**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Practical 11 – Arrays**

Up until now, everything that we have done in terms of programming has almost always relied on creating single variables to store single values. There are times when it is very useful to have a variable that is capable of storing several items of data. This is where the **array** comes in.

**What is an Array?**

Essentially, an array is a **list**. We are familiar with the use of lists in everyday life e.g. a class register contains a list of students’ names, a shopping list, a price list. In programming languages, an array is a **data structure** used to *simulate* a list.

In terms of computer memory, an array is a collection of memory cells which store a group of (usually) **related values** that are **all of the same type**, so we might have an array of integer values or an array of floating-point values **but not a mixture of the two**.

The entire array is always given a **name** by which it can be uniquely identified.

### Declaring Arrays

Arrays can be of **any type** i.e. int, char, float, double, String etc.

To actually create an integer array called marks which can store the marks of 10 students, we would declare it as follows:

int marks[] = new int[10];

Similarly, to declare an array of float called heights which stores the heights of up to 100 people we could declare it as follows:

float heights[] = new float[100];

Notice that arrays are declared different to ordinary variables. For one thing, there are **square brackets** after the array name on the left hand side of the statement and after the type on the right hand side of the statement. Inside the square brackets on the right is the **size** of the array. This numbers tells us the **maximum number of values** the array can store. **Once the size of the array has been set at declaration time, it is fixed and cannot be changed for that array**.

Also note the use of the keyword **new** on the right hand side. This tells us immediately that **an array is an object**, rather than just an ordinary variable. In general, an array is declared as follows:

arrayType arrayName[] = new arrayType[arraySize];

It could equally well be declared as follows:

arrayType[] arrayName = new arrayType[arraySize];

Both forms are in common use – you can take your pick.

So, in reality, when you have a line of code that creates an array, the **array name** is actually an **object reference** rather than a variable.

**Accessing Array Elements**

All arrays contain zero or more **elements**. These are the individual “slots” within the array that contain the actual values the array is storing.

Each element has a **subscript** or **index** which corresponds to the position of the element within the array.

Consider for now an array of **marks** which contains a list of the marks for 10 students

Array name: marks

23 87 76 12 34 78 90 66 22 34

subscript number: 0 1 2 3 4 5 6 7 8 9

We would refer to the **first** element of the array above as follows

marks[0] (and **not marks[1]** as you and I might think!!).

It is crucial to remember that the **first element in any array has subscript number zero**

We can see in this case the **value** of marks[0] is 23.

We would refer to the last element of the array here as **marks[9]** and we see its value is 34.

You should **never** **refer to an element outside the range of the array**, so e.g. marks[10] would be incorrect in this case as we are trying to access the 11th element of an array that only contains 10 elements. Java throws up a **runtime error** if we attempt to do this and our program duly crashes (unless evasive action has been taken by our code)

**Assigning values to Array Elements**

Naturally, we can also do things like assigning values to elements of an array e.g.

marks[2]=67;

could be used to set up the 3rd element of the array above with the value 67 (it would have the effect of replacing the current value 76 with 67). Note that the square brackets are used again here when referring to a specific element of an array.

Not surprisingly, we can also read in values dynamically (at runtime) into arrays so we could have, for example,

System.out.print(“Please enter a value for the 5th element of the array: ”);

marks[4] = input.nextInt();

which would read in a value for the 5th element of the array called marks. Note again that we need to specify explicitly using a subscript number (4) where exactly in the array we wish to have the number read into.

And it goes without saying that we can output the elements of an array in a similar fashion using either console or graphical output methods:

JOptionPane.showMessageDialog(null,“The values of elements 2 and 6 of the marks array are ” + marks[1] + “ and ” + marks[5], “Array Values”,JOptionPane.PLAIN\_MESSAGE);

### Initialising an Array

It is possible to **initialise** an array when it is being declared i.e. it can be set up with values at declaration time e.g.

int numbers[] = {23,56,34,45,12};

will create an integer array called numbers, allocate 5 elements to the array and set those elements up with the values shown.

If you do not initialise an array with values then they automatically receive default values as follows: int, float and double arrays get all zeros, char arrays get the null character (‘\0’) and object arrays (including String arrays) get all nulls.

### Processing Elements of an Array

When it comes to processing an array, a **for loop** will often be used, given that the number of elements in the array is always known and also can often be quite large.

So, for example, to read 10 values into the marks integer array we could do the following:

for(i=0;i<=9;i++)

{

System.out.print(“Enter the mark of student ” + (i+1)+ “: ”);

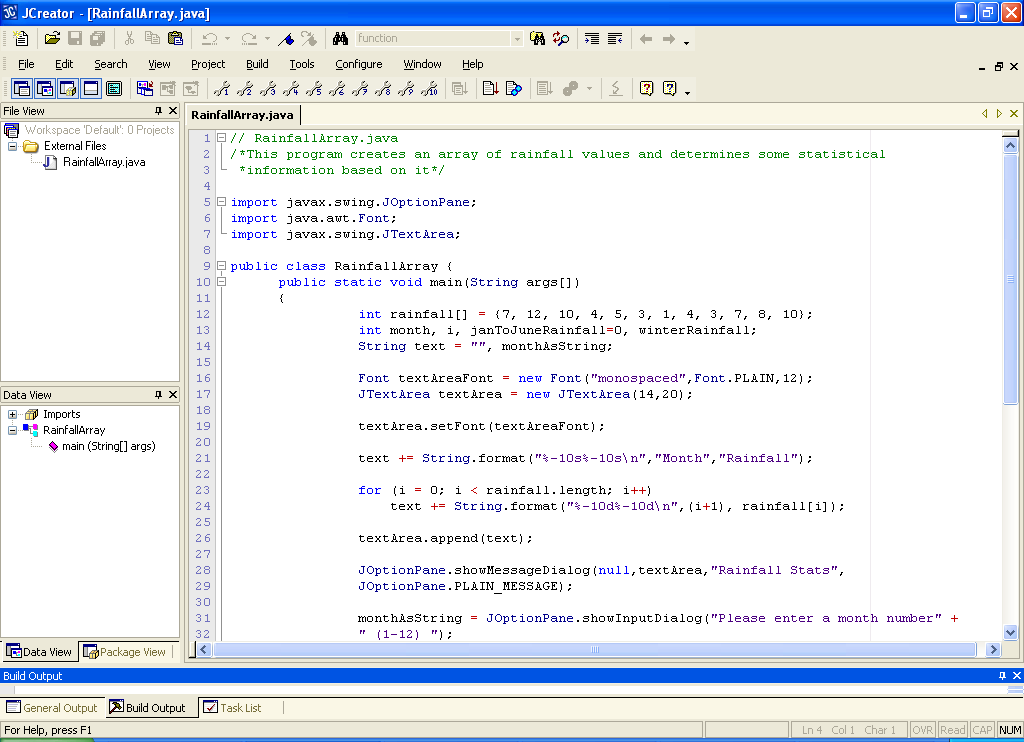
marks[i] = input.nextInt();

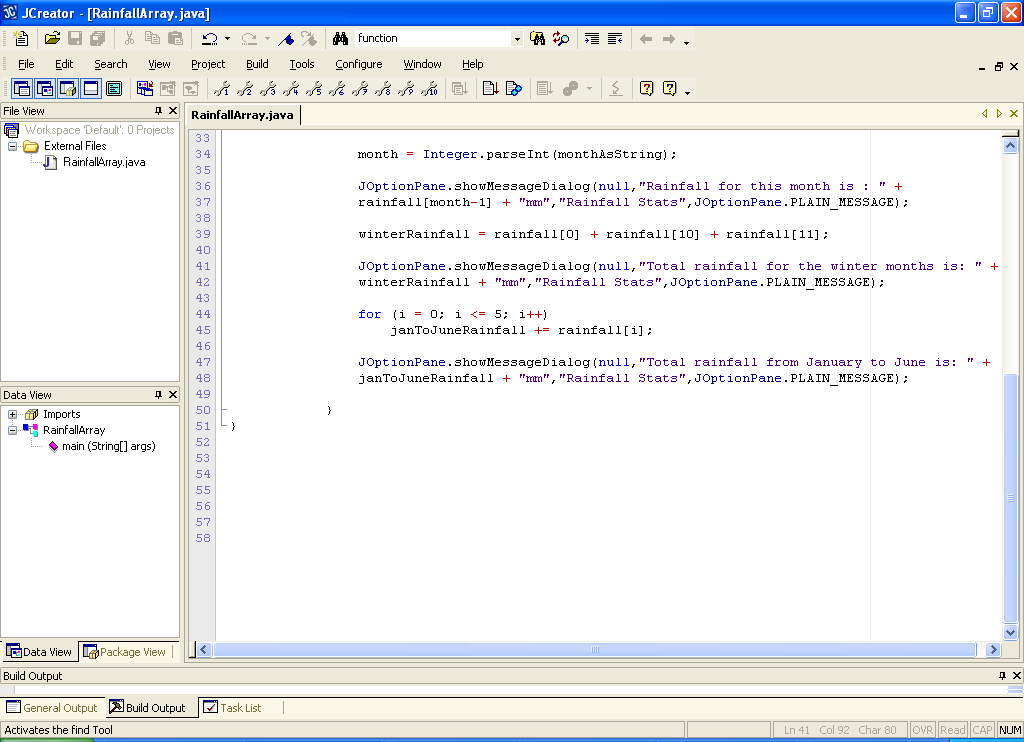
}

### A First Program using Arrays

**Aim:** We wish to write a program that will use an array of 12 integer values to store the rainfall values for the 12 months of the year.

**Java Code**:





**Analysis of Program**

• the code

int rainfall[] = {7, 12, 10, 4, 5, 3, 1, 4, 3, 7, 8, 10};

at the beginning of main() declares an array called rainfall and allocates 12 elements to it. Further to this, each block of the array is **initialised** with a particular value as follows:

Array name: rainfall

7 12 10 4 5 3 1 4 3 7 8 10

subscript number: 0 1 2 3 4 5 6 7 8 9 10 11

so we see that rainfall[0] has the value 7, rainfall[3] has the value 4 etc.

• some of the variable declarations come next as follows:

int month, i, janToJuneRainfall=0, winterRainfall;

here month is used to store the month value which the user will be prompted for,

i is just a loop counter, janToJuneRainfall is just used to store the rainfall total for the first six months of the year and winterRainfall is used to store the total rainfall during the winter months.

• the next section of code looks as follows:

Font textAreaFont = new Font("monospaced",Font.PLAIN,12);

JTextArea textArea = new JTextArea(14,20);

textArea.setFont(textAreaFont);

We have seen this kind of code before. The first line creates a Font object which will determine the type, style and point of font that will be used for the text area within the output dialog – **textAreaFont** is the **object reference** for this object.

The next line creates the **JTextArea** object. Recall that we use this GUI component because we can control the type of font used within it (something we cannot do with the message dialog). The numbers here refer to the number of rows and columns the text area can accommodate. These numbers don’t really matter though because the text area will “grow” to accommodate what we put into it.

The last line is the one that actually sets the font of the text area to that specified earlier. This is done through the object reference textAreaFont.

• the next line of code creates the heading for the table, which will be the first lump of information displayed by the program. Note that the operator += can be used with String variables to append information to the variable. String.format() is used to dictate the minimum space in the text area the text will take up (10 columns each in this case, left justified). This ensures that the table will be nicely aligned.

• the next code

for (i = 0; i < rainfall.length; i++)

text += String.format("%-10d%-10d\n",(i+1), rainfall[i]);

uses a for loop to iterate a total of **rainfall.length** times. When an array is created, its **size is stored in a field called length**. Therefore, when we wish to process an array in its entirety, we can use this length field to specify the upper limit value for the loop, as we are doing here. The length field is especially useful when arrays are passed into methods, as we shall see later. What is the value of rainfall.length?

Each time this loop iterates, a formatted string is appended to the end of the String variable (object reference) text. This string contains the values that will make up each line of the table. The (i+1) part will be the left column in the table and this is the month number (1-12). The rainfall[i] will be on the right column in the table and this is the actual rainfall values stored in the array.

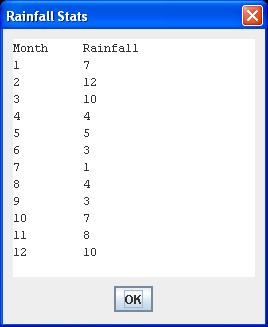
• the next section of code

textArea.append(text);

JOptionPane.showMessageDialog(null,textArea,"Rainfall Stats",

JOptionPane.PLAIN\_MESSAGE);

joins the string stored in the object reference text to the text area. Then the text area itself is placed onto the JOptionPane message dialog and when the program runs we see the table as indicated below:



• Once we hit return on the window above, the program then moves on again and next the user is asked to supply a month number between 1 and 12.

monthAsString = JOptionPane.showInputDialog("Please enter a month number" + " (1-12) ");

month = Integer.parseInt(monthAsString);

JOptionPane.showMessageDialog(null,"Rainfall for this month is : " +

rainfall[month-1] + "mm","Rainfall Stats",JOptionPane.PLAIN\_MESSAGE);

This is read in as a string, converted to an integer and then the rainfall figure for this particular month is displayed. Note that the code used is **rainfall[month – 1]**.

The reason for this is because the rainfall figure for month 1 is stored in the array element with subscript zero. This means we always must subtract 1 in order to get the correct rainfall figure. Failure to do this will result in either a **logical error** or, if we enter 12, a **runtime error**.

• the code

winterRainfall = rainfall[0] + rainfall[10] + rainfall[11];

JOptionPane.showMessageDialog(null,"Total rainfall for the winter months is: " + winterRainfall + "mm","Rainfall Stats", JOptionPane.PLAIN\_MESSAGE);

Just adds up the rainfall figures for January, November and December and displays the result.

• finally the code

for (i = 0; i <= 5; i++)

janToJuneRainfall += rainfall[i];

JOptionPane.showMessageDialog(null,"Total rainfall from January to June is: " + janToJuneRainfall + "mm", "Rainfall Stats", JOptionPane.PLAIN\_MESSAGE);

Contains a for loop that iterates 6 times. Each time it iterates the rainfall figure for a particular month from January to June is added to the counter variable janToJuneRainfall

Note the use of the arithmetic assignment operator for addition being used here.

The final total is then displayed.

**Organising your Work**

You should have a folder under X: called OOP1Stuff created. This time, create a folder called **Lab11** within OOP1Stuff to save your work from this lab session.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **RainfallArray.java** in your Lab11 folder. Now type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

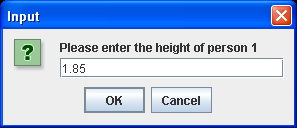
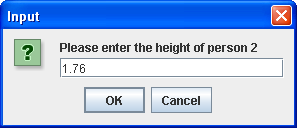
Once you are free from errors, run the program and test it fully. With proper testing you should see that this program is **not validated**

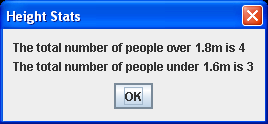
**Exercise 1**

The above program **lacks any security features** whatsoever in terms of **input validation**. Save the program as **Exercise1.java** and modify it so that when the user is asked to enter a value for the month, a check is performed to ensure the value entered is actually between 1 and 12 inclusive and will run a loop telling the user to re-enter the month value until a correct value is entered. Note here that you can **assume the user has entered a valid integer for the month value**, so there is no validation code required for this (this makes the validation job here relatively simple). Also alter the program so that it will also output the total spring rainfall.

**Exercise 2**

Write a program called **Exercise2.java** which uses an array called heights to store the heights of 10 people. Your program should not initialise the values of the array but set the values up dynamically at runtime instead by reading them in from the user. The program should then determine how many people were over 1.8m high and how many were below 1.6m and output these results. Your program should run as indicated in the following sample screenshots:

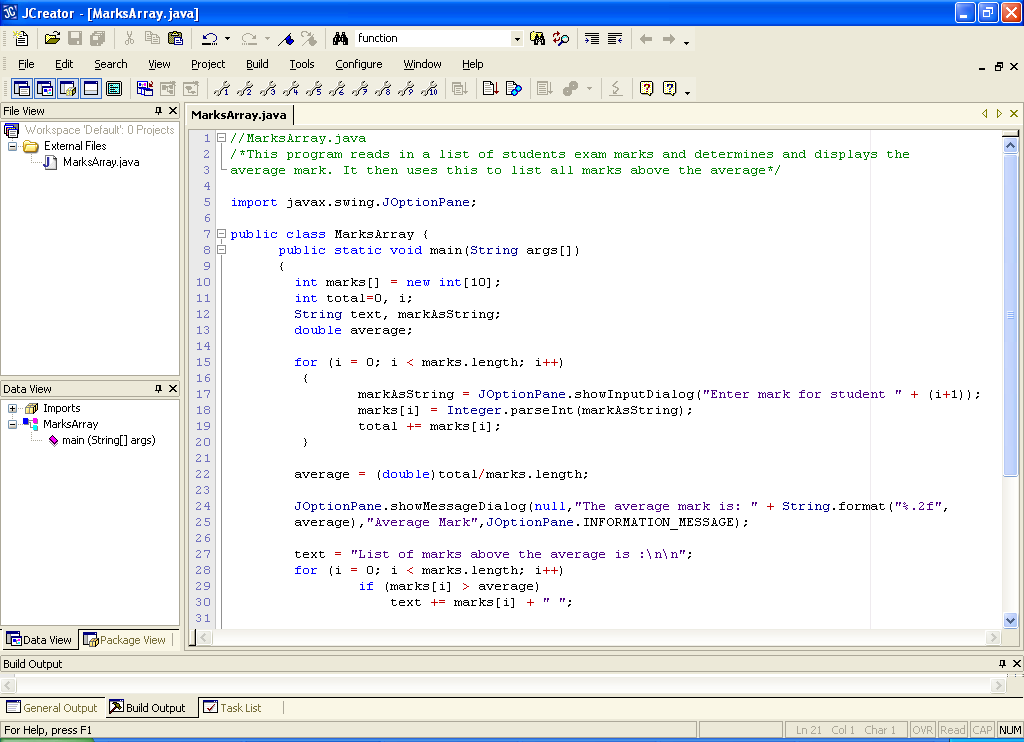
 

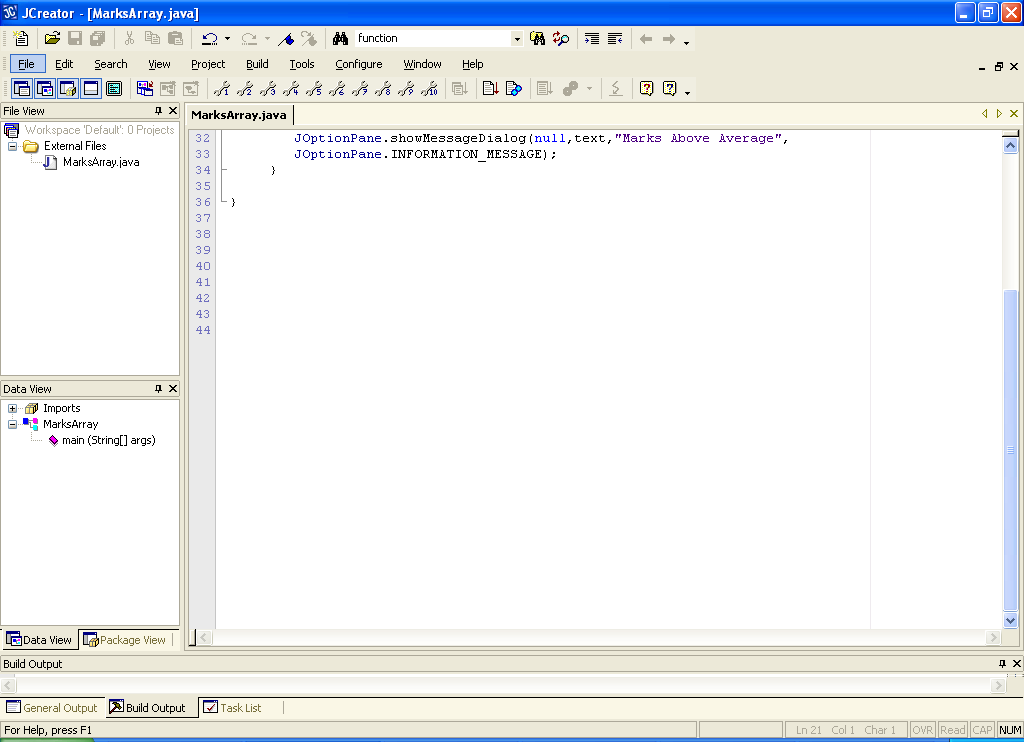


**Another Program on Arrays**

**Aim:** The next program reads in the examination marks of 10 students, determines and outputs the average mark and outputs all the marks above the average.

**Java Code:**





**Analysis of Program**

● The program begins by declaring an array of integers of size 10. The variable total is a counter that will keep track of the total of all the marks entered and i is just a loop counter.

● The first for loop iterates a total of marks.length i.e. 10 times and each time it iterates, the user is prompted for a mark. This is read in and converted to an integer. Then the value of variable total is updated by adding this mark to its current value.

● Once the first loop has completed we know that the value of total is equal to the sum of all the marks entered. We can now calculate the average mark. The code for this may seem a little strange:

average = (double)total/marks.length;

Notice that the variable total is being **typecast** here to become a double value for the purposes of this calculation. The reason for this is because both total and marks.length are integer values. You may recall from an earlier Java lab that **division of one integer by another integer in Java always results in an integer**! So, for example, if the value of total happened to be 788 and the value of marks.length is 10 then the value of

total/marks.length = 788/10 = 78 and not 78.8 as you might expect. Note that it isn’t even rounded up to 79 – simply put, everything after the decimal point is “chopped off”.

If you require the accurate average, the most appropriate course of action is to simply typecast either total or marks.length as a double just while this calculation is being performed. Now the calculation becomes a **double divided by an integer** and in Java this always results in a **double result**. I have chosen to typecast total here.

Not typecasting here would result in a **logical error** on most occasions.

● Once the average has been determined it is displayed to 2 decimal places using String.format().

Next there is another loop and this one also iterates 10 times. It goes through every element in the array and checks to see if the value stored in that element is greater than the average just calculated.

If it is, then the value is added onto the String variable (object reference) text for display purposes later.

● Finally, the list of marks above the average is displayed.

**Typing in Code for the Program Just Analysed**

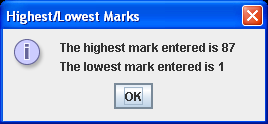
Click the **New File** icon on the JCreator IDE and save the file as **MarksArray.java** in your Lab11 folder. Now, for more practice with array syntax, type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and test it fully. With proper testing you should see that this program is **not validated**.

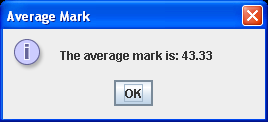
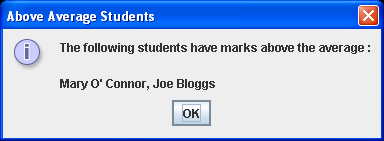
**Exercise 3**

You should now save the last program as **Exercise3.java** and modify it so that it displays the highest and lowest marks entered. We have covered this type of algorithm before so see if you can recall how to apply it here with arrays. Now, when the program runs, you should get an extra output window at the very end as follows:



**Exercise 4**

You should now save the **MarksArray.java** program as **Exercise4.java**. Use should now modify the program so that it creates an extra array called names whose type is String for storing student’s names. Naturally, it should have the same size as the existing marks array. When the program runs now, you will be prompted for both the mark of a student and their name also. The program still determines the average mark but this time, instead of displaying the list of marks above the average, the program should display the list of names for students who had a mark above the average. The final outputs now will look as follows:

**Exercise 5**

Write a console-based application called **Exercise5.java** that asks the user to input a list of 10 integers. Store these values in an array of the required size and output the 1st and 5th values in the array. Your program should also display the largest and smallest values in the array. Use a **for loop** to read in the values efficiently. Your output should run as indicated in the sample screenshot below.

