# CSC-240 Lesson 3: Control Structures

## Assignment 3

This assignment is loosely based up the exercises described in Chapter 4, Exercise 4.38 [4.37 in Edition 10] (Factorial, parts a and c), and Chapter 5, Exercise 5.17 (Calculating Sales). All the relevant requirements for this assignment are described below.

You should submit three Java programs, with names as given below, together with any additional files for classes that you may wish to have to complete your solutions. Each of the listed classes should contain a *main* method.

- `Factorial.java`—similar to the exercise described in Exercise 4.37a. See details of the full requirements below (12 points).
- `Ex.java`—similar to the exercise described in Exercise 4.37c. See details of the full requirements below (18 points).
- `CalculatingSales.java`—similar to the exercise described in Exercise 5.17, See details of the full requirements below (20 points).

No additional checking of the format of the input made by a user of these programs is needed beyond the checking that may be done by the **Scanner** class. If the input is not a valid representation of the appropriate value, the **Scanner** class will throw an **Exception**—learning to deal with the throwing of an **Exception** is a topic for a later Lesson.

> Each of these exercises require computation using Java primitive values (*i.e.*, doing what computers supposedly do best). However, getting a computer, *via* a programming language, to compute correctly takes care and attention.

### *Factorial.java*

This is an exercise in using repetition structures to calculate the mathematical values of the factorial function. Since methods have not yet been covered in detail, you may do the calculations in a class that contains just a *main* method. The looping structures require the use of count-controlled loops.

- This program computes the factorial function of a non-negative integer, which is written mathematically as *n!* (the parameter *n* followed by an exclamation mark).

  For values of *n* equal to or greater than 1, *n!* is defined as:

  $$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1,$$

  and for the value $n == 0$, is defined as:

  $$0! = 1.$$

  The value computed must be a Java integral value (either **int** or **long**). The program should ask the user for a value, $n \geq 0$. If the value entered is less than zero, a message must be given the user, and the user allowed to try again. If the value is 0, the program should output the value of 0! and exit. If the value is greater than zero, the factorial value of that value *n* should be printed (if possible).

  The program must then ***continue to loop*** and to ask the user for another number for which the factorial may be computed (after each result has been output), until a zero value is entered.

  If the user enters a number for which the factorial cannot be computed or cannot be computed correctly, tell the user that is the problem, and keep asking for another number until an acceptable number (or 0) is entered.

### *Ex.java*

This is an exercise in using repetition structures to calculate the mathematical values of sum of a series. Since methods have not yet been covered in detail, you may do the calculations in a class that contains just a *main* method. The computation itself requires looping structures that use of count-controlled loops.

- This program computes an *estimate* of the value of the mathematical value $e^x$, which is defined, for a given exponent $x \geq 0$, and for $n > 0$ terms of a series, as:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^{n-1}}{(n-1)!}$$

  As *n* is made greater, the estimate of *e* becomes more precise. If the computation is done with **double** computations, then after some number of additional terms (some value of *n*), the precision of the **double** representation will be reached, and no further improvement will be seen. However, for large values of *x*, it may not be possible to obtain an estimate.

  The value computed should be a Java floating-point value (please use **double**). The program should first ask the user for an integer value $n \geq 0$, which indicates the number of terms to be evaluated. If the value entered is less than zero, a message must be given the user, and the user allowed to try again. If the value is 0, the program should just exit.

  If the number of terms value is greater than 0, then the user should be asked to enter a value for the exponent *x*. The exponent may be any positive **double** value, zero or fractional. If a negative value of *x* is entered, the program should allow the user to start over by requesting the number of terms again. (The series works for negative values, but requires some additional precautions that will not be explored here).

  As before, make the program ***continue to loop*** until the user asks for a zero number of terms to be used.

  Some values of the exponent will prevent the program getting correct results. If the correct value cannot be computed, tell the user that is the problem, and keep asking for another number of terms and an exponent value until an acceptable number (or 0) is entered. (Note that if the exponent, *x == 1*, then the value obtained should be for just the value of *e* itself. It should always be possible to obtain an estimate for this value, for any number of terms.)

  Print out your estimates of the value of *e* raised to the given power to at least 15 significant places.

  ***Do not use the* Math.*pow* *or* Math.*exp* *methods for this exercise (other than for debugging or comparison purposes); the intent is to evaluate the given series in a manner similar to what the* Math.*exp* *method itself would have to use internally to compute the same values.***

- ***Example of Input and Output for the program Ex.java***

  Here are some examples of input and output for the program `Ex.java`. The number of terms shown in these examples are close to the minimum needed to achieve **double** type accuracy for the given exponents—you should attempt to replicate these results:

```
Enter a positive number of terms (or 0 to quit): 16
Enter an exponent (positive, zero or fractional): 0.5
After 16 terms estimate of e to the power 0.500000 is 1.64872127070013

Enter a positive number of terms (or 0 to quit): 18
Enter an exponent: 1
After 18 terms estimate of e to the power 1 is 2.71828182845905

Enter a positive number of terms (or 0 to quit): 67
Enter an exponent: 20
After 67 terms estimate of e to the power 20 is 485165195.409790

Enter a positive number of terms (or 0 to quit): 190
Enter an exponent: 100
After 190 terms estimate of e to the power 100 is 2.68811714181614e+43

Enter a positive number of terms (or 0 to quit): 936
Enter an exponent: 709
After 936 terms estimate of e to the power 100 is 8.21840746155499e+307

Enter a positive number of terms (or 0 to quit): 347
```

```
Enter an exponent: 1000
That value is too large to be able to compute that exponent of e; please try again.
```

### *CalculatingSales.java*

This is an exercise in using repetition structures to calculate the meaning of values related to real objects. Since methods have not yet been covered in detail, you may do the calculations in a class that contains just a *main* method. The looping structures require the use of sentinel-controlled loops. Care should be taken that dollar-and-cents values are properly and correctly presented.

- This program is an approximate simulation of an online ordering application. (This program really does not represent how this function might be done on the WWW.) For this simulation, there are five products available. Each product has a number, a name, and a price. The products are:

| Product number | Product name | Price/unit |
|---|---|---|
| 1 | SOAP | $2.98 |
| 2 | SHAMPOO | $4.50 |
| 3 | LOTION | $9.98 |
| 4 | CONDITIONER | $4.49 |
| 5 | MOISTURIZER | $6.87 |

  The user should be asked to enter a product (by number), and a quantity.

  - If the user requests a negative quantity, the user should be informed that may not be done, and should be allowed to go back and select another product.
  - If the user asks for a product that does not exist, the user should be informed that there is no product, and allowed to enter a different request. The user should be allowed to continue ordering products until product number “0” is requested. At that time, the program will produce an invoice, and terminate.
  - If the user requests the same product again, the quantity ordered is to be added to the currently ordered quantity of that product.

  The program should use one or more **switch** statements to determine the price and actions for each product requested. (This program could be simplified using arrays, but arrays have not been studied yet, and this is an opportunity to practice with **switch** statements.)

  The invoice, for each item ordered, should show the product number, the name of the product, the price per unit, and the extended price (the unit price multiplied by the number ordered). The invoice should also show the total cost of the order (no tax calculation is added for this program) The format of the invoice produced at the end must be formatted appropriately, and may be similar to the one shown below in this example.

- ***Example of Inputs and Outputs for CalculatingSales.java***

  Here is an example of inputs to and outputs from this program:

```
Enter a product number from 1 to 5 (or 0 to stop): 8
Enter quantity ordered: 6
No product available with that Id. Please continue.
Enter a product number from 1 to 5 (or 0 to stop): 7
Enter quantity ordered: -5
Quantity may not be negative. Please continue.
Enter a product number from 1 to 5 (or 0 to stop): 4
Enter quantity ordered: 15
Enter a product number from 1 to 5 (or 0 to stop): 2
Enter quantity ordered: 10
Enter a product number from 1 to 5 (or 0 to stop): 4
Enter quantity ordered: 2
Enter a product number from 1 to 5 (or 0 to stop): 1
Enter quantity ordered: 1
```

```
Enter a product number from 1 to 5 (or 0 to stop): 1
Enter quantity ordered: 2
Enter a product number from 1 to 5 (or 0 to stop): 0

    |    Item     |Quantity|  Price  |  Extension  |
-------------------------------------------------|
  1 |        SOAP |      3 |   2.98  |       8.94  |
  2 |     SHAMPOO |     10 |   4.50  |      45.00  |
  4 | CONDITIONER |     17 |   4.49  |      76.33  |
-------------------------------------------------|
                     Total for Order |     130.27  |
```

Note that in the above, a product number which does not correspond to an actual product is reported as “No product available with that Id” *via* a message to the user. If a negative quantity is requested, the user is also informed that this cannot be done.

***The final invoice should* not *show items that were* not *ordered (think of a real online ordering situation in which there may be 1,000,000 possible products).***

> The following *About the Assignment* topic contains further explanation, directions, and hints about this assignment. Please read that topic before starting this assignment.

---

### Important

Please follow the directions of Programming Assignment Identification in submitting your programming solutions.

If you have any questions or concerns about the Assignment or the *About the Assignment* topic, please use the Lesson Question discussion topic, or send a message by the D2L Internal Messaging system if you prefer.

**Note**: you may submit to the Assignment Submission Folders as many times and as often as you wish, up to the deadline time. Each submission is tagged with the date/time, and so each submission remains separate and distinct. Unless you leave instructions to the contrary, only the most recent of each file with the same name will be viewed for the purposes of grading. Details may be found in the topic *How To Submit and Get Feedback on Assignments* in the *How To* module.

Messages that accompany Assignment Submissions are read, and responded to, ***only*** when assignment submissions are graded (which is after the Assignment Submission Folder closing date/time). If you have a comment or question about an assignment, or a request for assistance, that needs an earlier response, then that comment, question or request should be made or asked *via* an Internal Message or the Discussion board, as these are usually read and answered every day.