COMP 150 Lab 5 - Arrays, Strings, etc

Github doesn't support inline latex formulas...
This pdf version of the instructions will be more readable, the $latex\_formulas$ will be replaced with actual equations.

In this lab:

- What arrays are and how to create, access and edit them.
- Traversing arrays and the enhanced `for` loop.
- Passing by reference, redundant references and copying.
- Nestability and multi-dimensional arrays.
- Algorithms, pseudocode and implementation via searching sorting.
- `String`s as sequences of `char`s
- Indexing `String`s, substrings, and other `String` methods.
- Escape sequences.
- Regular expressions and pattern matching.
- ArrayLists.

Task 1

You'll want to start by downloading the starting code for task 1 and opening the zipped project in your IDE.

The zipped project contains two files: `IntArrayMethods.java` and `IntArrayMethodsClient.java`. The client is complete, but none of the methods in `IntArrayMethods.java` are complete.

There is documentation for the entire project provided in the `docs` folder; open the `index.html` file (in the `docs` folder) in any web browser to see descriptions of all contained methods as they should function when completed. Note that this documentation matches the JavaDoc comments in `IntArrayMethods.java` in the `src` folder (the ones that start with `/**`); those comments are the text from which the html documentation was generated. IntelliJ can can generate documentation from such comments. To do so, go to `Tools` → `Generate JavaDoc`, select the source files for the JavaDoc Scope and select any desired output directory in which the generated documentation will be placed. You don't need to do this now; the documentation for this project has already been generated. In future projects there will sometimes be JavaDoc comments included in source files, but the documentation itself will be omitted, and you can choose whether to generate the documentation or just read the comments themselves.

You should complete all methods in `IntArrayMethods.java` as you go through the reading. `IntArrayMethodsClient.java` is complete, and can be used to test your methods in `IntArrayMethods.java`. Note that in order for the tests to be valid, you should complete the methods in `IntArrayMethods.java` in the order they're presented, as some of those further down assume that those above them are correct. For instance, if your `copy` method isn't working correctly, then many of the tests for sorting methods will fail even if the sorting method itself is correct, and if your `equals` method isn't correct then most subsequent tests will fail.

Remember not to do more work than necessary! Specifically, try not to repeat work; use the simpler methods in the more complex methods to avoid typing out the same loops over and over.

Arrays

So far, each variable we have created has stored one piece of data. The statement `int x = 5;` creates a variable `x` which can store a single integer value, and assigns that value to `5`. But sometimes it is convenient to store multiple values in sequence in a single variable; this is where arrays come into play. Arrays can store a sequence of values (all of a specified type).

We will start our discussion of arrays with an example:

```
String[ ] dayNames = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};
```

The snippet above creates an array of `String`s called `dayNames` containing 7 `String` values (the days of the week). The pieces of data contained in an array are called its **elements**. The array above has seven elements: the names of the seven days of the week, stored in `String` form.

Declaration

Adding square brackets `[ ]` after the data type in a declaration indicates that an array is being declared. The data type for an array if `String`s is `String[ ]`, that for an array of `int`s is `int[ ]`, and so on. Arrays can be declared to store any data type. All of the following would are valid array declarations (though some of them require importing the class stored in the array):

```
int[ ] testScores;
double[ ] monthlyRainfall
String[ ] dayNames;
Point[ ] corners;
MyClass[ ] arrayFilledWithMyClassInstancesWithAnObnoxiouslyLongIdentifier;
```

Initialization

Arrays can be initialized in two ways. The most obvious way is with an array literal, which consists of curly braces `{ }` containing the values in the array, separated by commas.

For instance:

```
int[ ] numbers = { 100, 97, 99, 82, 85, 74, 93 };
```

The line above declares an array of integers called `numbers` and initializes it as with seven integer values specified in the curly braces.

Arrays can also be initialized without initial values; the number of elements in the array must be specified, but their values need not be specified. This is done like an object construction, because arrays are objects:

```
int[ ] numbers = new int[7];
```

When an array is declared in this fashion, its elements are all set to the default value for the specified data type; `null` for objects, `0` for numeric primitives, `false` for `boolean`s, and the null character `'\0'` for `char`s. The array `numbers` above would be an array containing seven `int`s, each with value `0`.

Indexing

Each element of an array has an integer location, called its **index**. The first element in an array has index `0`, the second has index `1`, and so on.

If an array is initialized as follows:

```
int[ ] numbers = { 100, 97, 99, 82, 85, 74, 93 };
```

Then the following table shows the `numbers` array's values and their corresponding indices:

VALUE	**100**	**97**	**99**	**82**	**85**	**74**	**93**
INDEX	**0**	**1**	**2**	**3**	**4**	**5**	**6**

Notice that the largest index is `6` even though the array has 7 elements; this is because the indexing starts at `0`, not at `1`! You can think of the index as the **distance** from the start of the array; the first element **is** the start of the array, so its distance from the start is `0`.

Array elements can be accessed with their index in square brackets; to get the value `100` from the `numbers` array above, one would use the expression `numbers[0]`. To get the `99`, one would use the expression `numbers[2]`.

Traversing and the `length` attribute

The number of elements in an array can be accessed using its `length` attribute with the accessor operator `.`. The length of the numbers array above is `7`, and this value can be accessed with `numbers.length`.

You can go through every element in an array by incrementing an index in a loop. Because the first index is `0`, the last index is 1 less than the array length, so the index must remain less than the array length. The two loops in the snippet below both print all of the elements of the `dayNames` array.

```
String[ ] dayNames = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};

for (int index = 0; index < dayNames.length; index++)
{
    System.out.println( dayNames[index] );
}

int index = 0;
while (index < dayNames.length)
{
    System.out.println( dayNames[index] );
    index++;
}
```

Traversing with the enhanced `for` loop

There is an additional type of `for` loop for iterating through sequences. The loop below does the same as the two above (it prints every value in the `dayNames` array). The `:` can be thought of as "in" as far as understanding what the loop syntax means in english.

```
for (String name : dayNames)
{
    System.out.println( name );
}
```

EXERCISE 1 Consider the following array declaration and instantiation:

```
int[] myIntArray = { 1, 3, 5, 7, 9 };
```

1. What is the `length` of `myIntArray`?
2. What is the smallest index in `myIntArray`?
3. What value is stored at `myIntArray[2]`?
4. What is the largest index in `myIntArray`?

EXERCISE 2 Consider the following snippet:

```
double[] myDoubleArray = new double[10];

for (int i = 0; i < 10; i++)
{
    myDoubleArray[i] = Math.sqrt(i);
}
```

1. Before the `for` loop runs, what does `myDoubleArray` contain?
2. After the `for` loop runs, what is the first element in the array?
3. After the `for` loop runs, what is the last element in the array? What is its index?

EXERCISE 3 Find the error(s) in the following snippet. Fix it so it does what the comments say. What do you think an `ArrayIndexOutOfBoundException` signifies?

```
// declare and instantiate an array with the first 8 fibonacci numbers
int[] myIntArray = {0, 1, 1, 2, 3, 5, 8, 13};

// print out every element in the array
for (int i = 1; i <= myIntArray.length; i++)
{
    System.out.println(myIntArray[i]);
}
```

EXERCISE 4 Create a simple program which declares an array containing the names of all 12 months, in order. Then, traverse the array twice; once with a `for` loop using indexes, and once with an enhanced `for` loop using the `:` operator. Your indexed loop should not use a hard-coded `12`, and should instead use the array's `length` attribute. Try adding and removing elements from the array and ensure that you don't need to change the loops to still print every element in the array, regardless of how many or few there are.

Editing arrays

Values within arrays can be reassigned just like variables.

```
int[ ] numbers = {9, 8, 3, 6, 5, 4, 3, 2, 1};

numbers[2] = 7;
```

EXERCISE 5

Create a program which repeatedly prompts the user for integers from 0 to 100. Each input integer should be classified on a grading scale as follows:

- 0-59 : F
- 60-69 : D
- 70-79 : C
- 80-89 : B
- 90-100 : A

Store, in an array, the number of A's, B's, C's, D's and F's. When the user enters a number greater than 100 or less than 0, stop looping and print the number of occurences of each grade that the user input. You may use the `Scanner` class's `nextInt` method, and do not need to worry about dealing with bad user inputs.

Checking arrays for equality

Two arrays containing elements of the same data type are equal if:

1. They have the same length.
2. At each index, the elements in the two arrays are equal.

Checking if two arrays are equal, then, requires first checking that their lengths are the equal, and then iterating through them checking that the elements at each index are equal.

Note that requirement 2 above varies in meaning based on data types. With primitives, it means that the elements are equal as determined by the `==` operator. With objects it generally means that they are equal according to the objects' `equals` method, to check if their contained data is the same, though it sometimes might mean that they are equal using the `==` operator if you want to check if the arrays reference the same objects, as opposed to objects of the same type containing the same data.

Arrays are passed by reference

Arrays are objects, so they are passed by reference. In the snippet below, `array_1` and `array_2` are actually references to **the same array**, so editing `array_2` also edits `array_1`.

```
int[ ] array_1 = {9, 8, 3, 6, 5, 4, 3, 2, 1};
int[ ] array_2 = array_1;

System.out.println("array_1's address : " + array_1.toString());
System.out.println("array_2's address : " + array_2.toString());

array_1[2] = 7;

System.out.println("array_1's element at index 2 : " + array_1[2]);
System.out.println("array_2's element at index 2 : " + array_2[2]);
```

This has many implications. The most immediate example is: if a method edits an array that was passed in as an argument, the original array is also edited! To demonstrate, consider the following example:

```
public class Sandbox
{
    public static void main(String[] args)
    {
        // declare and initialize an int
        int myInt = 5;

        // print the int's value
        System.out.println("myInt before the incrementInt call : " + myInt);
```

```
        // call incrementInt on the int
        incrementInt(myInt);

        // print the ints value again
        System.out.println("myInt after  the incrementInt call : " + myInt);

        // declare and initialize an array of ints
        int[ ] myIntArray = {0, 2, 4, 6, 8};

        // use a loop to print all of the array's values
        System.out.print("myIntArray before the incrementIntArrayElements call : ");
        for (int x : myIntArray)
        {
            System.out.print(" " + x);
        }

        // call incrementIntArrayElements on the array
        incrementIntArrayElements(myIntArray);

        // print the arrays values again
        System.out.print("\nmyIntArray after  the incrementIntArrayElements call : ");
        for (int x : myIntArray)
        {
            System.out.print(" " + x);
        }
    }

    public static void incrementInt(int integer)
    {
        // increment the integer
        integer++;
    }

    public static void incrementIntArrayElements(int[] integer_array)
    {
        // go through the array, incrementing each element
        for (int i = 0; i < integer_array.length; i++)
        {
            integer_array[i]++;
        }
    }
}
```

Copying Arrays

Often, it is necessary to create an edited version of an array while keeping the original unedited. Because arrays are passed by reference, and not by value, it is necessary to copy the original array and then edit the copy.

For instance, when we sort an array, we must choose to do so either **in place** by rearranging the original array to be sorted, or **not in place** by creating a copy to sort and leaving the original unsorted.

To copy an array:

- declare and initialize another array with the same data type and length.
- copy each element from the original array into the copy.

The following class `IntArrayUtils` contains the method `copyIntArray` which takes an `int[ ]` as an argument, and copies this array into a second array, which it returns. You'll need to implement a very similar method in `IntArrayMethods.java` in the downloaded code for task 1.

```
public class IntArrayUtils
{
    public static int[ ] copyIntArray(int[ ] original)
    {
        int[ ] copy = new int[original.length];

        for (int i = 0; i < original.length; i++)
        {
            copy[i] = original[i];
        }
        return copy;
    }
}
```

`StringBuilder`

When building a `String` in pieces, it is prudent to use the `StringBuilder` class instead of repeatedlying using `String` addition. The reason has to do with how memory is allocated for `String` variables under the hood. When you create a `String`, enough space is allocated to store all of its contained characters, and then the variable is given a pointer to that allocated space to reference the `String` value.

Whenever a `String` variable is given a new value (say, through `String` addition), space is allocated for the entirety of this new `String` value and then the new value is written, character by character, into this new

space. Then, the space for the old `String` value is deallocated (assuming there are no other references to it).

Memory allocation and deallocation are expensive and we generally want to avoid doing them more than necessary; it is also wasteful to repeatedly copy the start of a `String` into larger and larger spaces when adding more to the end of it. In the example below, each time the `+=` operator is used in the loop, memory for the new `String` value is allocated, then the value is copied into this space, and then the memory for the old `String` value is deallocated.

```
Scanner scan = new Scanner(System.in);

System.out.println("Enter 10 words");
String words = "";

for (int i = 0; i < 10; i++)
{
    words += scan.next() + " ";
}

System.out.println("You entered: " + words);
```

The `StringBuilder` stores a list of individual `String`s, so that they can all be "added up" at once. The snippet below does the same as the one above, but uses `StringBuilder` to save time.

```
Scanner scan = new Scanner(System.in);

System.out.println("Enter 10 words");
StringBuilder words = new StringBuilder();

for (int i = 0; i < 10; i++)
{
    words.append(scan.next());
    words.append(" ");
}

System.out.println("You entered: " + words.toString());
```

This does not help the memory allocation and deallocation issue in terms of the **number** of allocations, as each "chunk" of the final `String` mush have space allocated, but it does make the **size** of the allocated chunks smaller, which makes allocation easier. It also saves us from repeatedly copying the first word in the `String` each time another word is added to the end.

If you type the first snippet (which uses `+=` on `String`s in a loop) in IntelliJ, the `+=` operator will be highlighted in yellow. Mousing over this highlight will reveal the warning `String concatenation '+=' in loop` along with a suggestion (in blue) to use the `StringBuilder` class instead. Clicking this suggestion will change the snippet to one very similar to the second one above, which uses `StringBuilder`.

One context in which the `StringBuilder` is useful is that of creating a `String` representation of an array.

`StringBuilder`'s have a lot of functionality which we haven't discussed here. Check out their documentation; search for "Java 8 StringBuilder" and the first search result should be Oracle's documentation.

EXERCISE 6 Look up the `StringBuilder` documentation. Read the description of the `reverse` method. The description is... well, accurate, but not particularly descriptive. The first time I read this description, I wondered: Does it reverse the order of its contained `String`s, but leave the `String`s themselves in their original order, or does it reverse the order of the characters within the contained `String`s as well? The documentation is not ambiguous here; it specifies that the contained character sequence is reversed. Often documentation is either ambiguous or well-written but too brief to provide any familiarity with the tool, and this often leaves the reader with questions like this. Test the `StringBuilder`'s reverse method. Is the output of the snippet below `"WorldHello"` or `"dlroWolleH"`?

```
StringBuilder hello = new StringBuilder();

hello.append("Hello");
hello.append("World");

System.out.println(hello.reverse().toString());
```

EXERCISE 7 Recall that in the previous lab, we created a method which takes as input a `String` and outputs `true` if that `String` is a palindrome. We used loops to do so. Try to repeat this venture with less work by finding an appropriate method from the `StringBuilder` class in its documentation.

Swapping array elements

When swapping array elements, there is a small problem which must be overcome. If one value is used to overwrite the other value, then the overwritten value is no longer accessible and cannot be used to overwrite the other. This problem is dealt with useing a temporary storage variable. The snippet below swaps the values at index `3` and `5` in the `numbers` array.

```
int[ ] numbers = {9, 8, 7, 4, 5, 6, 3, 2, 1};

int temp = numbers[3];
numbers[3] = numbers[5];
numbers[5] = temp;
```

Pseudocode, Sorting and Searching Arrays

Often it is desired to sort array data. There are many ways to do this. Before continuing, watch this video on Selection Sort.

Selection Sort is an **algorithm**: a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation. In this case, the problem being solved is that of sorting an array in non-decreasing order.

Algorithms are often represented in **pseudocode**. Pseudocode is simply a description of the algorithm written in a structured way (generally somewhat similar to code), but pseudocode is not written in any particular programming language; it is instead written for humans to read, so that they might implement algorithms in any suitable language.

Pseudocode is very loosely defined. A paragraph which unambiguously, precisely and completely describes an algorithm is also pseudocode. Anything that unambiguously, precisely and completely defines all of the steps can qualify, though generally a more structured code-like approach is preferred. Arguably, any adequately descriptive recipe could qualify as pseudocode, assuming the instructions unambiguously and precisely describe how to prepare the desired dish.

Selection Sort

Below we have pseudocode for Selection Sort:

```
Selection Sort
IN: arr is an unsorted array of numbers, with length n, indexed 0 to n-1

for i in [0, n-2]
    min = i
    for j in [i+1, n-1]
        if arr[min] > arr[j]
            min = j
    swap the elements in arr at indices i and min

OUT: arr has been sorted in non-decreasing order
```

Notice that the algorithm above is not written in Java, and will not compile in Java. It is instead written in a less formal format for humans to read. It is written in my preferred style of pseudocode. Blocks, denoted in Java with curly braces `{}` in Java, are denoted via indentation above; you can tell what is "inside" the outer loop because its contents are indented below it. Pseudocode often uses a mix of notations from programming and mathematics. In the pseudocode above, the pairs in brackets `[0, n-1]` and `[i, n]` denote closed intervals. In general, these would be closed intervals of real numbers, but because we're using the elements of these intervals as indexes we know they must be integers---this does not need to be specified as formally as it would in an actual program, because the pseudocode is intended to be read by humans. This is most apparent in the last line of the outer loop, which is written out as a sentence.

The structure of the pseudocode is arbitrary; I simply wrote it in a way that make sense to me. Pseudocode can really be any unambiguous sequence of instructions. Any recipe which describe how and when to add ingredients is arguably pseudocode, so long as it is sufficiently unambiguous. Usually, when instructions are referred to as pseudocode, it is in the context of either mathematics or programming, though, and in these situations psuedocode usually looks like a mix of code and english describing an algorithm independent of any programming language.

When an algorithm is translated from pseudocode into a programming language, this is called an **implementation** of the algorithm. Below is an implementation of Selection Sort for use on `int[ ]`s in Java.

```
public static void selectionSort(int[ ] arrayToSort)
{
    for (int i = 0; i < arrayToSort.length-1; i++)
    {
        int min = i;
        for (int j = i+1; j < arrayToSort.length; j++)
        {
            if (arrayToSort[j] < arrayToSort[min])
            {
                min = j;
            }
        }
        int temp = arrayToSort[i];
        arrayToSort[i] = arrayToSort[min];
        arrayToSort[min] = temp;
    }
}
```

Searching Unsorted Arrays

It is sometimes necessary to search an array for a specified value. If the value is there, it's index is generally what is returned. If the array isn't sorted, this can be a painfully slow procedure, as it is necessary to simply iterate through the array, one element at a time, looking for the specified value. This is called a sequential search:

```
Sequential Search
IN:     array A, desired value v

i = 0
while i < length(A)
    if A[i] is v, then return i
    i = i + 1

return -1

OUT:    the index of v, if A contains c
    otherwise, -1
```

Searching Sorted Arrays

There are a variety of ways to search a sorted array for a specified value, all faster than sequential search.

Check out this video on binary and sequential search. Note that the pseudo code written in the video looks more like Java than mine, but it is still pseudocode, not Java.

Bubble Sort

Watch this video on Bubble Sort.

EXERCISE 8 Write pseudocode for Bubble Sort.

Array Nestability and Multi-Dimensional Arrays

Arrays can contain elements of any data type, including other arrays. In this way, multi-dimensional arrays can be made. The following snippet declares and instantiates an array representing the multiplication table for integers 0 through 9.

```
int[ ][ ] multiplicationTable = {
        {0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
        {0,  1,  2,  3,  4,  5,  6,  7,  8,  9},
        {0,  2,  4,  6,  8, 10, 12, 14, 16, 18},
        {0,  3,  6,  9, 12, 15, 18, 21, 24, 27},
        {0,  4,  8, 12, 16, 20, 24, 28, 32, 36},
        {0,  5, 10, 15, 20, 25, 30, 35, 40, 45},
        {0,  6, 12, 18, 24, 30, 36, 42, 48, 54},
        {0,  7, 14, 21, 28, 35, 42, 49, 56, 63},
        {0,  8, 16, 24, 32, 40, 48, 56, 64, 72},
        {0,  9, 18, 27, 36, 45, 54, 63, 72, 81}
};
```

Here, each index of the `multiplicationTable` is a **sub-array** (i.e. a row) in the 2D array above; `multiplicationTable[0]` is `{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}`, `multiplicationTable[3]` is `{0, 3, 6, 9, 12, 15, 18, 21, 24, 27}`, and so on.

These sub-arrays are, themselves, arrays, so they can be indexed; `multiplicationTable[0][0]` is the first element of `multiplicationTable[0]`, i.e. `1`. `multiplicationTable[5][6]` is `30`.

Note that `multiplicationTable` has 10 sub-arrays, each of which contain 10 integers. The same array can be made by first declaring an array with default values (`0` s) and then assigning new values in a nested loop like this:

```
int[ ][ ] multiplicationTable = new int[10][10];

for (int i = 0; i < 10; i++)
{
    for(int j = 0; j < 10; j++)
    {
        multiplicationTable[i][j] = i * j;
    }
}
```

The above examples are 2 dimensional (notice the 2 sets of brackets in the data type `int[ ][ ]`). Arrays can be made with any desired dimension.

In a multi-dimensional array, sub-arrays do not need to all be the same shape. For instance, the following is a completely valid array initialization:

```
int[ ][ ] numbers = {
        {1},
        {2, 3},
        {4, 5, 6, 7},
        {8, 9}
};
```

Here `numbers` is an array containing 4 `int[ ]`s which have lengths 1, 2, 4 and 2 respectively.

EXERCISE 9 Create and test a method which iterates through a 2D `int` array like the `numbers` array above and prints every value in the array. Your implementation should make no assumptions about the number of contained 1D arrays, nor should it make any assumptions about the lengths of the contained 1D arrays. In other words, you should be able to change the "shape" of the array and your program should still work.

EXERCISE 10 Make your test from the previous exercise more compact as follows: create a 3D `int` array, containing several 2D arrays on which to test the method to print 2D array elements. Iterate through this 3D array, testing the method on each contained 2D array.

EXERCISE 11 Create a program which populates a 10×10 2D array called `distances` such that the value stored at each index `[x][y]` is the euclidean distance from point $(0, 0)$ to point (x, y). You should calculate the values in the array using nested `for` loops. Your program should also print values which allow you to observe whether your solution is correct or not.

EXERCISE 12 Repeat the previous exercise, but with a 3D array. The value stored at index `[x][y][z]` should be the euclidean distance from point $(0, 0, 0)$ to point (x, y, z).

`String` again, and Regular Expressions

We've explored some `String` methods briefly in prior labs. Here, we will explore a few of them in more detail. We will also touch on the `StringBuilder` class, which can be used to (you guessed it) build `String`s piece by piece, and regular expressions, which can be used to match patterns to more efficiently interpret and categorize `String` values.

`String`s are sequences

Much like arrays, `String`s are sequences whose elements (`char`s) can be accessed with their index.

Consider the following snippet:

```
String hello = "Hello";
```

The statement above creates a `String` called `hello` and stores in it the value `"Hello"`. We can visualize this value and its indexes much like an array:

`char`	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`
index	**0**	**1**	**2**	**3**	**4**

Where arrays use square brackets to access elements by their index, `String`s use the `charAt` method to access elements at specified indexes. From the for example, in the snippet above, `hello.charAt(1)` is an expression which would return the `'e'` from `"Hello"`.

EXERCISE 13 Create and test a method which takes as input a `String` value and prints each character in that `String` on its own line. For instance, when given the value `"Hello World!"` it should print out:

```
H
e
l
l
o

W
o
r
l
d
!
```

The `substring` method

A piece of a `String` containing multiple characters can be gotten with the `substring` method. The `substring` method takes two arguments: the start and end indexes of the desired substring. It returns a substring starting at the provided start index and ending with the index before the provided end index. With `hello` defined in the snippet above, `hello.substring(1, 4)` would return the substring `"ell"`; each character, starting at index `1` and before index `4`. The end index can be omitted; if it is, then all characters from the start index to the end of the `String` are included in the substring. For instance, `hello.substring(3)` returns `"lo"`: every character from index `3` to the end.

The `indexOf` and `lastIndexOf` methods

The `String` class's `indexOf` and `lastIndexOf` methods can be used to find the index of specified characters or substrings. `indexOf` is polymorphic. Its simplest form takes a `char` as an argument, and outputs the index of the first occurrence of that `char` in the `String` calling it, or `-1` if there no occurence of the designated character. For instance, `hello.indexOf('l')` returns `2`, because `2` is the index of the first `'l'` character.

`lastIndexOf` is very similar to `indexOf`, and the difference in behavior is implied by its name. What does `hello.lastIndexOf('l')` return with `hello` defined as above?

There are more complex forms of each of these functions, which take extra arguments to perform more complex tasks (like, say, finding the index of the first occurrence of the designated character on or after a specified index, or finding the starting index of a designated substring). Check out the Java 8 String API to learn more.

EXERCISE 14 Create a program which prompts the user for their full name, and then prints out each individual

name within their full name (correctly, whether their name has 1 part or n parts). It should print out each space-separated part of of the full name. on its own line. Given `"Ryan"` it should print out `Ryan`. Given `"John Jacob Jingleheimer Schmidt"` it should print:

```
John
Jacob
Jingleheimer
Schmidt
```

This can be done many ways. You could use the `substring` method alongside the `indexOf` method in a loop. You could loop through character by character deciding whether to print each new character on the current line or on the next line. The best (easiest) way, however, involves a `String` method that we haven't discussed, which will allow you to easily split up the `String` into an array of its space-separated (i.e. space-delimited) elements. I encourage you to explore the `String` documentation (search "Java 8 String" to find it). I recommend that you implement **all three** of these different ways to perform the same task.

Escape Sequences

In a `String` literal, the backslash `\` is used to start **escape sequences**. Escape sequences are sequences of characters that have a different meaning than the literal sequence. They are often used to represent "special" characters (like newline `'\n'` and tab `'\t'`). They are also used to ensure that other characters are interpretted literally, when they would otherwise have some additional meaning in context. For instance, putting the double quote `'"'` character in a `String` literal requires a backslash, otherwise the double quote character **ends the string**.

Try assigning and printing each of the following `String` literals to a `String` variable and printing them (if possible). Try to figure out what each of the escape sequences means, which examples below are invalid and why they're invalid.

`"She said "Hello""`
`"She said \"Hello\""`
`"Up here\nDown there"`
`"Block\n\tIndentedBlock"`
`"\"`
`"\\"`
`"\\\\"`
`"\\\\\\"`
`"Not this\rThis only but why?"`

The last one escape sequence, `\r`, denotes a **carriage return**, an antique carried from the typewriter into

early (bad) encodings for text files. On typewriters, going to a new line was done with two keystrokes, one to go down a line (the line feed) and one to go back to the left side (the carriage return). Most modern editors use just a line feed `\n` to denote **both** of these. Some editors (primarily on Windows machines) still use the carriage return after the line feed, which will lead to multiple headaches throughout your years of practice as a programmer when reading data from files.

EXERCISE 15 Create and test program which reads the contents of a text file character-by-character and prints out the number of new line characters (`'\n'`) and tab characters (`'\t'`) individually. Testing will involve creating a text file (or many) on which to test your program, of course. You may find this stackoverflow post useful in figuring out how to read an entire file's contents into one `String` .

Regular Expressions

Brace yourself. Regular expressions (often called regex) are **useful**, but they're also **tedious** and **essentially unreadable**. Be patient with this section; taking the time here to make sure you understand how to read and write these expressions (which often look like heiroglyphics) will provide an unmatched way to parse user inputs, among other things.

Check out this video on regular expressions. You can download the text editor that he's using (called Atom) here if you'd like to experiment with regular expressions in it.

EXERCISE 16 Write regular expressions to recognize each of the following patterns:

1. The letter `a` , alone.
2. The letter `a` , followed by the letter `b` .
3. The letter `a` , repeated 1 or more times.
4. The letter `a` , repeated 0 or more times.
5. The letter `a` , repeated exactly 5 times.
6. The letter `a` , repeated 3 to 5 times.
7. The letter `a` or the letter `b` , but not both.
8. The letter `a` followed by 4 or more of the letter `b` .
9. The letter `a` , maybe followed by the letter `b` but maybe not.
10. Any positive number of `a` s and `b` s, in any order.
11. A word. (Here, a "word" is any sequence of 1 or more "word characters", denoted with `\w`).
12. Literally any string.
13. The word `"Captain"` , followed by a space and then any single other word.
14. The word `"camelCase"` or the word `"UPPER_SNAKE_CASE"` .
15. Any word written in `UPPER_SNAKE_CASE`
16. Any word written in either `UPPER_SNAKE_CASE` or `lower_snake_case` but not a mix of the two.
17. Any sequence of 1 or more words, with spaces between them (but not before the first one or after the last

one) ending with a period.

18. Any sequence of 1 or more words, with spaces between them (but not before the first one or after the last one), where any word except the last one might (optionally) be immediately followed (before the space) by any of the characters `;` `:` `,`, and the last word is followed by one of `.` `!` `?`.

The video covers some universals of regular expressions. Most regular expression implementations have significantly more functionality built in. In Java, regular expressions are implemented through the `Pattern` class (imported from `java.util.regex`). You can find the documentation here.

The regular expression `\d{3}-\d{3}-\d{4}` could be used to match phone numbers in the form **XXX-XXX-XXXX**. In order to use this regular expression in Java, we need to create a member of the `Pattern` class using the regular expression in `String` form. The program below prompts the user for a phone number in a specified format, and then checks if the input matches the pattern:

```
import java.util.Scanner;

public class Sandbox
{
    public static void main(String[] args)
    {
        String phoneNumberRegex = "\\d{3}-\\d{3}-\\d{4}";

        System.out.println("Enter a phone number in the form XXX-XXX-XXXX");

        Scanner keyboard = new Scanner( System.in );
        String userPhoneNumber = keyboard.next();

        if (userPhoneNumber.matches(phoneNumberRegex))
        {
            System.out.println("That looks like a phone number to me!");
        }
        else
        {
            System.out.println("What is this garbage? I said a PHONE NUMBER.");
        }
    }
}
```

Notice the difference between the regular expression in the program above (in the Java implementation) and the one to match phone numbers before it: all of the escapes `\` are doubled. This is because the regex is being processed twice: first as a `String` and then as a regular expression. In other words, the `String` literal `\\d{3}-\\d{3}-\\d{4}` results in a `String` storing the sequence of characters

`\d{3}-\d{3}-\d{4}`, because the backslash character `\` is a metacharacter in `String`s, so it must be escaped to appear in a `String`.

EXERCISE 17 Create, test and debug implementations for in Java for the regular expressions created in the previous exercise. Feel free to skip some of the early ones if you're certain you understand them, but be certain to do all of the more complex ones (specifically the last three).

`ArrayList`

The `ArrayList` class is essentially an array wrapped in an object with a bunch of extra methods and capabilities. Anything you can do with an `ArrayList` can also be done with arrays with enough determination, but the pile of extra functionality that the `ArrayList` class provides (and which you therefore don't need to code) can make many tasks much simpler, particularly of the size necessary for the array is unknown when it is constructed. You can find its documentation here.

One of the most convenient differences between an `ArrayList` and an array is that `ArrayList`s are dynamically sized. This means that the number of elements they contain can be changed, and does not need to be specified during construction.

`ArrayList`s can only store objects, they cannot store primitives. Any time it would be convenient to store primitives in an `ArrayList`, just use the corresponding wrapper classes. For example, `int` data cannot be stored in an `ArrayList`, but `Integer` data can.

The `ArrayList` class contains many methods; we will only explore a few of them here, but the documentation describes them all.

The example below shows how to instantiate, and `ArrayList` with no elements, how to add elements to the end of the list, and how to access those elements, with their index.

```java
import java.util.ArrayList;
import java.util.Scanner;

public class Sandbox
{
    public static void main(String[] args)
    {
        ArrayList<String> words = new ArrayList<>();
        Scanner scan = new Scanner ( System.in );

        System.out.println("Enter words. Enter \"END\" to stop.");
        String userInput = scan.next();

        while (!userInput.equals("END"))
        {
            words.add(userInput);
            userInput = scan.next();
        }

        System.out.println("You entered " + words.size() + " words.");
        System.out.println("Here are the words you entered :");
        for (int i = 0; i < words.size(); i++)
        {
            System.out.println(i+1 + ". " + words.get(i) + " ");
        }
    }
}
```

Note the differences in how the `ArrayList` is accessed compared to the standard array: elements are accessed with their using the `get` method. The number of elements in the array is accessed using the `size` method. The value at a specified index can be changed using the `ArrayList`'s `set` method.

EXERCISE 18 Create a program which gets user inputs in `String` form. Inputs which are `"up"`, `"down"`, `"left"` or `"right"` should be stored in an `ArrayList` until the user enters the sentinel `"END"`. Once the user enters the sentinel, your program should treat the user's input as directions for navigating a 2D grid. It should start a `Point` at coordinates $(0, 0)$ and move then move 1 unit in the specified direction for each `String` stored in the `ArrayList`. It should print each point it passes.

If the user enters the sequence:

```
up
right
down
left
left
END
```

then the output should be something like:

```
start: (0, 0)
up   : (0, 1)
right: (1, 1)
down : (1, 0)
left : (0, 0)
left : (-1, 0)
```

Answers to Selected Exercises

SOLUTION 1

1. The `length` of `myIntArray` is `5`; it has 5 elements.
2. The smallest index in any array is `0`. Indexing starts at `0`.
3. The value stored at index `2` in `myIntArray` is `5`. Index `0` contains `1` and index `1` contains `3`.
4. The largest index is always 1 less than the length, because indexing starts at `0`. Thus, the largest index in `myIntArray` is `4`.

SOLUTION 2

1. Before the `for` loop runs, `myDoubleArray` contains 10 of the default `double` value, which is `0.0`.
2. After the `for` loop runs, the first element in the array (at index `0`) is `Math.sqrt(0)`, which is `0.0`.
3. After the `for` loop runs, the last element in the array (at index `9`) is `Math.sqrt(9)`, which is `3.0`.

SOLUTION 3

An `ArrayIndexOutOfBoundsException` signifies that an array (or some other indexable object) was accessed with an invalid index. The index might be negative, or it might be larger than the largest index in the

array.

```
// declare and instantiate an array with the first 8 fibonacci numbers
int[] myIntArray = {0, 1, 1, 2, 3, 5, 8, 13};

// print out every element in the array
for (int i = 0; i < myIntArray.length; i++)
{
    System.out.println(myIntArray[i]);
}
```

SOLUTION 4

```java
public class Sandbox
{
    public static void main(String[] args)
    {
        String[] monthNames = {
                "January",
                "February",
                "March",
                "April",
                "May",
                "June",
                "July",
                "August",
                "September",
                "October",
                "November",
                "December"
        };

        for (int i = 0; i < monthNames.length; i++)
        {
            System.out.println(monthNames[i]);
        }

        for (String month : monthNames)
        {
            System.out.println(month);
        }
    }
}
```

SOLUTION 5

Program:

```java
import java.util.Scanner;

public class GradeCounter
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner( System.in );
```

```
        int userInput;

        final int A_INDEX = 0;
        final int B_INDEX = 1;
        final int C_INDEX = 2;
        final int D_INDEX = 3;
        final int F_INDEX = 4;

        String[] gradeNames = {"A", "B", "C", "D", "F"};
        int[] grades = {0,0,0,0,0};

        while (true)
        {
            System.out.print("Enter the next grade : ");
            userInput = scan.nextInt();

            if (userInput < 0 || userInput > 100)
            {
                break;
            }
            else if (userInput >= 90)
            {
                grades[A_INDEX] += 1;
            }
            else if (userInput >= 80)
            {
                grades[B_INDEX] += 1;
            }
            else if (userInput >= 70)
            {
                grades[C_INDEX] += 1;
            }
            else if (userInput >= 60)
            {
                grades[D_INDEX] += 1;
            }
            else
            {
                grades[F_INDEX] += 1;
            }
        }

        System.out.println("\nYou entered the following grades:");

        for(int i = 0; i < grades.length; i++)
```

```
        {
            System.out.println(gradeNames[i] + " : " + grades[i]);
        }
    }
}
```

Sample Run:

```
Enter the next grade : 89
Enter the next grade : 88
Enter the next grade : 95
Enter the next grade : 77
Enter the next grade : 89
Enter the next grade : 99
Enter the next grade : 92
Enter the next grade : 87
Enter the next grade : 83
Enter the next grade : 72
Enter the next grade : 66
Enter the next grade : 67
Enter the next grade : 89
Enter the next grade : 72
Enter the next grade : 55
Enter the next grade : -1

You entered the following grades:
A : 3
B : 6
C : 3
D : 2
F : 1

Process finished with exit code 0
```

SOLUTION 7 The `StringBuilder`'s `reverse` method is useful here. A `String` is a palindrome if and only if reversing it does not change its value. Thus, if a `String` equals the result of reversing it, then the `String` is a palindrome.

SOLUTION 8

Here's some pseudocode for BubbleSort. Your style of pseudocode does not need to match mine, as long as it is complete, exact and unambiguous.

```
Bubble Sort
IN: arr is an unsorted array of numbers with length n, indexed 0 to n-1

for i in [1, n-1]
    for j in [0, n-i-1]
        if arr[j] > arr[j+1]
            swap the elements in arr at indices j and j+1

OUT: arr is sorted
```

SOLUTION 9

The method below will print out a 2D int array as described.

```java
public static void print2DIntArrayElements(int[][] arrayToPrint)
{
    for (int i = 0; i < arrayToPrint.length; i++)
    {
        System.out.println("Print sub-array at index " + i);
        for (int j = 0; j < arrayToPrint[i].length; j++)
        {
            System.out.println("\t" + arrayToPrint[i][j]);
        }
    }
}
```

SOLUTION 16

1. `a`
2. `ab`
3. `a+`
4. `a*`
5. `a{5}`
6. `a{3,5}`
7. `a|b`
8. `ab{4}b+`
9. `ab?`
10. `[ab]+`
11. `\w+`
12. `.*`
13. `Captain \w+`

14. `(camelCase)|(UPPER_SNAKE_CASE)`
15. `[A-Z0-9\$_]+`
16. `[A-Z0-9\$_]+|[a-z0-9\$_]+`
17. `\w+( \w+)*\.`
18. `\w+([,:;]? \w+)*[.!?]`

Lab Tasks

Task 1

You probably downloaded the task 1 starter code at the beginning of the lab. If not, download it now. Complete all methods (in order) in `ArrayMethods.java`, and run `ArrayMethodsClient.java` to test.

Task 2

Implement and test methods called `getIntFromUser` and `getDoubleFromUser` which use regular expressions to validate the user input `String`s before using the `Integer` and `Double` classes to parse the input. In both methods, the user should be re-prompted in an infinite loop until their input is valid.

A valid `int` literal consists of the following parts:

1. Optionally, a single `+` or `-` sign
2. 1 or more digits (`0` - `9`)

A valid `double` literal consists of:

1. Optionally, a single `+` or `-` sign
2. 1 or more digits
3. A period
4. 0 or more digits
5. optinally, a single `d` or `D` character

Optionally, when you've successfully tested and debugged both of these methods, try to do the same thing using `try` and `catch` instead of regular expressions (this will take some research).

Task 3

Create a text-based game of tic-tac-toe. The board state should be stored in a 3×3 2D `char` array, whose

elements are `'x'`, `'o'` or `'\0'` (the null character, for board spaces that are empty). After each turn, the board should be printed in a format like this:

```
     1   2   3

A    o |   |
    --- --- ---
B       | x |
    --- --- ---
C    o |   | x
```

At the beginning of the game, a random player (`x` or `o`) should be randomly assigned to go first. Then, the game should enter an infinite loop which:

1. Gets the current player's move.
    - The move should be gotten in `String` form using coodrinates like `A1`, `C2`, etc, and should be validated using regular expressions.
    - If the user's `String` represents space on the board, it should then be checked to ensure that that space is not already occupied.
    - The user should be reprompted in an infinite loop until they enter a valid move (i.e. a valid `String` input representing an empty board space).
2. Updates the board array with the player's move.
3. Checks if there is a winner. If so, prints who won and terminates.
4. Checks if the board is full. If so, prints that it is a tie and terminates.

Technically, this could all be done in one class consisting of just a main method. It is strongly recommended that you try to organize the steps of the game into smaller methods. Recall that you can pass the array representing the game board into methods and edit it in them, so steps 1 and 2 can be done together in a method which takes the game board and the current player (`x` or `o`) as an argument.

Optionally, put a little research into `enum` s, which can help you better organize the game.

Optionally, if you want to go really hard, try making a bot to play against, so you only have to make plays for one of the players. This bot can be as simple or complex as you like; it might make random moves, or it might play optimally using the minimax strategy, or anywhere in between.