Lab 6: Conditionals, Loops, and More!

Let's get started!

Today we'd like you to:

1. **Open Eclipse**. Remember to keep all your projects in the same workspace. Think of a workspace as a shelf, and each project is a binder of stuff on the shelf. If you need some help organizing your workspace, we'd be happy to help!
2. **Create a new Java project**. Call it Comp1510Lab06*LastNameFirstInitial* (non-modular)
3. **Complete the following tasks**. Remember you can right-click the src file in your lab 6 project to quickly create a new package and Java class.
4. When you have completed the exercises, **show them to your lab instructor**. Be prepared to answer some questions about your code and the choices you made.

What will you DO in this lab?

In this lab, you will:

1. Evaluate Boolean expressions
2. Make choices using the if and if-else statements
3. Write code that loops using the while statement
4. Review: write more code to pass JUnit tests

Table of Contents

1. Factorials

The factorial of n (written n!) is the product of the integers between 1 and n. Thus 4! = 1*2*3*4 = 24. By definition, 0! = 1. Factorial is not defined for negative numbers.

1. Write a program in a new class called **Factorial** inside package ca.bcit.comp1510.lab06 that asks the user for a non-negative integer and computes and prints the factorial of that integer.
2. You'll need a while loop to do most of the work—this is a lot like computing a sum, but it's a product instead. And you'll need to think about what should happen if the user enters 0.
3. Now modify your program so that it checks to see if the user entered a negative number. If so, the program should print a message saying that a nonnegative number is required and ask the user the enter another number.
4. The program should keep doing this until the user enters a nonnegative number, after which it should compute the factorial of that number. Hint: you will need another while loop before the loop that computes the factorial. You should not need to change any of the code that computes the factorial!
5. Now modify your program so it will also validate that the value entered is an integer as well as being non-negative. That is, if the user enters any string that cannot be understood as an integer, your program will give a message saying that a number is required (as well as the previous validations).
6. Check that your program will work with the following input (as well as other combinations of invalid input:
    a. AAA
    b. -2
    c. BBB
    d. -2
    e. 4
7. It is easy to get tied in knots with this twofold validation (input must be number and also nonnegative). You should wrestle with the problem and come up with a good solution.
    a. Hint: you can write a support method `int readInt(Scanner scan)` and use it in a validation loop to check for a nonnegative int. `readInt` can contain a validation loop to make sure an integer is entered using `scan.hasNextInt()`.
    b. The while loop is somewhat annoying for factorial here. We will later see for loops which or more appropriate
    c. If you remember your math, there is a way of writing factorial with no loops at all.

2. Baseball Statistics

The local Kids' League coach keeps some of the baseball team statistics in a text file organized as follows: each line of the file contains the name of the player followed by a list of symbols

indicating what happened on each at-bat for the player. The letter h indicates a hit, o an out, w a walk, and s a sacrifice fly. Each item on the line is separated by a comma. There are no blank spaces except in the player name. So, for example the file could look as follows:

```
Sam Slugger,h,h,o,s,w,w,h,w,o,o,o,h,s
Jill Jenks,o,o,s,h,h,o,o
Will Jones,o,o,w,h,o,o,o,o,w,o,o
```

The file BaseballStats.java contains the skeleton of a program that reads and processes a file in this format. Study the program and note that three Scanner objects are declared.

- One scanner (scan) is used to read in a file name from standard input.
- The file name is then used to create a scanner (fileScan) to operate on that file.
- A third scanner (lineScan) will be used to parse each line in the file.

Also note that the main method throws an FileNotFoundException. This is needed in case there is a problem opening the file.

Complete the program as follows:

1. First add a while loop that reads each line in the file and prints out each part (name, then each at-bat, without the commas) in a way similar to the URLDissector program in Listing 5.10 of the text. In particular inside the loop you need to
    a. read the next line from the file
    b. create a comma delimited scanner (lineScan) to parse the line
    c. read and print the name of the player, and finally,
    d. have a loop that prints each at bat code.
2. Compile and run the program to be sure it works.
3. Now modify the inner loop that parses a line in the file so that instead of printing each part it counts (separately) the number of hits, outs, walks, and sacrifices. Each of these summary statistics, as well as the batting average, should be printed for each player. Recall that the batting average is the number of hits divided by the total number of hits and outs.
4. Test the program on the files *stats.dat* and *stats2.dat* .

3. Enhancing the Name class

Now that we can use loops and conditionals, we can make some changes to our Name class that will enhance its utility.

1. Copy the Name class in Eclipse from your lab 5 project to your lab 6 project.
2. **Modify the setters**. If the proposed new value (the parameter) is an empty String or if it is composed of white space only, do not assign the value to the instance variable. Do nothing (ignore the value). If the String parameter is not empty and it is not white space, assign the parameter to a local variable and format the String so that the first

letter is capitalized, and the rest of the letters are lower case before assigning the new String to the instance variable.

3. **Modify the constructor** too. Disallow empty Strings or Strings composed of white space. Make sure all Strings are stored in name format, i.e., first letter capitalized and the rest in lower case. If any parameter is empty or white space, assign a suitable default value to the corresponding instance variable instead. For example, if someone tries to create a Name using (“ “, “Margaret”, “”), the first name and the last name should be assigned default values like JANE and DOE.
4. Modify the **getCharacter** method so that if the user passes an integer that exceeds the length of the name, the character @ is returned instead.
5. Write a class called **NameDriver** to prove to your lab instructor that your methods work. You can do this by creating some Names and printing their toStrings. Create correct names and names that use white space or empty Strings for their components. Mutate them, too.

4. More Mathematics

Let’s revisit the math class and satisfy some more unit tests (refer to Lab 5 for details not repeated here):

1. Create a new class called **Mathematics2** inside the package ca.bcit.comp1510.lab06:
    a. Do not include a **main method** inside the class definition. Remember the main method is what gets executed first when we start our program. In this case, the Mathematics class is not a complete program.
    b. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
        i. The name of the class and a (very) short description
        ii. An @author tag followed by your name
        iii. An @version tag followed by the version number.
2. There is a file called **Mathematics2Test.java on D2L**. Download it and copy it to the same package. Don’t edit this file. Fix the issues as in lab 5.
3. You have created a method stubs in the Mathematics class and eliminated all the compiler errors. **Save all your changes**.
4. Here comes the punch line. This won’t work until all the compiler errors have been resolved by making method stubs in the Mathematics class. Hold on to your hats. Run the test file. Right click on it and choose **Run As > JUnit Test**.
5. The JUnit window automatically opened and revealed the results of the JUnit tests. Each test method is run independently. Each method contains a simple test that compares the output or return value from a method with an expected value. If a method returns a wrong value, there is a X beside the test. If any tests fail, the coloured strip is red.
6. Your task this lab is to implement the methods in the Mathematics class so the JUnit tests in the MathematicsTest class pass. You can click on any of the failed tests, or you can view the code in the MathematicsTest file to see how the methods are being tested.
7. When your tests all pass, the coloured strip will be green.
8. Here are some helpful Javadoc comments for you to use for each element you must complete. There are hints in the comments about how your methods should be implemented:

```
/**
 * Returns the area of the square with the specified edge length.
 *
 * @param edgeLength
 *        of the square.
 * @return area as a double
 */
public double getSquareArea(double edgeLength) {
/**
 * Returns the sum of the specified values.
 *
 * @param first
 *        operand
 * @param second
 *        operand
 * @return sum of the operands
 */
public double add(double first, double second) {
/**
 * Returns the product of the specified values.
 *
 * @param first
 *        operand
 * @param second
 *        operand
 * @return product of the operands
 */
public double multiply(double first, double second) {
/**
 * Returns the difference of the specified values.
 *
 * @param first
 *        operand
 * @param second
 *        operand
 * @return difference of the operands
 */
public double subtract(double first, double second) {
/**
 * Returns the quotient of the specified values. If the divisor is zero,
 * returns zero instead of NaN or infinity.
 *
 * @param first
 *        operand
 * @param second
 *        operand
 * @return quotient of the operands
 */
public double divide(double first, double second) {
/**
 * Returns the absolute value of the specified integer.
 *
 * @param number
```

```
     *          to test
     * @return absolute value of number
     */
    public int absoluteValue(int number) {
    /**
     * Converts the specified number of feet to kilometres.
     * @param feet to convert
     * @return kilometres in the specified number of feet
     */
    public double convertFeetToKilometres(double feet) {
    /**
     * Returns the sum of the numbers between zero and the
     * first parameter that are divisible by the second
     * number.  For example, sumOfProducts(10, 3) will return
     * 3 + 6 + 9 = 18, and sumOfProducts(10, 2) will return
     * 2 + 4 + 6 + 8 + 10 = 30 and sumOfProducts(-10, 2) will
     * return -2 + -4 + -6 + -8 + -10 = -30.
     * @param bound the upper bound
     * @param divisor the divisor
     * @return sum
     */
    public int sumOfProducts(int bound, int divisor) {
    /**
     * Returns a random number between 10 and 20 inclusive,
     * but NOT 15.
     * @return random number in range [10, 20] excluding 15.
     */
    public int getRandomNumberBetweenTenAndTwentyButNotFifteen() {
```

5. You're done! Submit your work.

If your instructor wants you to submit your work in the learning hub, export it into a Zip file in the following manner:

1. Right click the project in the Package Explorer window and select export…
2. In the export window that opens, under the General Folder, select Archive File and click Next
3. In the next window, your project should be selected. If not click it.
4. Click *Browse* after the "to archive file" box, enter in the name of a zip file (the same as your project name above with a zip extension, such as Comp1510Lab06BloggsF.zip if your name is Fred Bloggs) and select a folder to save it. Save should take you back to the archive file wizard with the full path to the save file filled in. Then click Finish to actually save it.
5. Submit the resulting export file as the instructor tells you.