Introduction

Welcome to the Intro to Java: Functional Programming, Lesson 4 problem set! These problem sets are an opportunity for you to practice the concepts you learned in class before moving on to the next lesson. Learning a computer programming language is similar to learning a human language. Nobody can pick it up overnight, there's a lot of vocabulary and syntax to remember. Language learners often speak of the moment when they realized they stopped translating in their head and actually started thinking in their second language. This will happen with Java too! Eventually, you will be able to consider a task that needs coding and immediately imagine what Java code would complete it. To get there, though, requires practice.

That's where the problem sets come in. They aren't mandatory, and they aren't graded. They're just extra learning materials to help you along.

Completing the Problem Sets

There isn't a right or wrong way to work on these. Some problems require you to examine code or do some arithmetic. You can take notes on paper, print this document and use the space provided, or try to do it all in your head—whatever works for you. For the exercises that require programming, we highly recommend that you pick your favorite text editor, open a blank text file, and try writing out the code.

Question 1

What will be printed by this block of Java code?

```
int n = 10;
while (n < 50) {
    n = n * 2;
}
System.out.println(n);
```

A. `10`

B. `40`

C. `50`

D. `80`

Question 1 Solution

D. `80`

The first time the condition `n < 50` is evaluated, `n` will have a value of 10. Since 10 is less than 50, the contents of the loop will be evaluated, and `n` will be doubled to a value of 20. Then the condition is evaluated again. 20 is also less than 50, so the loop will run a second time, doubling `n` again to a value of 40. The condition is still true, so the loop will be run at least one more time, doubling `n` to a value of 80. Now the condition is false, so the loop will stop executing and the program will move to the print statement, printing the current value of `n`, 80.

Question 2

Complete the `factorial()` function below. It should return the product of all the numbers from 1 to the parameter `n`. For example, `factorial(5)` should return 120 because 1 x 2 x 3 x 4 x 5 = 120. Think about what kind of loop you want to use to accomplish this.

Starting code:

```
public int factorial(int n) {

}
```

Question 2 Solution

Example solution code:

```
public int factorial(int n) {
    int factorial = 1;
    for (int i = 1; i <= n; i++) {
        factorial = factorial * i;
    }
    return factorial;
}
```

Question 3

Complete the code in this function to find and return the lowest index in the String array `stringArray` that the String target occurs. If the String `target` does not occur in `stringArray`, -1 should be returned.

Starting code:

```
public int indexOfFirstOccurrence(String[] stringArray, String target)
{
    return -1;
}
```

As an example of how this function should work, this code should print 1, because the word "Java" appears at index 1.

```
String[] sentence = {"Learning", "Java", "is", "fun."};
int indexOfWordJava = indexOfFirstOccurrence(sentence, "Java");
System.out.println(indexOfWordJava);
```

Hint: you cannot compare two Strings using the == operator! This will be false unless the two Strings are actually the same String object, not just two Strings with the same letters. To check whether two Strings have the same letters, use the `equals()` method: `if (myString1.equals(myString2))`.

Question 3 Solution

Example solution code:

```
public int indexOfFirstOccurrence(String[] stringArray, String target)
{
    for (int i = 0; i < stringArray.length; i++) {
        if (stringArray[i].equals(target)) {
            return i;
        }
    }
    return -1;
}
```

Question 4

A savings account yields 5% interest annually. This Java function is designed to determine how many years it will take for you to have $1,000,000 on deposit given an initial value. The parameter represents the initial deposit, and the function should return an integer number of years before there will be $1,000,000 or more in the account. Write a loop to determine the number of years, and return that value.

(Hint: Do we know how many times this loop needs to iterate? Does this mean a for loop or a while loop is more appropriate?)

Starting code:

```
public int yearsTilOneMillion(double initialBalance) {
    return 0;
}
```

Question 4 Solution

Example solution code:

```
public int yearsTilOneMillion(double initialBalance) {
    int years = 0;
    double balance = initialBalance;
    while (balance < 1000000) {
        years++;
        balance = balance * 1.05;
    }
    return years;
}
```

Question 5

Complete the Java function below to print out all the Strings in the String array parameter in reverse order. (The String at the highest index should be printed first, then the String at the next highest index, and so on. The String at index 0 should be printed last.)

For example, if a String array called weekdays had value

```
{"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"}
```

then this function call:

```
printInVerverse(weekdays);
```

would print:

```
Friday
Thursday
Wednesday
Tuesday
Monday
```

Starting code:

```
public void printInReverse(String[] stringArray) {

}
```

Question 5 Solution

Example solution code:

```
public void printInReverse(String[] stringArray) {
    for (int i = 0; i < stringArray.length; i++) {
        //When i has its smallest possible value, 0, the expression
        //below will be the length of the string array minus one,
        //which is the highest index.  When i has its largest possible
        //value, stringArray.length - 1, this expression will be
        //0, which is the the lowest index.
        int indexToPrint = stringArray.length - 1 - i;
        System.out.println(stringArray[indexToPrint]);
    }
}
```

A neater way to solve this would be to structure the for loop differently. For loop variables do not need to begin at 0, and do not always need to go up by one. This loop is structured so that the loop variable `i` begins at the largest index and decreases by 1 every time the loop iterates. (The syntax `i--` is shorthand for `i = i - 1;`.)

```
public void printInReverse(String[] stringArray) {
    for (int i = stringArray.length - 1; i >= 0; i--) {
        System.out.println(stringArray[i]);
    }
}
```

Question 6

Complete the function below, which finds the range covered by an integer array. Inside the function, find the smallest value in the parameter array, and find the largest value, and return the largest value minus the smallest value. If the array has length 0, return -1.

For example, if the variable myIntArray had the value

```
{1, 0, 2, 3, -1, 2}
```

then the function call

```
findRange(myIntArray)
```

would return 4. The largest value in the array is 3, the smallest value is -1, and $3-(-1)=4$.

Starting code:

```
public int findRange(int[] intArray) {
    return -1;
}
```

Question 6 Solution

Example solution code:

```
public int findRange(int[] intArray) {
    if (intArray.length == 0) {
        return -1;
    }
    int smallest = intArray[0];
    int largest = intArray[0];
    for (int i = 1; i < intArray.length; i++) {
        if (intArray[i] < smallest) {
            smallest = intArray[i];
        }
        if (intArray[i] > largest) {
            largest = intArray[i];
        }
    }
    return largest - smallest;
}
```

Question 7

What will be printed by this block of Java code?

```
int rows = 3;
for (int i = 1; i <= rows; i++) {
    String thisRow = "";
    for (int j = 0; j < i; j++) {
        thisRow = thisRow + "#";
    }
    System.out.println(thisRow);
}
```

Question 7 Solution

```
#
##
###
```

The outer for loop will iterate three times: once with `i` taking the value 1, once with `i` taking the value 2, and once with `i` taking the value 3. After the third loop, `i` will go up to 4 and the condition `i < rows` will no longer be true, and the loop will stop. Each time the outer loop executes, a new String called 'thisRow' is created. The inner loop adds `"#"` to the end of `thisRow` each time it iterates. For any given iteration of the outer loop, the inner loop will iterate `i` times: when `i` is 1, the inner loop will iterate once, with `j` taking the value of 0. At the end of the first iteration of the outer loop, `thisRow` is simply `"#"`. The second time the outer loop iterates, `i` is 2, and the inner loop iterates twice: once with `j` taking the value 0 and once with `j` taking the value 1. Since it iterates twice, `"#"` is added to `thisRow` twice. When `thisRow` is printed at the end of the second iteration of the outer loop, it has the value `"##"`. Similarly, the third time the outer loop iterates, the inner loop will iterate three times, and `thisRow` will be `"###"` when printed.

Question 8

Let's improve the `monopolyRoll()` function from the previous problem set. Recall that in Monopoly, players roll two six-sided dice to determine their move. If the same value is on both dice, this is called "rolling doubles," and it means they go again. In the last problem set, you wrote a function that rolled two six-sided dice and, if the values on both die were the same, rolled two more and returned the sum. This time, write a function that does the same except it continues rolling until two non-equal values appear on the two dice. In other words, the function should behave as follows:

1. Generate two random numbers in the 1 to 6 range.
2. If they are not the same, return the sum of all numbers rolled so far.
3. If they are the same, keep track of the total rolled so far and return to step 1.

Think about what kind of loop you want to define to repeat these steps. Again, you may want to define a separate function for generating dice rolls.

Optional challenge: in Monopoly, if a player rolls doubles three times in a row, they go to "jail." Modify your function to keep track of the number of rolls made so far. If three consecutive doubles are rolled, return -1 instead of continuing to roll.

Optional challenge 2: in the last lesson, you learned about while loops and for loops as a way to make a computer program repeat. There is another method for making a program repeat using only functions called recursion. A "recursive" function is one which calls itself inside itself. See if you can write the `monopolyRoll()` function by calling `monopolyRoll()` inside the function itself to handle the case where further rolls are made.

You can get an overview of recursion in Java here:
`https://howtoprogramwithjava.com/java-recursion/`

Question 8 Solution

Example solution code:

```
public int diceRoll(int sides) {
    //This expression generates a random double in the interval
    //[0, sides).
    double randomNumber = Math.random() * sides;
    //Our random number is now in the interval [1, sides + 1)
    randomNumber = randomNumber + 1;
    //Casting the random number to an integer will round it down
    //to an integer in the 1 to sides range.
    return (int) randomNumber;
}

public int monopolyRoll() {
    int roll1 = diceRoll(6);
    int roll2 = diceRoll(6);
    int total = roll1 + roll2;
    while (roll1 == roll2) {
        roll1 = diceRoll(6);
        roll2 = diceRoll(6);
        total = total + roll1 + roll2;
    }
    return total;
}
```

Question 8 Solution (continued)

Example solution code for optional challenge 1 (assume `diceRoll()` is the same as previously shown):

```
public int monopolyRoll() {
    int roll1 = diceRoll(6);
    int roll2 = diceRoll(6);
    int total = roll1 + roll2;
    //An extra variable is added to keep track of how many rolls
    //have been made.
    int rollsSoFar = 1;
    while (roll1 == roll2) {
        //Here, we return -1 if doubles have been rolled too
        //many times in a row.
        if (rollsSoFar >= 3) return -1;
        roll1 = diceRoll(6);
        roll2 = diceRoll(6);
        total = total + roll1 + roll2;
        rollsSoFar = rollsSoFar + 1;
    }
    return total;
}
```

Question 8 Solution (continued)

Example solution code for optional challenge 2 (assume `diceRoll()` is the same as previously shown):

```
public int monopolyRoll() {
    int roll1 = diceRoll(6);
    int roll2 = diceRoll(6);
    if (roll1 != roll2) {
        return roll1 + roll2;
    } else {
        //In the case where the two rolls are equal, we want to
        //return the current roll plus the return value of another
        //call to monopolyRoll(). This is called making a
        //recursive call.  The recursive call will handle making
        //additional rolls, and recursive calls will keep getting
        //made until a roll is made where the two values are not
        //equal.
        return roll1 + roll2 + monopolyRoll();
    }
}
```