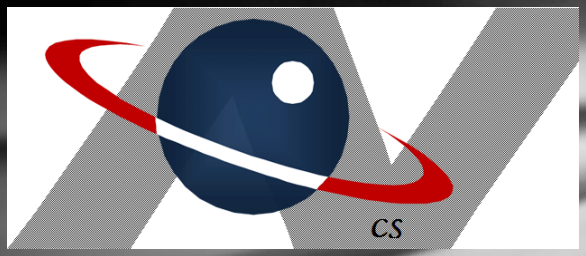
O(N) CS LESSONS

Lesson 6B – Loops 2

Counting and Accumulating



By John B. Owen

Table of Contents

Objectives

- In the previous lesson, you studied the three basic styles of loops and wrote methods for 15 classic loop processing techniques that involved different output patterns.
- In this lesson you will learn ways to use loops in problem solving that require repetitive actions, such as accumulating sums and products, and counting.

Loops that calculate

- A loop can be set up to **count** the occurrence of certain events, such as all of the multiples of 5 between two input values.
- A loop can also **sum,** or accumulate all of the values between two numbers, or even **multiply** all of them.

Loops that calculate

- Loops can also be used to detect if numbers are prime, accumulate compounded interest in a bank account…the possibilities are endless.

Variables for calculating loops

- These special looping techniques need extra variables to accumulate the count, sum, or product.
- In addition to correctly setting up your loop control variable, it is very important to properly initialize these accumulating variables (give them appropriate beginning values).

Variables for calculating loops

- If a variable is used to **count** or **sum**, you must set it to a beginning value of zero.
- If a variable is used to accumulate the **product** of some values, what beginning value must it have to work properly?

Variables for calculating loops

- If you thought of the value 1, you are right!
- Why can't it be set to zero like the counter variable or summing variable?
- Because zero multiplied by anything remains zero!

Initializing variables for calculating loops

- Here are some typical examples of initializing variables used in counting, summing, and multiplying:

```
int count = 0;
int sum = 0;
int prod = 1;
```

Return methods

- Since these methods involve a calculation, we will revert back to defining return methods, ones that do a calculation and return an answer.
- Although any loop will do for these methods, all examples shown here use a while loop.

Counting loop – example 1

MyLoops2.java | MyLoops2_1.in

```
68      /**This method calculates and returns the count of
69       *the number of values between
70       *two input parameters.
71       */
72      public static int countMtoN(int M, int N)
73      {
74        int x=M;
75        int count = 0;
76        while(x<=N)
77        {
78            count++;
79            x++;
80        }
81        return count;
82      }
83      public static void main (String [] args)
84        throws IOException
85      {
86        Scanner f = new Scanner(new File("MyLoops
87        while (f.hasNext())
88        {
89            int m = f.nextInt();
90            int n = f.nextInt();
91            out.printf("There are %d values between %d and %d.\n",
92                    countMtoN(m,n),m,n);
93        }
94      }
95  }
```

- Here is a loop that counts the number of values between two numbers.
- Notice that the method loop structure is the same as if you were outputting the values.
- Also notice how the output statement in main calls this method.

MyLoops2.java | MyLoops2.in

```
1 5
2 17
3 1
4 24
5
```

```
There are 13 values between 5 and 17.
There are 24 values between 1 and 24.
Press any key to continue...
```

Counting loop – example 1

MyLoops2.java | MyLoops2_1.in

```
68      /**This method calculates and returns the count of
69       *the number of values between
70       *two input parameters.
71       */
72      public static int countMtoN(int M, int N)
73      {
74        int x=M;
75        int count = 0;
76        while(x<=N)
77        {
78            count++;
79            x++;
80        }
81        return count;
82      }
83      public static void main (String [] args)
84        throws IOException
85      {
86        Scanner f = new Scanner(new File("MyLoops2.in"));
87        while (f.hasNext())
88        {
89            int m = f.nextInt();
90            int n = f.nextInt();
91            out.printf("There are %d values between %d and %d.\n",
92                    countMtoN(m,n),m,n);
93        }
94      }
95  }
```

- Instead of an output statement as the action of the loop, a counter variable increments as each step is taken.
- After the loop is finished, the value of the counter is returned.

MyLoops2.java | MyLoops2.in

```
1 5
2 17
3 1
4 24
5
```

```
There are 13 values between 5 and 17.
There are 24 values between 1 and 24.
Press any key to continue...
```

Counting loop – example 2

```
/**This method counts and returns the number of even values between
 *two received parameters.
 */
public static int countMtoNEvens(int M, int N)
{
  int x=M;
  int count = 0;
  while(x<=N)
  {
      if(x%2==0)
          count++;
      x++;
  }
  return count;
}
public static void main (String [] args)
  throws IOException
{
  Scanner f = new Scanner(new File("MyLoops2.in"));
  while (f.hasNext())
  {
      int m = f.nextInt();
      int n = f.nextInt();
      out.printf("There are %d even values between %d and %d.\n",
              countMtoNEvens(m,n),m,n);
  }
}
```

- In this example, the same loop structure is used, but only the even numbers are counted.

MyLoops2.java | MyLoops2.in

```
5
17
1
24

```

```
There are 6 even values between 5 and 17.
There are 12 even values between 1 and 24.
Press any key to continue...
```

Accumulating loop – example 3

MyLoops2.java | MyLoops2.in

```
/**This method accumulates and returns the sum
 *of all the values between two received parameters.
 */
public static int sumMtoN(int M, int N)
{
  int x=M;
  int sum = 0;
  while(x<=N)
  {
      sum+=x;
      x++;
  }
  return sum;
}
public static void main (String [] args)
  throws IOException
{
  Scanner f = new Scanner(new File("MyLoops2.in"));
  while (f.hasNext())
  {
      int m = f.nextInt();
      int n = f.nextInt();
      out.printf("The sum of all values between %d and %d is %d.\n",
              m,n,sumMtoN(m,n));
  }
}
```

- Here is a loop that ACCUMULATES the sum of all of the values between two parameter values.

MyLoops2.java | MyLoops2.in

```
5
17
1
24
```

```
The sum of all values between 5 and 17 is 143.
The sum of all values between 1 and 24 is 300.
Press any key to continue...
```

Accumulating loop – example 4

oops2.java × MyLoops2.in

```
/**This method accumulates and returns the product of
 *all multiples of 5 between two input parameters.
 */
public static int prodMtoNmultFive(int M, int N)
{
  int x=M;
  int prod = 1;
  while(x<=N)
  {
      if(x%5==0)
          prod*=x;
      x++;
  }
  return prod;
}
public static void main (String [] args)
  throws IOException
{
  Scanner f = new Scanner(new File("MyLoops2.in"));
  while (f.hasNext())
  {
      int m = f.nextInt();
      int n = f.nextInt();
      out.printf("The product of all multiples of 5"
          +" between %d and %d is %d.\n",m,n,prodMtoNmultFive(m,n));
  }
}
```

- Here is a loop that will ACCUMULATE the PRODUCT of all of the multiples of five between two values.

MyLoops2.java | MyLoops2.in ×

```
1 5
2 17
3 1
4 24
5
```

```
The product of all multiples of 5 between 5 and 17 is 750.
The product of all multiples of 5 between 1 and 24 is 15000.
Press any key to continue...
```

Lesson Summary

- In this brief lesson, you learned some advanced techniques using loops – how to count and accumulate values.
- Now it is time to practice with several examples.

Labs

- MyLoops2 will be the name of this series of labs. As you did before, create a separate folder and file called MyLoops2 and do your work there.

Labs

- All methods will be return methods.
- The loop style choice is up to you, just be sure to use each of the three styles equally in the methods you define, so that you gain proficiency in all three styles.
- The first method is done for you to help you get started.

Method 1 – *sigmaN*

MyLoops2.java | MyLoops2_1.in

```
13      /**Method 1 - This method calculates the
14       *summation of all the values between
15       *1 and an input parameter.
16       */
17      public static int sigmaN(int N)
18      {
19        int i=1;
20        int sum = 0;
21        while(i<=N)
22        {
23            sum+=i;
24            i++;
25        }
26        return sum;
27      }
28    /**main method
29     */
30      public static void main (String [] args) throws IOException
31      {
32        Scanner f = new Scanner(new File("MyLoops2_1.in"));
33        while (f.hasNext())
34        {
35            int x = f.nextInt();
36            out.printf("The sum of all integers from 1 to %d is %d.\n",
37                x,sigmaN(x));
38        }
39      }
```

```
The sum of all integers from 1 to 5 is 15.
The sum of all integers from 1 to 17 is 153.
The sum of all integers from 1 to 10 is 55.
The sum of all integers from 1 to 24 is 300.
Press any key to continue...
```

MyLoops2.java | MyLoops2_1.in

```
1  5
2  17
3  10
4  24
5
```

- WAM that will perform the classic SIGMA summation process, as shown here.

$$\sum_{i=1}^{n} i$$

- This symbol means to sum all the values of i from 1 to n.
- Here is a solution.

Method 2 – *prodN*

- WAM that receives an integer N and performs the PRODUCT accumulation process, as symbolized below.

$$\prod_{i=1}^{n} i$$

MyLoops2.java | MyLoops2_2.in

```
5
8
10
4
```

C:\Program Files (x86)\Xinox Software\JCreatorV5LE\GE2001.exe

```
The product of all integers from 1 to 5 is 120.
The product of all integers from 1 to 8 is 40320.
The product of all integers from 1 to 10 is 3628800.
The product of all integers from 1 to 4 is 24.
Press any key to continue...
```

Method 3 – *countMultZfromXtoY*

- WAM that receives three integers X, Y and Z and returns the count of all multiples of Z between X and Y. Below is a glimpse of the method header and method call. Notice the precondition, which guarantees a certain situation prior to the method call.

MyLoops2.java | MyLoops2_3.in

```
1 5 50 2
2 8 100 7
3 10 1000 12
4 4 9 1
5
```

C:\Program Files (x86)\Xinox Software\JCreatorV5LE\GE2001.exe

```
There are 23 multiples of 2 between 5 and 50.
There are 13 multiples of 7 between 8 and 100.
There are 83 multiples of 12 between 10 and 1000.
There are 6 multiples of 1 between 4 and 9.
Press any key to continue...
```

```
/**This method receives three parameters X, Y, and Z,
 *calculates and returns the count of the multiples of Z
 *between X and Y.
 *Precondition:  X < Y, Z >= X and Z <= Y.
  */
 public static int countMultZfromXtoY(int X, int Y, int Z)
```

```
out.printf("There are %d multiples of %d between %d and %d.\n",
    countMultZfromXtoY(x,y,z),z,x,y);
```

Method 4 – *populationGrowth*

- In a biology experiment, Cory finds that a sample of microorganisms doubles in population every 12 hours. He wants to know how many hours it will take to reach a certain microorganism population level if he starts with 1000 organisms each time.
- WAM to receive an integer N representing a desired organism population level, and determine how many hours it will be until the population level reaches N or more organisms.

MyLoops2.java | MyLoops2_4.in

```
1 10000
2 100000
3 1000000
```

```
It took 48 hours for 1000 organisms
to reach or exceed a population of 10,000.
It took 84 hours for 1000 organisms
to reach or exceed a population of 100,000.
It took 120 hours for 1000 organisms
to reach or exceed a population of 1,000,000.

Press any key to continue...
```

Method 5 – *manhattanCostX*

- In 1626, the Dutch settlers purchased Manhattan Island from the Native American Lenape Indians. According to legend, the purchase price was 60 guilders, or about $24. Suppose that the Indians had invested this amount at 3% annual interest, compounded annually. If the money had earned interest from the start of 1626 to the end of any particular year, how much money would the Indians have in the bank, say by the end of 2011?
- WAM to calculate and return the current value of the original investment up to year X (integer parameter).
- Currency output format is required, with commas for values equal to or exceeding $1,000.

MyLoops2.java **MyLoops2_5.in**

```
1626
1627
1726
1826
1926
2014
```

```
The value of an original $24.00 investment in 1626
compounded annually at 3% has a value of $24.72 in 1626.
The value of an original $24.00 investment in 1626
compounded annually at 3% has a value of $25.46 in 1627.
The value of an original $24.00 investment in 1626
compounded annually at 3% has a value of $475.08 in 1726.
The value of an original $24.00 investment in 1626
compounded annually at 3% has a value of $9,130.48 in 1826.
The value of an original $24.00 investment in 1626
compounded annually at 3% has a value of $175,475.25 in 1926.
The value of an original $24.00 investment in 1626
compounded annually at 3% has a value of $2,365,329.52 in 2014.
```

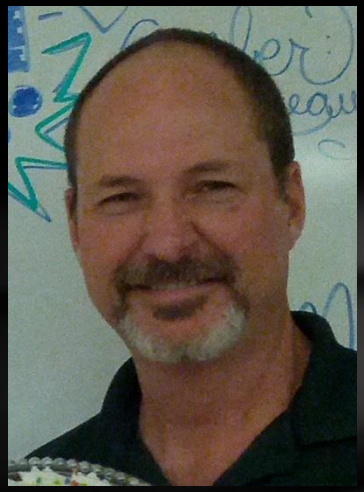
JavaDoc

- Complete the documentation for all of the methods, run the JavaDoc utility, and make any adjustments as necessary for completeness and clarity.

CONGRATULATIONS!

- You now know how to use loops to do calculations requiring repetitive actions.
- *Lesson 6C will explore using loops for a wider variety of file input patterns and more complex iterative calculations.*

Thank you, and have fun!



To order supplementary materials for all the lessons in this package, such as lab solutions, quizzes, tests, and unit reviews, visit the O(N)CS Lessons website, or contact me at

John B. Owen
captainjbo@gmail.com