1 Project Instructions

- Create a folder named **p1**.
- Copy all your **.java** source code files to **p1**. Do not copy the **.class** files or any testing files.
- Compress the **p1** folder creating a **zip archive** file named **p1.zip**.
- Upload **p1.zip** to the Programming Project 1 (P1) submission page in Gradescope.
- The project deadline may be found in the *Course Schedule* section of the Syllabus.
- Consult the Syllabus for the late submission and academic integrity policies.

2 Learning Objectives

1. To properly use the *Integer* wrapper class.
2. To declare and use *ArrayList<E>* class objects.
3. To write code to read from, and write to, text files.
4. To write an exception handler for an I/O exception.
5. To write a Java class and instantiate objects of that class.

3 Background[1]

Let *list* be a nonempty sequence or list of non-negative random integers, each in the range [0, 32767] and let n be the length of *list*, e.g., here is one sample list:

$list$ = { 2, 8, 3, 2, 9, 8, 6, 3, 4, 6, 1, 9 }

where $n = 12$. List elements are numbered with an **index** or **subscript** starting at 0, so the first 2 in *list* is at index 0, the 8 to the right of the 2 is at index 1, the 3 to the right of the 8 is at index 2, and so on, up to the final 9 which is at index 11.

3.1 Run Up Definition

We define a **run up** in a *list* to be a $(k + 1)$-length subsequence of *list* $(k > 1)$, starting at index i $(0 \le i < n)$ denoted by $list_i$, $list_{i+1}$, $list_{i+2}$, ..., $list_{i+k}$ $(0 \le k < n)$ that is **monotonically increasing**.

Monotonically increasing means that $list_{i+j} \le list_{i+j+1}$ for each $j = 0, 1, 2, 3, ..., k$) or in other words, each successive integer in the run up subsequence is greater than the integer which precedes. Since the length of a run up is $k + 1$ the value of k is 1 less than the length of the run up. For example, if the length of the run up is 5 then k is 4.

Note that a subsequence consisting of only one integer cannot be a run up because if it were, the length of the subsequence would be $k + 1 = 1$, meaning that $k = 0$, but by definition, k must be > 1. Consequently, all runs up must be subsequences of length ≥ 2.

It may help to think of k as counting the number of spaces or holes in between each integer of the run up subsequence. Consider the run up {3 4 6}. The length of this subsequence is 3 meaning that k is 2. Notice that there are two spaces or holes in between 3 and 4 and in between 4 and 6. That is what k represents.

3.2 Run Down Definition

Similarly, a **run down** in *list* is a $(k+1)$-length subsequence $(k > 1)$ starting at index i $(0 \le i < n)$ denoted by $list_i$, $list_{i+1}$, $list_{i+2}$, ..., $list_{i+k}$ $(0 \le k < n)$, that is **monotonically decreasing**.

Monotonically decreasing means that $list_{i+j} \ge list_{i+j+1}$ for each $j = 0, 1, 2, 3, ..., k$) or in other words, each successive integer in the run down subsequence is less than the integer which precedes it. Since the length of a run down is $k + 1$ the value of k is 1 less than the length of the run down. For example, if the length of the run down is 7 then k is 6.

Note that a subsequence consisting of only one integer cannot be a run down because if it were, the length of the subsequence would be $k + 1 = 1$, meaning that $k = 0$, but by definition, k must be > 1. Consequently, all runs down must be subsequences of length ≥ 2.

1 A student asked me where the idea and rationale for this project came from. In statistics, there is a pair of tests of randomness which are called the *runs up* and *runs down* tests. When I was a grad student in computer science, I wrote my master's thesis on pseudorandom number generation algorithms, and to test the randomness quality of the various algorithms I investigated, one of the tests I wrote and performed was the *runs up* and *runs down* tests.

It may help to think of k as counting the number of spaces or holes in between each integer of the run down subsequence. Consider the run down {9 8 6 3}. The length of this subsequence is 4 meaning that k is 3. Notice that there are three spaces or holes: in between 9 and 8; in between 8 and 6; and in between 6 and 3. That is what k represents.

3.3 The Runs Up in the Example List

For the example *list* shown above we have the following runs up:

$list_0$ to $list_1$ = {2, 8} is monotonically increasing because $2 < 8$. The length of this subsequence is $k + 1 = 2$ so $k = 1$.
$list_3$ to $list_4$ = {2, 9} is mono. increasing because $2 < 9$. The length of this subsequence is $k + 1 = 2$ so $k = 1$.
$list_7$ to $list_9$ = {3, 4, 6} is mono. increasing because $3 < 4 < 6$. The subsequence length is $k + 1 = 3$ so $k = 2$.
$list_{10}$ to $list_{11}$ = {1, 9} is mono. increasing because $1 < 9$. The subsequence length is $k = 2$ so $k = 1$.

Note that we do not consider {3, 4} and {4, 6} to be runs up because both those subsequences are subsequences of the subsequence {3, 4, 6} which *is* a run up.

3.4 The Runs Down in the Example List

For the example *list* shown above we have the following runs down.

$list_1$ to $list_3$ = {8, 3, 2} is monotonically decreasing because $8 > 3 > 2$. The subsequence length is $k + 1 = 3$, so $k = 2$.
$list_4$ to $list_7$ = {9, 8, 6, 3} is mono. decreasing because $9 > 8 > 6 > 3$. The subsequence length is $k + 1 = 4$, so $k = 3$.
$list_9$ to $list_{10}$ = {6, 1} is mono. decreasing because $6 > 1$. The subsequence length is $k + 1 = 2$, so $k = 1$.

Note that we do not consider {8, 3} and {3, 2} to be runs down because both those subsequences are subsequences of the subsequence {8, 3, 2} which *is* a run down.

3.5 The Problem

Given a list of integers of variable length, we are interested in the value of k for each run up and run down in the list. In particular we are interested in the total number of runs for each possible value of k, which we shall denote by $runs_k$, $1 < k < n$. For the example *list* with $n = 12$, we have:

k	$runs_k$	runs	
1	4	{2, 8}, {2, 9}, {1, 9}, and {6, 1}	Note: Subsequence lengths are $k + 1 = 1 + 1 = 2$
2	2	{3, 4, 6} and {8, 3, 2}	Note: Subsequence lengths are $k + 1 = 2 + 1 = 3$
3	1	{9, 8, 6, 3}	Note: Subsequence lengths are $k + 1 = 3 + 1 = 4$
4	none	none	
5	none	none	
...			
10	none	none	
11	none	none	

Finally, we define $runs_{total}$ to be be the sum from $k = 1$ to n - 1 of $runs_k$. For the example *list*, $runs_{total} = 4 + 2 + 1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 = 7$.

4 Software Requirements

For this project, you will write a complete Java program which meets these software requirements:

1. This is Software Requirement 1 (SWR 1). Open a file named **p1-in.txt** containing n integers, $1 \leq n \leq 1000$, with each integer in [0, 32767]. There will be one or more integers per line. Here is a sample input file for the *list* of §3.

   Sample p1-in.txt
   ```
   2 8 3
   2 9
   8
   6
   3 4 6 1 9
   ```

2. SWR2. The program shall compute $runs_k$ for $k = 1, 2, 3, ..., n - 1$.

3. SWR 3. The program shall compute $runs_{total}$.

4. SWR 4. The program shall produce an output file named **p1-runs.txt** containing $runs_{total}$ and $runs_k$ for k = 1, 2, 3, ..., n - 1. The file shall be formatted as shown in the example file below:

   Sample p1-runs.txt

```
runs_total: 7
runs_1: 4
runs_2: 2
runs_3: 1
runs_4: 0
runs_5: 0
runs_6: 0
runs_7: 0
runs_8: 0
runs_9: 0
runs_10: 0
runs_11: 0
```

5. SWR 5. If the input file **p1-in.txt** cannot be opened for reading (because it does not exist) then display an error message on the output window and immediately terminate the program, e.g., you program shall display a message similar to this and then exit:

   run your program... test case input file could not be opened...

```
Oops, could not open 'p1-in.txt' for reading. The program is ending.
```

6. SWR 6. If the output file **p1-runs.txt** cannot be opened for writing (e.g., because write access to the file is disabled or you are trying to open the file in a location which is prohibited) then display an error message on the output window and immediately terminate the program, e.g.,

   run your program... test case output file could not be opened...

```
Oops, could not open 'p1-runs.txt' for writing. The program is ending.
```

5 Software Design Requirements

You are free to create your own software design for the program or you may follow our software design described here.

1. This is Software Design Requirement 1 (SWDR 1). Declare a class named *Main* in *Main.java*. This class shall contain the *main*() method. The *main*() method shall instantiate an object of the *Main* class and call *run*() on that object, see template code below. Other than *main*() there shall not be any static or class methods within *Main*.

```
// Main.java
public class Main {

    public static void main(String[] pArgs) {
        Main mainObject = new Main();          // Or you can just write: new Main().run();
        mainObject.run()                       // in place of these two lines.
    }

    private void run() {
        // You will start writing code here to implement the software requirements.
    }
}
```

2. SWDR2. One of the primary objectives of this programming project is to learn to use the *java.util.ArrayList<E>* class. Therefore, you **are not permitted** to use primitive 1D arrays. Besides, you will quickly discover that the ArrayList class is more convenient to use in this program than a 1D array would be.

3. SWDR3. *ArrayList<E>* is a generic class meaning: (1) that it can store objects of any reference type, e.g., *E* could be the classes *Integer* or *String*; and (2) when an *ArrayList* object is declared and instantiated, we must specify the class of the objects that will be stored in the *ArrayList*. For this project, you need to define an ArrayList that stores integers, but you cannot specify that your ArrayList stores **int**s because **int** is a primitive data type and not a class. Therefore, you will need to use the *java.lang.Integer* wrapper class:

```
ArrayList<Integer> list = new ArrayList<>();
int x = 1;
list.add(x); // Legal because of Java autoboxing.
```

4. SWDR 4. You must write an **exception handler** that will catch the *FileNotFoundException* that gets thrown when the input file does not exist (make sure to test this). The exception handler will print the friendly error message as shown in Software Requirement 5 and immediately terminate the Java program. To immediately terminate a Java program we call a static method named *exit*() which is in the *java.lang.System* class. The *exit*() method expects an **int** argument. For this project, it does not matter what **int** argument we send to *exit*(). Therefore, terminate the program this way by sending -100 to *exit*().

```
try {
     // Try to open input file for reading
} catch (FileNotFoundException pException) {
     // Print friendly error message
     System.exit(-100);
}
```

5. SWDR 5. Similar to Item 4, you must write an exception handler that will catch the *FileNotFoundException* that gets thrown when the output file cannot be opened for writing. The exception handler will print the message as shown in Software Requirement 6 and then terminate the program by sending -200 to *exit*().

6. SWDR 6. Your programming skills should be sufficiently developed that you are beyond writing the entire code for a program in the *main*() method or all of it in *run*(). You shall partition the functionality of the program into multiple methods. Remember, a method should have one function, i.e., it should do one thing and it should generally be short. If you find a method is becoming too lengthy or overly complicated because you are trying to make that method do more than one thing, then divide the method into 2, 3, 4, or more smaller methods, each of which does one thing.

7. SWDR 7. Avoid making every variable or object an instance variable. For this project, we do not require any instance variables in the class *Main* so **you shall not declare any instance variables** in the class. Rather, all variables should be declared as local variables in methods and passed as arguments to other methods when appropriate.

8. SWDR 8. Neatly format your code. Use proper indentation and spacing. Study the examples in the book and the examples the instructor presents in the lectures and posts on the course website.

9. SWDR 9. Put a comment header block at the top of each method formatted thusly:

```
/**
 * A brief description of what the method does.
 */
```

10. SWDR 10. Put a comment header block at the top of each source code file—not just for this project, but for every project we write, e.g.,

```
/**********************************************************************************************
 * CLASS: classname (classname.java)
 *
 * DESCRIPTION
 * A description of the contents of this file.
 *
 * COURSE AND PROJECT INFORMATION
 * CSE205 Object Oriented Programming and Data Structures, semester and year
 * Project Number: project-number
 *
 * AUTHOR: your-name, your-asuriteid, your-email-addr     ** Note: Include a second author line if
 * AUTHOR: your-name, your-asuriteid, your-email-addr     you work with a partner **
 *********************************************************************************************/
```

5.1 Software Design: Pseudocode

To help you complete the program, you may follow our design in this section and implement this pseudocode if you wish. Alternatively, you may create and implement your own design, so long as your program meets the software requirements discussed in §4 and the software design requirements specified in §5.

Class *Main*

Declare a static int constant RUNS_UP which is equivalent to 1
Declare a static int constant RUNS_DN which is equivalent to -1

-- Note: In the Java implementation, all of these methods will be private methods declared inside the *Main* class.
-- Static method *main*() must call *run*(), so in essence, *run*() becomes the starting point of execution.

Static Method *main*(*pArgs* : String[]) **Returns** Nothing
    Instantiate an object of this class and then call method *run*() on the object
End Method *main*

-- It is imperative that *run*() catches and handles the *FileNotFoundException*s that may get thrown in
-- *readInputFile*() and *writeOutputFile*() when the input and output files cannot be opened for reading and writing.

Instance Method *run*() **Returns** Nothing
    Declare ArrayList of Integers named *list*
    Try
        list ← *readInputFile*("p1-in.txt")
    Catch *pException* : FileNotFoundException
        Display error message as described in SWR 4
        Terminate the program with exit code of -100
    End
    Declare and create an ArrayList of Integers named *listRunsUpCount*
    Declare and create an ArrayList of Integers named *listRunsDnCount*
    listRunsUpCount ← *findRuns*(*list*, RUNS_UP)
    listRunsDnCount ← *findRuns*(*list*, RUNS_DN)
    Declare ArrayList of Integers *listRunsCount* ← *mergeLists*(*listRunsUpCount*, *listRunsDnCount*)
    Try
        writeOutputFile("p1-runs.txt", *listRunsCount*)
    Catch *pException* : FileNotFoundException
        Display error message as described in SWR 5
        Terminate the program with exit code of -200
    End
End Method *run*

-- *pList* is the ArrayList of Integers that were read from "p1-in.txt". *pDir* is an int and is either RUNS_UP or
-- RUNS_DN which specifies in this method whether we are counting the number of runs up or runs down.

Instance Method *findRuns*(*pList* : ArrayList of Integers, *pDir* : int) **Returns** ArrayList of Integers
    listRunsCount ← *arrayListCreate*(*pList.size*(), 0) -- Create ArrayList of proper size and init each element to 0
    Declare int variables initialized to 0: *i* ← 0, *k* ← 0 -- the left arrow represents the assignment operator
    While *i* < *pList.size*() - 1 **Do**
        If *pDir* is RUNS_UP **and** *pList* element **at** *i* **is** ≤ *pList* element **at** *i* + 1 **Then** -- This is a run up
            Increment *k*
        ElseIf *pDir* **is** RUNS_DN **and** *pList* element **at** *i* **is** ≥ *pList* element **at** *i* + 1 **Then** -- This is a run down
            Increment *k*
        Else
            If *k* **does not equal** 0 **Then**
                Increment the element at index *k* of *listRunsCount*
                k ← 0
            End if
        End If

        Increment *i*
    End While

If *k* **does not equal** 0 **Then**
Increment the element at index *k* of *listRunsCount*
End If
Return *listRunsCount*
End Method *findRuns*

Instance Method *mergeLists*(*pListRunsUpCount* : ArrayList of Integers, *pListRunsDnCount* : ArrayList of Integers)
Returns ArrayList of Integers
listRunsCount ← *arrayListCreate*(*pListRunsUpCount.size*(), 0)
For *i* ← 0 to *pListRunsUpCount.size*() - 1 Do
Set element *i* of *listRunsCount* to the sum of the elements at *i* in *pListRunsUpCount* and *pListRunsDnCount*
End For
Return *listRunsCount*
End Method *mergeLists*

Instance Method *arrayListCreate*(*pSize* : **int**; *pInitValue* : **int**) **Returns** ArrayList of Integers
Declare and **create** an ArrayList of Integers named *list*
Write a for loop that iterates *pSize* times and each time call *add*(*pInitValue*) to *list*
Return *list*
End Method *arrayListCreate*

-- Make sure to throw the *FileNotFoundException* that is raised when the output file cannot be opened for writing.

Instance Method *writeOutputFile*(*pFilename* : String; *pListRuns* : ArrayList of Integers) **Returns** Nothing
Throws *FileNotFoundException*
out ← **open** *pFilename* for writing
out.println "runs_total: ", *the sum of the elements of pListRuns*
For *k* ← 1 to *pListRuns.size*() - 1 **Do**
out.println "runs_", *k*, " ", the element at index *k* of *pListRuns*
End For
Close *out*
End Method *output*

-- Make sure to throw the *FileNotFoundException* that is raised when the input file cannot be opened for reading.

Instance Method *readInputFile*(*pFilename* : **String**) **Returns** ArrayList of Integers **Throws** *FileNotFoundException*
in ← **open** *pFilename* for reading
Declare and **create** an ArrayList of Integers named *list*
While there is more data to be read from *in* **Do**
Read the next integer and add it to *list*
End While
Close *in*
Return *list*
End Method *readInputFile*

End Class *Main*