***Chapter 7***

**Introducing Arrays**

As we’ve learned, using variables makes your programs flexible. Thanks to variables, we can conveniently store data in our programs and retrieve the data later by name. Until now, you’ve learned about various types of numerical variables, including integer, long integers, floating-point, and double floating-point variables. Now --- a handy data structure called an array.

Often in your programs, you’ll want to store many values that are related in some way. Suppose you are writing a program that manages student grades. Assume a class has 10 students. One method of storing this information would be to declare 10 variables and assign scores into each separate variable.

int score1=70,

 score2=80,

score3=75,

score4=60,

score5=95,

score6=85,

score7=90,

score8=65,

 score9=55,

score10=90;

A subsequent reference to a score is cumbersome. For example, the code to display each score requires 10 separate System.out.println statements such as the following:

 System.out.println(“Student #1’s grade =” + score1);

 System.out.println(“Student#2’s grade =” + score2);

 System.out.println(“Student#3’s grade =” + score3);

 …

 System.out.println(“Student #10’s grade =” + score10);

A *one-dimensional array* can enable an efficient storage and retrieval of this kind of information. A *one-dimensional array* is a string of contiguous storage locations located in primary memory. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is:

 *type var-name[ ];*

The type determines the data type of each element that comprises the array. The statement:

 int score[ ];

defines a variable name that can point to a contiguous string of integers in primary memory. Initially, score is set to **null**, which represents an array with no value. To link score with a physical array of integers, you must allocate one using **new** and assign it to score. The **new** operator allocates memory.

 score = new int[10];

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| score[0] | score[1] | score[2] | score[3] | score[4] |
| score[5] | score[6] | score[7] | score[8] | score[9] |

The index (or subscript) begins with 0 and ends at 9.

Elements of the array can be initialized as follows:

score[0] = 70,

 score[1] = 80,

score[2] = 75,

score[3] = 60,

score[4] = 95,

score[5] = 85,

score[6] = 90,

score[7] = 65,

 score[8] = 55,

score[9] = 90;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| score[0] 70 | score[1] 80 | score[2] 75 | score[3] 60 | score[4] 95 |
| score[5] 85 | score[6] 90 | score[7] 65 | score[8] 55 | score[9] 90 |

A 2nd initialization technique:

int score[ ] = {70, 80, 75, 60, 95, 85, 90, 65, 55, 90};

In this case, array space is automatically created large enough to hold the number of elements specified between the curly braces. There is no need to use **new**.

*Array Processing*

For loops are most useful when processing an array. The following for statement computes the total of the 10 scores:

 int total = 0;

 for (int i = 0; i <= 9; total += score[i++]);

The above for statement performs the following actions:

 total = 0;

 total = total + score[0]; --- total=0+70

 total = total + score[1]; --- total=70+80

 total = total + score[2]; --- total=150+75

 total = total + score[3]; --- total=225+60

 total = total + score[4]; --- total=285+95

 total = total + score[5]; --- total=380+85

 total = total + score[6]; --- total=465+90

 total = total + score[7]; --- total=555+65

 total = total + score[8]; --- total=605+55

 total = total + score[9]; --- total=660+90

The following for will display the test scores:

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

 System.out.println(“Student #” + (i+1) + “ score =” + score[i]);

*Multidimensional Arrays*

In Java, multidimensional arrays are actually arrays of arrays. To declare a multidimensional array variable, you must specify each additional index using another set of square brackets:

 int multiplication\_table[ ] [ ] = new int [4][5];

The above statement allocates a multidimensional array in memory. Internally this matrix is implemented as an *array* of *array* of int. It can be visualized as follows:



The following code will build a 4 by 5 multiplication table using nested for loops:

 for (int x = 0; x < 4; x++) this statement is executed 20 times for (int y = 0; y < 5; y++)

 multiplication\_table[x][y] = x \* y;

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, the following code allocates memory for the first dimension of **multiplication\_table** when it is declared. It allocates the second dimension manually:

 int multiplication\_table [ ] [ ] = new int [4] [ ];

 multiplication\_table [0] = new int [5];

 multiplication\_table [1] = new int [5];

 multiplication\_table [2] = new int [5];

 multiplication\_table [3] = new int [5];

While there is no advantage to individually allocating the 2nd dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. Example:

 int multiplication\_table [ ] [ ] = new int [4] [ ];

 multiplication\_table [0] = new int [1];

 multiplication\_table [1] = new int [2];

 multiplication\_table [2] = new int [3];

 multiplication\_table [3] = new int [4];

 *A Note to C / C + + Programmers About Pointers*

If you are an experienced C / C + + programmer, then you know that these languages provide support for pointers. However, no mention of pointers has been made. The reason for this is simple: Java does not support or allow pointers. (Or more properly, Java does not support pointers that can be accessed and / or modified by the programmer.) Java cannot allow pointers, because doing so would allow Java applets to breach the firewall between the Java execution environment and the host computer. (Remember, a pointer can be given any address in memory --- even addresses that might be outside the java run-time system.) Since C / C + + makes extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the confines or the execution environment, you will never need to use a pointer, nor would there be any benefit is using one.

*A Closer Look at Argument Passing*

There are two ways that a computer language can pass an argument (or parameter) to a subroutine. The first way is *call-by-value*. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument used to call it.

The 2nd way an argument can be passed is *call-by-reference*. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

 public class mystery {

 public void meth (int x, int y)

{

 x \*= 2;

 y /= 2;

 }

 public static void main( String args[ ] )

 {

int a = 10, b = 20;

meth(a, b);

*question: a=?\_\_\_\_ b=?\_\_\_\_\_*

 }

 }

public class mystery {

 public void square ( int x )

{

 x = x \* x;

 }

 public static void main( String args[ ] )

 {

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

*question: a[4]=?\_\_\_\_*

square ( a[4] );

*question: a[4]=?\_\_\_\_*

 }

 }

*Call by Reference*

The situation changes dramatically when you pass an array to a method. This is because arrays are passed by reference. Keep in mind that when you create an array, you are only creating a reference to the array. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same array as that referred to by the argument. This effectively means that arrays are passed to methods by use of call-by-reference. Changes to the array inside the method do affect the array used as an argument. For example:

public class mystery {

 public void square\_element (int x[ ], int element)

{

 x[element] = x[element] \* x[element];

 }

 public static void main( String args[ ] )

 {

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

*question: a[4]=?\_\_\_\_*

square\_element ( a, 4 );

*question: a[4]=?\_\_\_\_*

 }

 }

This could be x.length

public class mystery {

 public void zero (int x[ ], int length)

{

 for (int index=0; index<length; index++)

 x[index] = 0;

 }

 public static void main( String args[ ] )

 {

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

*question: a[4]=?\_\_\_\_*

zero ( a, 5 );

*question: a[4]=?\_\_\_\_*

 }

 }

**Assignment for Chapter 7**

Use a single subscripted array to solve the following problem. Write a program that reads an unspecified number of test scores (from 0 to 100%) in random order. Your program should display how many scores are above or equal to the average and how many scores are below the average. Do all your calculations in integer form (i.e., truncate the average). Enter a negative number to signify the end of the input. Assume that the maximum number of scores is 10.

Your program must do the following:

1. When a negative number is entered, or after the 10th value is entered, you must:
	1. make the textField no longer editable (you may do this by calling the textField’s setEditable method)
	2. display a message in the status bar indicating that no more data can be entered
	3. display 3 lines of output:
		1. The class average is \_\_\_\_\_
		2. The # of scores greater than or equal to the average is \_\_\_\_
		3. The # of scores less than the average is \_\_\_\_\_

2. Error check the data entered verifying that scores do not exceed 100% (displaying an appropriate error message in the status bar). Scores over 100% should not be considered in the computations of your program.

Your **paint** routine should display the three lines of summary at the end, only (i.e., only after the user has either entered a negative score or a tenth valid score).

Your program must have the three methods listed below, each of which should receive as an input parameter an array of integers, and an integer indicating the number of valid entries contained in the array, and then return an integer result:

1. **computeAverage**
2. **aboveAverage**
3. **belowAverage**

These methods should be called by **paint** when displaying your 3 line summary of results. A correct method heading for computeAverage would look like this:

 **int computeAverage(int x[ ], int length)**

Be sure to test all cases, and **submit sufficient screen captures** to convince me that your program works. Insufficient screen captures will lead to a lower grade on this project.

Note: Use the showStatus method to display messages on the status bar.

 **showStatus(“message”);**

Extra Credit: After all the scores have been entered (in RANDOM order) display the scores in ascending order below the 3 lines of summary output.