Java – Loops

9.01 Number.java

Write a utility class called Number and implement the static methods below. No Strings of any type allowed (String, StringBuffer, StringBuilder, etc.). Any use of a String type will result in a failing grade. The lab uses the **for**-loop control structure.

static long **factorial**(int n) – returns the factorial of *n*. You may assume the factorial fits in a long and that n >= 0. Five factorial (5!) is equal to 5 * 4 * 3 * 2 * 1 (! is a mathematical symbol; it is the logical negation in Java so don't use it in this method).

static int **GCD**(int num1, int num2) – returns the greatest common divisor of two positive numbers. The GCD of two numbers is the largest whole number that divides both evenly.

(Algorithmic help if you're stuck: start at the smaller of the 2 numbers and check to see if it divides both numbers. If yes, then you're done, otherwise try one less and repeat until you reach 1.) For a challenge, learn the Euclidean algorithm and implement it recursively.

static int **sumEvens**(int start, int stop) – returns the sum of all even numbers from start to stop inclusive. sumlEvens(3, 8) -> returns 18 because 4 + 6 + 8 == 18

static int **sumSquares**(int n) – returns the sum of the squares from 1 to *n* $(1^2 + 2^2 + 3^2 \ldots n^2)$. You may assume n will be greater than 1. sumSquares(4) -> returns 30 as 1 + 4 + 9 + 16 == 30.

static int **sumFactors**(int n) – returns the sum of all factors of a positive number *n*. sumFactors(8) -> returns 15 as the factors of 8 are 1, 2, 4, and 8. Loop from 1 to *n* and if the loop counter divides *n* evenly, add it to an accumulator.

static boolean **isPrime**(int n) – returns true if *n* is prime and false otherwise. A prime number is only divisible by itself and 1. The first few prime numbers are 2, 3, 5, 7, 11, and 13. Apart from the number 2, all even numbers are not prime. Check this first. Write a loop to check if n is divisible by odd numbers starting at 3 and stopping at the square root of n (no point in going any further). For those interested, loop up an ancient algorithm for finding a prime number called Sieve of Eratosthenes.

static double **calculatePI** (int n) – returns an approximation of π by *n* terms from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

boolean **isPerfect**() – returns true if the number is perfect and false otherwise. A perfect number is equal to the sum of its divisors (excluding self). 6 is perfect as 1 + 2 + 3 == 6. You may invoke any method from this class to reduce redundancy.

NUMBER SAMPLE DATA

```
// Sample Data

public static void main(String[] args) {
    System.out.println(Number.factorial(5) + " == 120");
    System.out.println(Number.GCD(100, 24) + " == 4");
    System.out.println(Number.GCD(18, 24) + " == 6");
    System.out.println(Number.sumEvens(1, 10) + " == 30");
    System.out.println(Number.sumSquares(10) + " == 385");
    System.out.println(Number.sumFactors(8) + " == 15");
    System.out.println(Number.isPerfect(6) + " == true");
}
```

NUMBER SAMPLE OUTPUT

```
120 == 120
4 == 4
6 == 6
30 == 30
385 == 385
15 == 15
true == true
```

9.02 Integer.java

Write an immutable Integer class that wraps the int primitive data type into an object. No Strings of any type allowed (String, StringBuffer, StringBuilder, etc.) or using Java's Integer class. Any use of these will result in a failing grade. This lab uses the **while** loop control structure. Note this class is using OOP and the class is immutable which means none of the methods modify the private instance variable.

Integer () - constructs and initializes the integer 7.

Integer(int num) - initializes the object with num value.

int getN() – accessor method that returns the integer.

int **digitCount**() – returns the number of digits.

    Integer value = new Integer(99);

value.digitCount(); // -> returns 2 as 99 has two digits

int **digitSum**() – returns the sum of the digits as a positive number.

Integer value = new Integer(911);

value.digitSum(); // -> returns 11 as 9 + 1 + 1 == 11

double **digitAverage**() – returns the average of the sum of the digits. No loop required as you can call two methods from above for this one. Always be cognizant of integer division when dividing.

int **digitDivisorSum**() – returns the sum of the digit divisors. (new Integer(145)).divisorSum() returns 6 as 1 + 5 == 6. Five divides 145 and one divides 145. Four does not.

int **reverse**() – returns the digits in reverse order. Given any number, you can reverse the numbers by multiplying each digit by a power of 10 and adding them together. The first or rightmost digit needs to be multiplied by 10^(total digits – 1). Precondition: n > 0.

long **toBase**(int base) – returns the number converted to the given base(binary to base 9). Converting from base 10 to any base follows the same process. Note the return type of this method so watch out for overflow. Precondition: n >= 0 (i.e., you don't have to check for it).

Convert 236 from base 10 to base five:

Keep dividing by 5, until your quotient is zero.

```
236 ÷ 5 == 47 r 1

 47 ÷ 5 ==  9 r 2

  9 ÷ 5 ==  1 r 4

  1 ÷ 5 ==  0 r 1
```

Remember the remainders go backwards so 236 base 10 is 1421 base 5

int **populationCount**() – returns the number of one bits in the binary representation of the number. You may assume the binary representation will fit into a long and the number is positive.

Integer value = new Integer(35);

value.populationCount(); // -> returns 3 as 35 in binary (100011) has 3 ones.

boolean **isOdious**() – returns true if odious and false otherwise. An odious number is a non-negative number that has an odd number of 1's in its binary expansion. Precondition: number >= 0 (i.e., you don't have to check this). No loop required as you can invoke a method from this class.

boolean **isEvil**() – returns true if evil and false otherwise. An evil number is a non-negative number that has an even number of 1's in its binary expansion. Precondition: number >= 0. Invoke *isOdious* and return the negation.

INTEGER SAMPLE DATA

```
// Sample Data

public static void main(String[] args) {
    Integer value = new Integer(1252);
    System.out.println(value.digitCount() + " == 4");
    System.out.println(value.digitSum() + " == 10");
    System.out.println(value.digitAverage() + " == 2.5");
    System.out.println(value.digitDivisorSum() + " == 5");
    System.out.println(value.reverse() + " == 2521");
    System.out.println(value.toBase(4) + " == 103210");
    System.out.println(value.toBase(2) + " == 10011100100");
    System.out.println(value.populationCount() + " == 5");
    System.out.println(value.isOdious() + " == true");
    System.out.println(value.isEvil() + " == false");
}
```

INTEGER SAMPLE OUTPUT

```
4 == 4
10 == 10
2.5 == 2.5
5 == 5
2521 == 2521
103210 == 103210
10011100100 == 10011100100
5 == 5
true == true
false == false
```

9.03 Strings.java

Implement all the static methods listed below.

static String **reverse**(String str) – returns *str* in reverse order.

static int **numVowels**(String str) – returns the number of vowels (a, e, i, o, u) in *str*.

static String **getUpper**(String str) – returns all the uppercase letters concatenated together.

static String **everyNth**(String str, int n) – returns every *n*th letter starting at index zero.

static String **box**(String str) – returns a box representation of *str* (check output below).

static String **box2**(String str) – returns an alternative box representation of *str* (check output below).

static String **staircase**(String str) – returns a staircase representation of *str* (check output below).

static String **triangleA**(int n) – returns a triangle of asterisk of size *n* (check output below).

static String **triangleB**(int n) – returns a triangle of asterisk of size *n* (check output below).

static String **triangleC**(int n) – returns a triangle of asterisk of size *n* (check output below).

static String **triangleD**(int n) – returns a triangle of asterisk of size *n* (check output below).

static String **multiplicationTable**(int n) – returns a multiplication table as a String (check output below). The numbers need to be right justified and spaced appropriately. Notice one space between the columns with the largest value.

    String.format("%3d", 2) -> returns " 2"; notice the two leading spaces

static String **makeHappy**(String str) -> returns a String where all characters have been replaced with dashes(-) except all unhappy faces :(are replaced with happy one's :).

static int **countCaps**(String str) -> returns a count of all words in *str* that start with a capital letter. For simplicity, a word will be anything that is preceded by a space.

    countCaps("Nothing iS ImpOsSible") – returns 2

static boolean **isBalanced**(String str) -> returns true if str contains a balanced number of parenthesis.

    isBalanced("(4*(3+3))") – returns true

STRINGS SAMPLE DATA

```
// Sample Data

public static void main(String[] args) {
    out.println(Strings.reverse("hello") + " == olleh");
    out.println(Strings.numVowels("eggOmelet") + " == 4");
    out.println(Strings.getUpper("bAdeB fC") + " == ABC");
    out.println(Strings.everyNth("0123456789", 3));
    out.println("\n" + Strings.box("Box") + "\n");
    out.println(Strings.box2("CompSci")+ "\n");
    out.println(Strings.staircase("Comp")+ "\n");
    out.println(Strings.triangleA(5));
    out.println(Strings.triangleB(5));
    out.println(Strings.triangleC(5));
    out.println(Strings.triangleD(5));
    out.println(Strings.multiplicationTable(10)+ "\n");
    out.println(Strings.makeHappy(":(sad :(faces:("));
    out.println(Strings.isBalanced("(1+3 * (3)+4) / (3+7)"));
}
```

STRINGS SAMPLE OUTPUT

```
olleh == olleh
4 == 4
ABC == ABC
0369

Box
Box
Box

CompSci
o     c
m     S
p     p
S     m
c     o
icSpmoC
```

STRINGS SAMPLE OUTPUT

```
C
Co
Com
Comp


*
**
***
****
*****

*****
****
***
**
*

    *
   **
  ***
 ****
*****

*****
 ****
  ***
   **
    *


   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100


:)----:)-----:)
true
```