**ICS 102 Lab08: One-dimensional Arrays**

**Objectives:**

* Learning how to declare, create and initialize 1D-arrays
* Learning how to use loops in 1D-array manipulation.
* Learning some 1D-array applications:
  1. Learn how to search arrays (using linear search).
  2. Solving problems that require the computation of a data value from a group of data values, and then comparing the computed value with each data value in the group. Example:
     + Counting the number of grades below and/or above the average.
  3. Tokenizing a string using **String** split method.
  4. Using array elements as counters.
  5. Implementing a dictionary using parallel arrays.

**Note:** 1D-arrays have other applications that will be covered in ICS201 and ICS202 such as sorting and implementing other data structures, such as [heaps](http://en.wikipedia.org/wiki/Heap_(data_structure)), [hash tables](http://en.wikipedia.org/wiki/Hash_table), [deques](http://en.wikipedia.org/wiki/Double-ended_queue), [queues](http://en.wikipedia.org/wiki/Queue_(data_structure)), [stacks](http://en.wikipedia.org/wiki/Stack_(data_structure)), and [strings](http://en.wikipedia.org/wiki/String_(computer_science)).

**1.1 1D-Array declarations**

In Java arrays are objects that contain collections of elements all of the same type. The elements are arranged in consecutive memory locations.

The syntax of array declaration is:

**dataType[ ] arrayName;**

or

**dataType arrayName[ ];**

Examples:

int[ ] studentID; // declares a reference variable **studentID** of type **int[ ]**

double price[ ]; // declares a reference variable **price** of type **double[ ]**

String[ ] studentName; //declares a reference variable **studentName** of type **String[ ]**

Note: Declaring a variable of array type does not create an array object or allocate any space for array components. It creates only the variable itself, which can contain a reference to an array (i.e., it can contain the starting address of an array object).

**1.2 Array Creation**

An array is created by an array creation expression or an array initializer:

1. Array creation expression: **new dataType[numberOfElements]**

**dataType[ ] arrayName;**

**arrayName = new dataType[numberOfElements];**

or

**dataType arrayName[ ];**

**arrayName = new dataType[numberOfElements];**

Note: An array declaration may be combined with an array creation expression:

**dataType[ ] arrayName = new dataType[numberOfElements];**

or

**dataType arrayName[ ] = new dataType[numberOfElements];**

Examples:

**int[ ] studentID = new int[5];** // creates an int array object of length 5 and

//elements: studentID[0], studentID[1],

// studentID[2], studentID[3], studentID[4]

// and assigns the starting address of the object

// to the reference variable **studentID**

**String[ ] studentName = new String[7];**

**Note:** It is possible to provide the array size at run-time:

**System.out.println(“Enter the array size: ”);**

**int size = scanner.nextInt( );**

**double[ ] grades = new double[size];**

1. **Array Initializers**

An array initializer is written as a comma-separated list of expressions, enclosed by braces { and }. An *array initializer* may be specified in a declaration, or as part of an array creation expression, to create an array and provide some initial values.

Example: The following 4 array initializations are equivalent:

|  |
| --- |
| int[] x = {1, 2, 3, 4, 5}; |
| int x[] = {1, 2, 3, 4, 5}; |
| int[] x = new int[] {1, 2, 3, 4, 5}; |
| int[] x;  x = new int[] {1, 2, 3, 4, 5}; |

Other array initialization examples:

int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };

char ac[] = { 'n', 'o', 't', ' ', 'a', ' ',

'S', 't', 'r', 'i', 'n', 'g' };

Note:

* Each initializer must be assignment-compatible with the array's component type, or a compile-time error occurs.
* The length of the array to be constructed is equal to the number of initializers.

Once an array object is created, **its length never changes**. To make an array variable refer to an array of different length, a reference to a different array must be assigned to the variable.

Example:

**int[ ] num1 = {20, 5, 6, 12}; // num1 refers to an array object of length 4**

**int[ ] num2 = {30, 15, 60, 40, 59, 70, 100}; // num2 refers to an array object of length 7**

**num1 = num2; // both num1 and num2 refer to the array object with length 7**

**// the array object with length 4 has no reference and will be**

**// garbage collected**

**1.3 Array Members**

An array object has the following member:

* The public final field **length**, which contains the number of components of the array. The field **length** may have a positive or zero value.

**1.4 Array Access**

A component of an array is accessed by an array access expression that consists of an expression whose value is an array reference followed by an indexing expression enclosed by [ and ], as in A[i].

All arrays in Java are 0-origin. An array with length ***n*** can be indexed by the integers **0** to ***n*-1**.

Arrays must be indexed by **int** values; **short**, **byte**, or **char** values may also be used as index values because they are subjected to unary numeric promotion and become **int** values.

An attempt to access an array component with a long, float or double index value results in a compile-time error.

All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an **ArrayIndexOutOfBoundsException** to be thrown.

**1.5 Partial arrays**

It is possible to use a 1D-array of size **n** in which only the first **m** elements of the array are initialized, where **m < n**. Such an array is called a **partial array**. In such a case, it is necessary to know the actual number of initialized elements.

**1.6 Parallel arrays**

Two or more full or partial 1D-arrays of the same size where index-corresponding elements store related data.

Example: The following three parallel arrays store Quiz01 and Quiz02 grades for 5 students:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 3 | 4 | 5 |
| IDs | 90012350 | 90012351 | 90012352 | 90012353 | 90012354 |
|  |  |  |  |  |  |
| Quiz01 | 10.0 | 7.5 | 6.4 | 9.6 | 8.2 |
|  |  |  |  |  |  |
| Quiz02 | 5.4 | 6.0 | 8.3 | 4.0 | 7.6 |

**Note**: Usually parallel arrays are implemented as arrays of references. Arrays of references will be introduced in a later ICS 102 lab.

**2. Using loops in traversing 1D-arrays**

Once an array is initialized, a loop can be used to access its elements:

**int[] values = {5, 16, 7, 3, 6, 9};**

**for(int k = 0; k < values.length; k++)**

**System.out.print(values[k] + " ");**

# Traversing an array using for-each loop (Advanced or Enhanced for loop):

The for-each loop was introduced in Java5. It is used to access each successive value in an array or a collection.

The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

## **Syntax of for-each loop:**

**for**(data\_type variable : array | collection){ . . . }

Example: Here is a loop written as both a *for-each* loop and a basic *for* loop.

double[] ar = {1.2, 3.0, 0.8};

int sum = 0;

for (double d : ar) {

sum += d;

}

And here is the same loop using the basic *for-loop*. It requires an extra iteration variable.

double[] ar = {1.2, 3.0, 0.8};

int sum = 0;

for (int i = 0; i < ar.length; i++) {

sum += ar[i];

}

Limitations of for-each loop:

Although the enhanced *for* loop can make code much clearer, it can't be used in some common situations.

* **Only access no modification**. Array or collection elements can not be modified within a for-each loop.

Example:

int[] x = {1, 2, 3, 4, 5};

for (int num : x){

num = num\*2; // only changes num, not the array element

System.out.print(num + " ");

}

System.out.println();

for(int k = 0; k < x.length; k++)

System.out.print(x[k] + " ");

Outputs:

2 4 6 8 10

1 2 3 4 5

* **Only single array or collection**. It is not possible to traverse two arrays or collections at once

Example: The following cannot be easily converted to a for-each loop:

for(int k = 0; k < x.length; k++){

if(x[i] == y[i]){

. . .

}

}

* **Only single element**. Use only for single element access, eg., not to compare successive elements.

Example: The following cannot be easily converted to a for-each loop:

for(int k = 0; k < x.length - 1; k++){

if(x[i] > x[i + 1]){

. . .

}

}

* **Only iterates forward over the array or collection in single steps**.

Example: The following cannot be converted to a for-each loop:

for(int k = x.length - 1; k >= 0; k--){

System.out.println(x[i]);

* **To get the index of an element, an additional variable is required**.

**int[] x = {10, 2, 30, 4, 50};  
 int k = 0; // additional variable  
 for(int num : x){  
 if(num > 8)  
 System.out.print(k + " ");  
   
 k++;  
 }**

**3. Some 1D-array applications**

**3.1 Searching a 1D-array using Linear search:**

// . . .

**boolean found = false;**

**for(int i = 0; i < x.length; i++){**

**if(key == x[i])**

**found = true;**

**break;**

**}**

**if(found)**

**System.out.println(key + " is in the array");**

**else**

**System.out.println(key + " is not in the array");**

**3.2 Solving problems that require the computation of a data value from a group of data values, and then comparing the computed value with each data value in the group.**

Example: Counting the number of grades above the average in a group of grades

Put the grades in a 1D-array

Use a loop to find the average

Use another loop to find the number of grades above the average

**// . . .**

**double sum = 0, average;**

**int count = 0, k;**

**for(k = 0; k <= arraySize – 1; k++){**

**System.out.printf("Enter grade#%d: ", k + 1);**

**array[k] = scanner.nextDouble();**

**sum += array[k];**

**}**

**average = sum / arraySize;**

**for(k = 0; k <= arraySize – 1; k++){**

**if(array[k] > average)**

**count++;**

**}**

**3.3 Tokenizing a string using String split method**

String tokenization is a process where a string is broken into several parts. Each part is called a token. For example, if "I am going" is a string, the discrete parts—such as "I", "am", and "going"—are the tokens. Java provides standard classes and methods to implement the tokenization process. One such method is the **String** class **split** method:

**public String[] split(String regularExpression)**

The split method splits **the invoking string object** string around matches of the given [regular expression](file:///D:\171\ICS102_Term171\Software\jdk-8u144-docs-all\docs\api\java\util\regex\Pattern.html#sum). A regular expression defines a search pattern for strings. The tokens are returned in a **String** array.

Example:

**import java.util.Scanner;  
 public class TokenizeString{  
 public static void main(String[] args){  
 Scanner scanner = new Scanner(System.in);   
 System.out.println("Enter string to tokenize: ");  
 String str = scanner.nextLine();  
 System.out.println("Using blank and tab characters as delimiters: ");  
 String[] tokens1 = str.split("[ \t]+");  
 for(int k = 0; k < tokens1.length; k++)  
 System.out.println(tokens1[k]);  
  
 System.out.println("Using the delimiters: \"[ \\t.,;:?!]+\"");  
 String[] tokens2 = str.split("[ \t.,;:?!]+");  
 for(String token : tokens2)  
 System.out.println(token);  
   
 scanner.close();  
 }  
 }**

**3.4 Array elements as counters**

To avoid long if-structures or switch statements, array elements may be used as counters in certain problems. Example, find the frequency of each lowercase alphabetic character in a text-file.

import java.util.Scanner;

import java.io.\*;

public class CountFrequencies {

public static void main(String[ ] args) throws IOException{

FileInputStream fis = new FileInputStream("input.txt");

int[ ] countLowerCaseLetter = new int[26]; // initialized to zeros

int characterCode;

char ch;

while((characterCode = fis.read()) != -1){

ch = (char) characterCode;

if(Character.isLowerCase(ch))

countLowerCaseLetter[ch - 'a']++;

}

System.out.printf("%9s%12s%n%n", "Character", "Frequency");

int index;

for(ch = 'a'; ch <= 'z'; ch++){

index = ch - 'a';

if(countLowerCaseLetter[index] != 0)

System.out.printf("%9c%12d%n", ch, countLowerCaseLetter[index]);

}

}

}

**3.5 Implementing a Dictionary using parallel arrays**

**import** java.util.Scanner;

**public** **class** Dictionary01 {

**public** **static** **void** main(String[] args){

Scanner scanner = **new** Scanner(System.***in***);

String[] word = {"door", "house", "boy", "girl", "book", "car", "pen", "tree"};

String[] meaning = {"بَابٌ", "بَيْتٌ", "وَلَدٌ", "بِنْتٌ", "كِتَابٌ", "سَيَّارَةٌ", "قَلَمٌ", "شَجَرَةٌ"};

System.***out***.printf("The English-Arabic dictionary contains the following words:%n");

**for**(**int** k = 0; k < word.length; k++)

System.***out***.print(word[k] + " ");

System.***out***.printf("%n%nEnter an English word to translate into Arabic: ");

String inputWord = scanner.next();

**boolean** found = **false**;

**for**(**int** k = 0; k < word.length; k++){

**if**(word[k].equalsIgnoreCase(inputWord)){

System.***out***.printf("%s %s", inputWord, meaning[k]);

found = **true**;

**break**;

}

}

**if**(! found)

System.***out***.printf("Sorry! The word %s does not exist in the dictionary.", inputWord);

scanner.close();

}

}

**Laboratory Tasks**

Task 1

Write a Java program that prompts for and reads 10 integers from the user into an **integer** array of size 10, it

then:

- calculates the sum and average of positive numbers of the array and displays them.

[Note: consider 0 as positive]

- Displays the array indexes where positive integers appear in the array.

**Note**: The program must display the message: "There are no positive values" if the array does not contain any

positive value.

Sample program runs:

|  |
| --- |
|  |
|  |
|  |

Task 2

Write a Java program that prompts for and reads the size **n** of a 1D-array of type **double**. If the size is invalid,

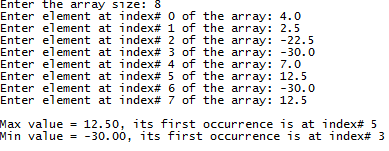
the program displays an appropriate error message and loops until the user enters a valid size. If the size is

valid, the program prompts for and reads **n** numbers into the array, it then finds and displays the maximum

and minimum values and the indexes (subscripts) of the first occurrences of these maximum and minimum

values in the array.

Sample program run:



Task 3

Write a Java program that prompts for and reads the size **n** of an integer array, it prints an appropriate error message and loops as long as **n** is not a valid array size. If the array size is valid the program initializes an integer array of size **n** with **n** pseudo-random integers from 1 to 30.

Print the initialized array then prompt the user to enter a **target value** to search for in the array by using **linear search**. If present, your program should display its first position in the array, otherwise print a message that the value is not in the array.

Note: To generate a random integer in the interval [1, 30] use the method call:

**(int)(Math.random( )\*30) + 1**

**random( )** returns a double value in the interval [0.0, 1.0)

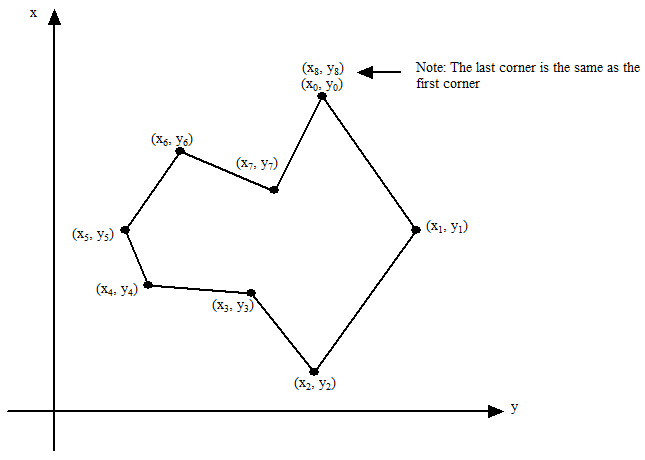
Sample program runs:

|  |
| --- |
|  |
|  |

Task 4

Consider a closed polygon with **n** distinct corners: **(x0, y0), (x1, y1), . . . , (xn-1, yn-1)**

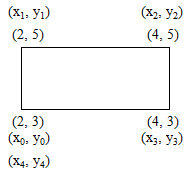
Example:



The area of such a polygon can be computed by the formula:

where **(xn , yn)** is the repeated first point of the polygon, i.e., the corner **(x0, y0)**.

Example: For the rectangle:



The area is 1/ 2.0 \* | (x1 + x0)(y1 – y0) + (x2 + x1)(y2 – y1) + (x3 + x2)(y3 – y2) + (x4 + x3)(y4 – y3)|

= 1/ 2.0 \* | (2 + 2)(5 – 3) + (4 + 2)(5 – 5) + (4 + 4)(3 – 5) + (2 + 4)(3 – 3)|

= 4.0

To implement the formula for a polygon with **n** distinct corners, use two parallel arrays of x- and y-coordinates of the polygon corners, each array of size **n + 1**, and in which the coordinates of the first corner are repeated in the last parallel array elements:

Example: For the above rectangle, the parallel arrays are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x0 |
| x-coordinates | 2.0 | 2.0 | 4.0 | 4.0 | 2.0 |
|  |  |  |  |  |  |
| y-coordinates | 3.0 | 5.0 | 5.0 | 3.0 | 3.0 |
|  | y0 | y1 | y2 | y3 | y0 |

Write a Java program that prompts for and reads the number **n** of corners of a polygon. If the size is less than 3, the program displays an appropriate error message and terminates; otherwise it prompts for and reads the **n** x- and y-coordinates of the polygon corners into two parallel arrays of size n + 1. The program then initializes the last elements of the two arrays to the x- and y-coordinate of the first corner respectively. It then calculates and displays the area. Use square meters for the area.

Sample program runs:

|  |
| --- |
| **Enter number of corners > 2: 4**  **Enter x, y coordinates for corner 1: 0.0 0.0**  **Enter x, y coordinates for corner 2: 1.0 1.0**  **Enter x, y coordinates for corner 3: 0.0 2.0**  **Enter x, y coordinates for corner 4: -1.0 1.0**  **Polygon area = 2.00 cm^2** |
| **Enter number of corners > 2: 4**  **Enter x, y coordinates for corner 1: 1.0 0.0**  **Enter x, y coordinates for corner 2: 5.0 4.0**  **Enter x, y coordinates for corner 3: 4.0 5.0**  **Enter x, y coordinates for corner 4: 0.0 1.0**  **Polygon area = 8.00 cm^2** |

Task 5

Write a Java program that prompts for and readsthe size ofan array of type **double**. If the size is invalid, the program displays an appropriate error message and loops until the user enters a valid size. If the array size is valid the program prompts for and reads the array elements. The program then determines whether the array is sorted in non-decreasing order or not by displaying one of the following messages: “The array is sorted in non-decreasing order” or “The array is not sorted in non-decreasing order”.

**Note:** An array **x** with **n** elements is sorted in non-decreasing order if:

x1 ≤ x2 ≤ x3 ≤ . . . ≤ xn

Sample program runs:

|  |
| --- |
|  |
|  |