Exercise 5 - Test Driven Development

Objective

The objective of this exercise is to perform some test driven development using scala and specifically, with the IntelliJ IDE. If you want are using eclipse then you need to install the scalatest plugin for your project to get the context menu items for test. If you are using sbt then there is a ‘test’ command you can use to run the tests in the code.

References

Chapter 5 of Slides: 94 - 116

Overview

To setup scalatest for IntelliJ we need to add a ‘test’ folder to the project.

* Download the scalatest jar from scalatest.org. You need to make sure you have the correct version to match the version of scala you are running. We should be using 2.11, but sbt uses 2.9. It is very important to get the correct version!
* Right click the project name and select new folder, call it anything you like, but ‘test’ is traditional.
* Right click the new test folder, go to ‘Mark directory as’ and select ‘Test Sources Root’. This will enable IntelliJ to find your tests

Now we need to add the scalatest jar and the scala xml jar, open the project structure from the file menu (or on windows the shortcut key is Ctrl, Shift, Alt and S)

* Go to modules on the right, then the dependencies tab. Add a new dependency by clicking on the + symbol at the bottom of the screen and select new Jar
* Navigate to where you have downloaded scalatest.jar and include it in the project. Click apply
* Add another jar by clicking the same + icon as before. Navigate to the directory where scala has been installed. In the scala/idea/lib directory there should be a file called scala-xml\_2.11-1.0.2.jar or similar. Add this to the project.

The scalatest website has a set of standard tests we can use to see if we have setup IntelliJ correctly. Create a new file called ExampleSpec in your test folder and add the following code:

import java.util.NoSuchElementException

import collection.mutable.Stack

import org.scalatest.\_

class ExampleSpec extends FlatSpec with Matchers {

"A Stack" should "pop values in last-in-first-out order" in {

val stack = new Stack[Int]

stack.push(1)

stack.push(2)

stack.pop() should be (1)

stack.pop() should be (1)

}

it should "throw NoSuchElementException if an empty stack is popped" in {

val emptyStack = new Stack[Int]

a [NoSuchElementException] should be thrownBy {

emptyStack.pop()

}

}

}

Run the test by right clicking in the file and selecting ‘run’. If the test is failing to run due to scala/xml/MetaData not being found, remember to add the XML dependency to your project outlined above.

If everything is setup correctly the output should say that everything is okay and 2/2 tests completed successfully.

Writing our tests for existing code

1. Write some tests for the bank account class we have been working on in the exercise. Check boundary cases for each of the methods as well as some random fields. It is useful to know if you can withdraw £50 from your account if you only have £50 in there to start with (using the >= symbol in the if statement rather than just the > statement changes the behaviour of the withdraw method!)
2. Write a test for the intPower algorithm that calculates x^n

def intPow(x: Double, n: Int): Double = {  
 // n is >0 and even  
 if (n > 0 & n % 2 == 0) { val y = intPow(x, n/2); y\*y }  
 // n is odd...  
 else if (n > 0) x \* intPow(x, n - 1)  
 else if (n == 0) 1  
 else /\*(n < 0)\*/ 1 / intPow(x, -n)  
}

1. Test the function (or reuse your original one) that finds if a year is a leap year.

Test Driven Development

The principle behind test driven development is that we write the test and ensure that it fails before we write the code to satisfy the test. Follow this pattern while completing this set of exercises. You can use any of the types of tests available with scala test. Try to mix them up and use multiple styles. You can also check if your tests are valid by deliberately breaking some of the code!

1. Write a function that checks if an email address is valid. You should check that there is an @ symbol present and at least one period (full stop symbol).
   * Note: This won’t actually guarantee a valid email address; to do this we would need to use regular expressions, something outside the scope of this exercise!
2. Write a class which stores five numbers and has the following methods
   * max – returns the largest number
   * min – returns the smallest number
   * average – returns the average
   * product – returns the product of the numbers (all the numbers multiplied together )
   * repeating – returns true or false based on if any of the numbers are the same
   * Hint: Look at the scala.math api to see if there is anything you can use for this!
3. Create a class that has methods that implement the Luhn algorithm. This is used to calculate if a credit card number is valid or not. More information about it can be found online, including fake credit card numbers to try with your code.

http://www.freeformatter.com/credit-card-number-generator-validator.html

The algorithm has the following steps:

|  |  |
| --- | --- |
| Take a credit card number | 4 5 5 6 7 3 7 5 8 6 8 9 9 8 5 5 |
| Remove the last digit | 4 5 5 6 7 3 7 5 8 6 8 9 9 8 5 |
| Reverse the number | 5 8 9 9 8 6 8 5 7 3 7 6 5 5 4 |
| Double the odd numbered digits in the list (if indexing from 1. As computer scientists index from zero we start from zero and double the even numbered digits. The pattern is [double, leave, double, leave etc] | 10 8 18 9 16 6 16 5 14 3 14 6 10 5 8 |
| Subtract 9 from the numbers over 9 | 1 8 9 9 7 6 7 5 5 3 5 6 1 5 8 |
| Add the numbers together | 85 |
| Mod 10 | 5 |
| Add this to the number at the end of the original card number. If this is divisible exactly by 10 then you have a valid number! | 5 + 5 = 10  10 % 10 = 0  Valid! |

You can split your class up how you like, here is one option:

class luhn {  
  
 def removeLastDigit(n: String)   
 def reverse(n : String)   
 def doSums(n: String)   
 def mod10(n : Int)  
 def checkValid(n: Int, finalNumber: Int)

}

val number = "4556737586899855"  
  
val l = new luhn  
  
val step1 = l.removeLastDigit(number)  
val step2 = l.reverse(step1)  
val step3 = l.doSums(step2)  
val step4 = l.mod10(step3)  
l.checkValid(step4, number.charAt(number.length-1).asDigit)

Hints:

* Some parts of this exercise are easier to do treating the number as a string, such as removing the last digit and reversing the list of numbers. You can then iterate over every character in the string.
* To convert from characters to numbers use the asDigit method, NOT asInt. asInt will return the ASCII character code for the character.
* Generate tests for each step of the algorithm and check that each part is working before moving on.
* Doubling, subtracting nine and summing the numbers can be done in one for loop. If you try and do it individually you will need a list object, and we’ve not looked at those yet!

This last exercise is a tricky one, so good luck!