}

@Test

public void validatePasswordTooShort() {

assertFalse(Foo.validatePassword("12345"));

}

}

If we run this test, we’ll get eight passes and one failure. The failure is currently for our expected successful call with the valid password. This has given us a set of tests we can use to verify that we won’t make software that breaks our expected failures either.

So let’s start addressing our base class, and try to make a method that meets all of our test requirements. As a reminder, the requirements are that it must be longer than six and shorter than 20 characters, and must not contain spaces, must contain at least each of a one capital letter, lower-case letter, number, and special character.

public class Foo {

public static boolean validatePassword(final String password) {

boolean special = false, digit = false, upperCase = false, lowerCase = false;

if (password != null && password.length() > 5 && password.length() < 21) {

for (int index = 0; index < password.length(); index++) {

final Character character = password.charAt(index);

if (Character.isWhitespace(character))

return false;

else if (Character.isLowerCase(character))

lowerCase = true;

else if (Character.isUpperCase(character))

upperCase = true;

else if (Character.isDigit(character))

digit = true;

else if (33 <= character.charValue() && 127 >= character.charValue())

special = true;

}

}

return special && digit && upperCase && lowerCase;

}

}

Here, again, a lot of programmer preference may be used to determine a best way to do it. Probably the shortest and easiest would have been to use a Pattern or regular expression, but this allows us to see each bit, and compare it to our tests.

Looking at our method, we can see that we make a placeholder for each type of desired character in our password; each is set to false as we’re starting with the assumption that they don’t exist and we’ll be working to prove otherwise. We start by checking that the password is the right length. Note also that there’s a quick null-check there, too; this wasn’t in the requirements, ‘though we did write a test for it, but it is essential for program safety as the password.length() would throw a NullPointerException if we didn’t check and a null was passed. Then we iterate over each character in the string, checking to see if it meets our requirements. Upon finding a space (or other whitespace character such as tab or newline) we stop and return false. There is a little cheat in the “detection” of special characters as we assume any ASCII printable character that isn’t a number or letter (found in the earlier branches of our conditions) is acceptable.

If we run the test methods now, we’ll see that all nine of them pass. All three of our valid passwords are accepted, and none of the other eight expected failures are.

Source: http://www.objectpartners.com/2010/12/21/how-to-write-junit-tests/