CPSC 150L Lab 9
jUnit and Debugging

Fall 2017

1 Introduction

This lab will focus on debugging and testing using jUnit.

2 Exercises

Fork and clone the `cpsc150l-lab09` repository from the 150 students group on Gitlab.

2.1 Debugging `PersonName`

In the `cpsc150l-lab09` repository, there is a file called `PersonName.java` that contains a method called `numberNames`. However, this method has errors in it. You need to debug the code so that it passes all of the tests on WebCAT. The specification for the method is as follows.

- `public int numberNames(String wholeName)`

  Names traditionally have a number of names within them, e.g. a name can be comprised of a first name, middle name, last name, and suffix (Jr., Sr., etc.) among others.

This method should take a `String` parameter that contains a person's name. It will then return the number of names in `wholeName`. Moreover, the method should ignore extraneous whitespace at the beginning, end, and in between names. Consider the following table for example input and output.

s	numberNames(s)
"Matt"	1
" Matt "	1
"Joe Smith"	2
" Bob L. Smith"	3
"Bobby Paul Smith Jr."	4

Upload your code to WebCAT under Lab09 via web or push to Gitlab to test it.

Exercise 1 Complete

Run:

```
git add .
git commit -m "Completed exercise 1"
git push origin master
```

2.2 Making jUnit Tests for `PersonName.numberNames`

For the second part of this lab, create a test class named `PersonNameTest`. Take each case provided in the `PersonName.main` method and create a jUnit test in `PersonNameTest` that tests that case. If you help writing jUnit tests, refer to Section 5.2. By the time you finish, you should have 5 test methods in `PersonNameTest.java`.

Ensure that the tests that you wrote pass.

Question 1: *Show your lab instructor the jUnit tests you wrote and push to Gitlab.*

Exercise 2 Complete

Run:

```
git add .
git commit -m "Completed exercise 2"
git push origin master
```

---

3 Hints

1. Use the BlueJ debugger to step through the program and monitor the internal variables
2. Add print statements inside the `numberNames` method to track progress

4 Common Mistakes

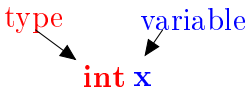
Some solutions to common mistakes for this lab are as follows.

1. Do **not** make `PersonName.numberNames static`!
2. Do not test multiple cases in one test method!
3. Remember to use the `@Test` annotation when declaring a jUnit test method.
4. In order for a test to pass, you must assert something.

5 Tutorial

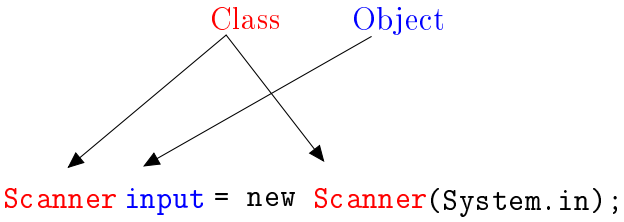
5.1 What is an Object?

Before moving forward, let's talk about objects and classes. Objects and classes are similar in their relationship to variables and types, i.e. Object is to variable as class is to type. When we declare an int, it looks like this:



where the first word describes the type and the second word is a variable or an instance of the type.

In previous programs you have seen the use of objects like the scanner. The word "class" is like a type and the word "object" is like a variable.



In this example, a variable (object) named `input` of type (class) `Scanner` is created in order to read input from the keyboard. Although we are ignoring the "`System.in`" part, this too will be explained before the end of the lab.

After creating the "`input`" object, we used input to read data in statements like this:

Code:

```
int i = input.nextInt();
```

If you look at the API (`https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`), you'll see other Scanner methods, like `nextLine` and `hasNext`.

As you can see, this class includes a large number of functions that can be used. You were able to use the Scanner class in prior programs without knowing much about what Scanner really can do, but the interface was right because you didn't need to know it and weren't forced to use it... the complexity matched the need.

5.2 Writing jUnit Tests

When writing a program, we need to ensure that the answers that the program produces are correct. This is where testing comes into play, but how do we properly test code? The first method is to try every single input combination possible (called *exhaustive testing*), however this is usually too much work. For example, consider a method that sums two integers. In Java, integers can range from -2147483648 to 2147483647. Thus we have 4294967296^2 different combinations to test. If we suppose that each run takes 1 microsecond (10^{-6} seconds), then the entire test will take 214 days. This is why we prefer to do a different kind of testing, called boundary value testing.

Suppose we are writing a program that determines someones pay at the end of each week. We need to test cases where an employee worked less than 40 hours, exactly 40 hours, and more than 40 hours. While this form of testing (and most others) are incomplete, they give us a nice way to determine if a program behaves the way we want it.

5.2.1 How to Write a jUnit Test

Suppose we are writing a method

Code:

```
public static boolean lessEqual(int a, int b)
```

that returns whether or not $a \leq b$ in a class named `Comparisons`. Before you can write jUnit tests, you must first create a *test class*. For this example, call the test class, `ComparisonsTest`. In the header of the class type the following lines.

Code:

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

This will enable you to declare test methods and use assertions to test the code. We declare a test method using the `@Test` *annotation* like follows.

Code:

```
@Test
public void lessEqualsTest1() {
    // Test code here
}
```

In jUnit if we want to *assert* that two variables are equal, we use

Code:

```
assertEquals(failMessage, expected, actual)
```

where `failMessage` is the message to display should the assertion fail, `expected` is the expected value, and `actual` is the actual value. The `assertEquals` method will fail if and only if `expected` and `actual` are not equal. Now that we have a way to test code, lets write a test method. We want to test that if $a < b$, that `lessEqual(a,b)` will return `false`. To do so, we write the following method.

Code:

```
@Test
public void lessEqualsTestLessThan() {
    int a            = 1;
    int b            = 9001;
    boolean expected = true;
    boolean actual   = Comparisons.lessEqual(a,b);
    assertEquals("The method returned that 1 > 9001", expected, actual);
}
```

After this test, we need to write two more tests to complete our boundary testing of

`lessEqual`, namely a test for when $a == b$ and $a > b$. In addition to `assertEquals`, we have a few more assertions we can make if we so choose.

- `assertTrue(failMessage, b)`

  This test fails and displays `failMessage` if and only if `b` is `false`.

- `assertFalse(failMessage, b)`

  This test fails and displays `failMessage` if and only if `b` is `true`.

- `assertArrayEquals(failMessage, arr1, arr2)`

  This test fails and displays `failMessage` if and only if `arr1` and `arr2` are not the same (that is if and only if `arr1` and `arr2` do not have the same length or do not agree at every index).