Arrays in Java

So far we have talked about variables as a storage location for a single value of a particular data type. We can also define a variable in such a way that it can store multiple values. Such a variable is called a **data structure.** An **array** is a common data structure used to store a collection of values of the same type, where the collection's size does not change once it is declared.

An array is a list of data items represented by a single variable name. It can be thought of as a collection of variables all of which are referenced using that single name. Each value in the array is called an element and individual elements are accessed using an index which is an integer enclosed in [ ]. The index specifies the position of a particular element in the array.

Declaring arrays

In defining an array in Java we have to indicate that it is an array and also specify the data type of the elements that will be stored there. For example if we wanted to define an array called grades that contained 5 elements of type double, we could write the following:

```
final int NUM_GRADES = 5;
double [] grades = new double [NUM_GRADES];
```

The word `double` indicates the data type of the elements in the array, the `[]` indicates that it will be an array, and `grades` is the name of the array.
The expression to the right of the = operator instructs the compiler to allocate an array capable of storing `5` (the value of the constant `NUM_GRADES`) double values.

NOTE: An array is a reference data type. The memory location of the array is an address to the starting location of the array itself.

Array Initialization

An array's items are automatically initialized to the default value of their type. For the numeric types the default values are 0.0 for floats and 0 for integers. For reference types the default value is null. Note that normal variables are not automatically initialized. This default initialization is for arrays only.

Arrays can also be initialized using a special `{ }` notation. This notation can only be used at the time an array is being declared. It cannot be used for regular assignment statements. In the following example, a new `int[]` will be created and its size will be 7 (i.e., the number of values specified in brackets).

```
int[] scores = {0, 1, 2, 3, 4, 5, 6 };
```

Accessing Array Elements

An individual element of an array can be accessed by specifying the name of the array, followed by the element's position in the array (it's index) which is enclosed in brackets. Like characters

in a string, the first element of the array has an index of 0. An array element can be thought of as a variable whose type is the array's item-type.

The array subscript operation validates each access to an array. If the subscript is outside the range of valid index values things can go wrong!!!!

The array **length** property contains the number of elements in an array. It is found by giving the name of the array followed by a dot and the word length (e.g., `grades.length`). Note that length is a property, not a method… so there are no `( )` after the word length as there is for a String object's `length()` method.

Since the index of an array begins with 0, the index of the last element is always the array's length - 1. Thus the expression `anArray[anArray.length - 1]` can be used to access the last element of the array.

Calling Methods with Arrays

Arrays can be passed as parameters but again there has to be an indication that the parameter is an array. If we were to pass grades as a parameter, the receiving method needs to define a parameter capable of storing the array's handle. NOTE: When an array is passed as a parameter it really only passes the address of the starting location of the array (the handle). This is done by placing a pair of brackets between a parameter's type and its name.

For example, if we had a method with an array parameter called anArray we could define it as follows:

```
public static double average(double[] anArray)
```

Processing Array Elements

A for loop is ideal for accessing each of the elements of an array.

For example, to get grade information into an array, we could write the following:

```
for (int i = 0; i < array.length; i++) {
    System.out.print("Enter grade #" + (i+1)+ " of "
                     + array.length + ": ");
    array[i] = keyboard.nextDouble();
}
```

A complete program for entering grades, determining their average, and printing them and the average is as follows:

```
/* This program will read in grades, find the average and print out
 * the average of those grades
 * @author Carter
 *
 */
import java.util.Scanner;
```

```
public class Grades {
      public static void main(String[] args) {
            final int NUM_GRADES = 5;
            double[] grades = new double[NUM_GRADES];
            readGradesInto(grades); // this is a call to a method that
                                    // passes an array parameter
            System.out.printf( "\nThe average is %.2f%n", average(grades));
            print(grades);
      }

      private static void readGradesInto(double[] array) {
            Scanner keyboard = new Scanner (System.in);
            System.out.println("To compute the grade ");
            for (int i = 0; i < array.length; i++) {
                System.out.print("Enter grade #" + (i+1)+ " of " +
                          array.length + ": ");
                array[i] = keyboard.nextDouble();
            }// end of for loop
      }

      public static double average(double[] array) {
            double sum = 0.0;
            for (int i = 0; i < array.length; i++) {
                sum = sum + array[i];
            } // end of for loop
         return (sum / array.length);
      }

      public static void print(double[] array){
            for (int i = 0; i < array.length; i++) {
                System.out.println("Grade #" + (i+1) + ": " + array[i]);
            }
      }
}
```

The for each loop

Java has a special version of the for loop called a **for-each loop**. In finding the average of grades we could write an average() method that calculates the sum of all grades as follows:

```
double sum =0.0;
for (double item : anArray) {
    sum = sum + item;
}
return sum / anArray.length;
```

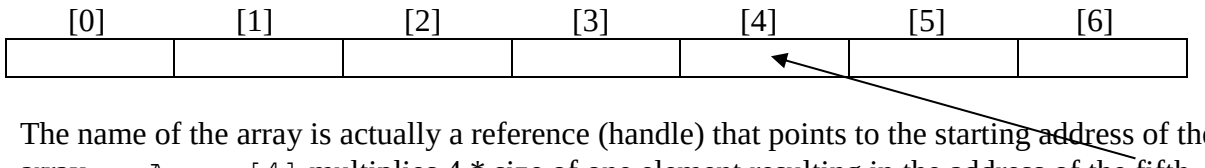
This repeats the statement `sum = sum + item;` for each item in the array.

NOTE: the for each loop can be used to *read* an array's items, but cannot be used to *write* into them. For example we could **not** use a for each loop for `readGradesInto()` above because that reads a number from the keyboard and writes it to an array element. However, we could use it for `average()` and `print()` since in those methods we are only reading an array's items.

Arrays and Memory

An array's elements reside in adjacent memory locations and Java's subscript operator makes use of this adjacency to access any element of an array in the same amount of time.

For example, when Java evaluates the expression `anArray[i]`, it multiplies the index `i` by the size of one item (the amount of storage space that the data type uses) and adds the resulting product to the reference location specified in `anArray`.



The name of the array is actually a reference (handle) that points to the starting address of the array. `anArray[4]` multiplies 4 * size of one element resulting in the address of the fifth element of the array.

An array is a random access data structure which means it takes the same amount of time to access any item in the. Arrays are not good, however, if we want to change the contents of the array by inserting an element in between existing elements.

It is not uncommon for some positions in an array to be unused. For example, if we were reading an undetermined number of grades (with a maximum of 30) we would make our array size 30, but keep track of how many elements are actually used (i.e., read in). We could then pass that number as an additional parameter to any method that uses the array. The size of our array for practical purposes is not the actual size as determined by the length property, but rather the number of elements in the array that are actually used. The grades program would now be as follows.

```
import java.util.Scanner;
public class GradesUndeterminedAmount {

	public static void main(String[] args) {
		final int NUM_GRADES = 30;
		double [] grades = new double[NUM_GRADES];
		int sizeOfArray;  // The actual amount of grades read in
		sizeOfArray = readGradesInto(grades);
		System.out.printf( "\nThe average is %.2f%n",
				average(grades, sizeOfArray));
		print(grades, sizeOfArray);
	}

	public static int readGradesInto(double[] array) {
		Scanner keyboard = new Scanner (System.in);
		int arraySize = 0;
		double grade;
		System.out.print("Enter grade #" + (arraySize + 1) +
				" enter -1 to quit");
		grade=keyboard.nextDouble();
		while (grade >= 0) {
			array[arraySize] = grade;
			arraySize++;  // increase size by 1
```

```
                System.out.print("Enter grade #" + (arraySize+1)+
                            " enter -1 to quit");
                grade=keyboard.nextDouble();
            }
            return arraySize;
        }

        public static double average(double [] anArray, int arraySize) {

            double sum = 0.0;
            for (int i = 0; i < arraySize; i++) {
                sum = sum + anArray[i];
            }
            return sum / arraySize;
        }

        public static void print(double[] arr, int arraySize){
            for (int i = 0; i < arraySize; i++) {
                System.out.println("Grade #" + (i+1) + ": " + arr[i]);
            }
        }
}
```

Multidimensional Arrays

Thus far we have only considered arrays that have a single dimension: length. That length determined the amount of space needed by the array. They are referred to as 1-dimensional arrays. We can also declare multiple dimensional arrays. For example, a 2-dimensional array could be used to store data in a table. A 2-dimensional array can be declared as follows:

```
double [][] myTable = null;
```

We use two pairs of brackets instead of one to indicate that this is a two-dimensional array. In general we would use N brackets for an N-dimensional array.

We can allocate memory for a multi-dimensional array though the use of the **new** operator. myTable = new double[rows][columns]; Where rows represents the number of rows in our table and columns represents the number of columns. We can also initialize the positions in a multi-dimensional (in this case two) array as follows:

```
double[][] anArray = { {0, 1, 2, 3, 4, 5, 6 },
                       {7, 8, 9, 10, 11, 12, 13}
                       {14, 15, 16, 17, 18, 19, 20} };
```

This creates a two-dimensional array with 3 rows and 7 columns.

To access elements in a one-dimensional array we used one subscript operator. To access an element in a 2-dimensional array, we use two subscript operators.

```
return myTable[row][col]; // access element at row, col.
```