Computer Science 1-2 2019 Lab 6

A quick note on libraries in Java: we have been using one library quite extensively, that is,

```
java.io.*
```

The syntax above says that there is an io library and it may have several sub-libraries. We want them all, so that is why there is a ".*". In computer science, the "*" is called a "regular expression". Way beyond the scope of this course, but it is a placeholder meaning "give me all the libraries that start with the string "java.io.". You have also used the

```
java.util.Scanner
```

library. Note: we could have asked for Java.util.* and gotten the Scanner library, but we would have gotten a whole lot of other libraries too which would just slow down our compilation.

Now, in the next program we need to be able to generate "random" numbers. This is a big deal in computer science and in almost all aspects of the real world. Being able to have random numbers enables modeling of real world things. All of statistics (and therefore all research) depends on random numbers. True random numbers are a graduate course in itself. If you are interested in the subject, Donald Knuth's book: "The Art of Programming: Volume 2 Seminumerical Algorithms" is still considered the finest treatment of the subject. In fact, there are actually companies that provide tables of random numbers that may be used in research with guarantees about the randomness. Because, guess what … it is impossible to write a program to generate truly random numbers. Bizarre.

For our purposes, the following code illustrates how to get a random number in Java.(I really mean a "pseudo-random" number given my assertion above but don't want to type "pseudo" every few lines.) Notice that the random number I get is an integer. Go research the Java random number class by googling for the universe of ways you can have random numbers in Java. This code also introduces a new **java statement.** The statement is a for-loop. It is a lot like a while loop but is used for different conditions.

The following two loops **are equivalent in behavior**: one is a while-loop; one is a for-loop.

```
int i=0;

while (i <=10) {
  System.out.println(i);
  i++;
}
```

```
for (i=0; i<=10;i++) {

System.out.println(i);

}
```

The two loops are *equivalent*. The second loop has three controls:

1) The initial value of the loop-control variable (i = 0;)
2) The condition to check entering the loop each time (i<=10;)
3) What to do at the end of the loop before going to the top again (i++);

So, why have two ways? Programmers tend to use while-loops when looping on conditions: e.g. keep looping until the user gives a -1 input. Programmers tend to use for-loops when looping a specific number of times. e.g. Loop exactly 10 times.

Going forward we use for-loops for a very specific kind of data structure (something beyond ints, floats, doubles, Booleans etc.!).

Now, here is code that works with random numbers and for-loops! Understand this thoroughly before trying the problems.

```
import java.io.*;
import java.util.Random;

class num {

 public static void main(String args[]) {

    int valuereturned; //This is the number we get back from the
                       //nextInt call to the Random class
    int i;

    // The next statement creates an instance of the object
    // Random;

    Random myrand = new Random();

    for (i=0; i<10; i++) {
       //The call to nextInt fetches a brand new random number
       //with a value between 0 (inclusive) and 2000 (exclusive)
       valuereturned = myrand.nextInt(2000);
       System.out.print(firstnum + " ");
    }
```

```
}

}
```

In reading the code, note that a Random is an **object** (like a String!) We get one by asking for one with the "new" operator. Once we have one (myrand above) we can keep asking for that **object** to give us another random number (int, double, etc.). So, Random itself is an object and it has methods which will give us a random number back. We are getting close to what object oriented programming is all about.

A typical output of the above program may look like:

723 1421 981 642 3 789 84 122 1845 1101

with every number from 0 to 1999 being equally likely. Note in the above call to nextInt that we gave a bound of 2000. You could have used any bound you want (a nice feature).

Successive runs of this program would each result in a different series of random numbers. This does not match current Java documentation and I need to discover why, but trials I have run show different numbers every run. (Note: for debugging purposes you usually want the same numbers to show up on successive runs. There are ways to do this. EXPLORE!)

Program 1) YOUR FIRST GAME!

Write a program to play a guess a number game. The computer should think of a number between 1 and 1000 and the user has to try and guess the number by typing it in. If the user is incorrect, the computer should say "Too low" or "Too high" and then let them guess again. Keep track of how many guesses they make and print that out and the end along with a congratulations. **Computer Science question: What is the <u>most</u> number of guesses this should ever take?**

Program 2: Print the following pattern, using horizontal tabs (see ASCII code) to separate the numbers in the same line. Let the user decide how many lines to print (i.e. what number to start at). So, if the user enters 4 you should print:

4

3 3

2 2 2

1 1 1 1

Program 3) Draw a rectangle. Write a program that lets the user enter the width and height of a rectangle, then draws the rectangle to the screen using *. For instance, if the user said 4 wide and 3 tall, the program would print out:

****
* *
****

Hint: Try these test cases before showing me: 3X3, 5X4, 1X3, 5X1

Program 4) Fab Factorials. Write a program that prints out a table of factorials. For instance, if the user types in 5, then the output should be:

1!= 1 = 1

2!= 2 x 1 = 2

3!= 3 x 2 x 1 = 6

4!= 4 x 3 x 2 x 1

5!=5 x 4 x 3 x 2 x 1

Sometimes students want to use the **break** statement to break out of a loop. **YOU MAY NOT DO THIS IN THIS COURSE!! The only place a "break" may be used is in a switch-statement.**

6) **Challenge: The 1000th Prime** Write a program that computes and prints any prime number up to the 1000th. The user enters the number of the prime they want, and then the program finds that prime number. When it finds the nth prime number (up to 1000th), then it prints it out and quits. So, if I asked for the 5th prime number the program should print out:

The 5th prime number is 11.

Hints:

a) You need some variables to keep track of which prime number you are on and keep track of where you are in your loop.
b) You only need to test odd numbers for primeness (along with 2). A number isn't prime if another number can divide it (% operator)
c) Think about how many numbers you actually need to check…you can stop sooner than you think!
d) If you want to check that your code is correctly finding primes, you can find primes at http://primes.utm.edu/lists/small/1000.txt