CSC142 Supplemental Reading: Week06

Contents

1 Arrays

An **array** is a group of values of the same data type. Arrays are the most fundamental way to group together data. In Java, an array of any data type may be created. The "strongly typed" nature of Java extends to arrays: a "double array" may hold only values of type *double*.

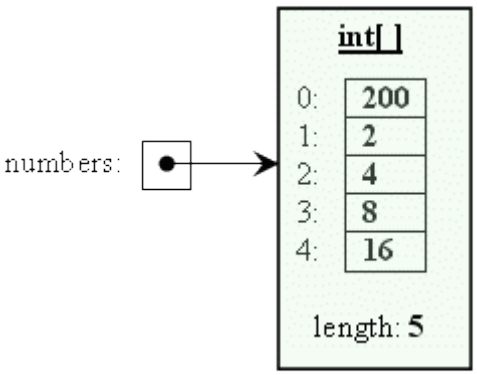
Here is a short bit of code that declares, creates and uses an array:

```
int[ ] numbers = new int[5];
System.out.println( "initial: " + numbers[2] );
for( int i = 0; i < numbers.length; i++ )
    numbers[i] = (int)( Math.pow( 2, i ) );

numbers[0] = 200;
for( int i = 0; i < numbers.length; i++ )
    System.out.println( "\n" + numbers[i] );


/* Printed to the console:
initial: 0
200
2
4
8
16
*/
```

Object Diagram:



Notice several things about this program:

- The **data type** of the array variable **numbers** is **int[ ]**, which we would read as "**int array**".
- **An array is an object** -- the keyword ***new*** must be used to **construct an Array object** before it can be used.
- The syntax for constructing an array object is unique, and includes the **length** of the array (the number of values it can hold) -- also called its **capacity**. The length of an array must be specified when the object is constructed; **the length can never be changed**.
- Each value stored in an array is called an **element** (or an array element).
- When an array is constructed, the value of every element is automatically **initialized**. The automatic initialization rules are: 0 for numbers, *null* for object references, *false* for type *boolean*, and '\0' for type *char*.
- Each array element is identified by an **index number**. The first index number is zero, and they count up from there. The index number is sometimes also called an array **subscript**. Index numbers always begin at the number 0 and end at one less than the length of the array. In this example, the length is 5 and the index numbers are 0 through 4.
- An individual **element** of an array may be accessed by using the name of the array variable followed by square brackets [ ] containing the index number of the element. The expression: **numbers[2]** can be read "numbers **sub** 2", meaning "the element in the array numbers at index 2".
- The expression used inside the square brackets may be any expression that evaluates to a legal index. It may be a constant (e.g. numbers[2]), a variable (e.g. numbers[i]), or any expression of type *int* (e.g. numbers[i + 2]). Subscripts are type *int* for all arrays, not just *int* arrays. If the index value provided is not legal, an ArrayIndexOutOfBoundsException is thrown.

- An individual element in this array (like numbers[2]) has data type ***int*** and may be used just like any other ***int*** variable. You can use it in any expression (as in the second and last lines of the program); you can pass it as an argument to a method; and you can assign a value to it (the 4th and 5th lines of the program).
- One of the more confusing points about arrays is this: **don't confuse the index number with the value stored in the array**. For example, numbers[4] contains the value 16. 4 is the index number; 16 is the contents of (or value of) the element.
- The array object has one property called '**length**' that holds the number of elements in the array. The array in this example has a length of 5 -- it has 5 elements, numbered 0 through 4.
- The array object has not other methods or properties. Any algorithm you want to perform on an array (such as print it, sum it, find the largest value, etc) must be programmed by you.
- The loops shown illustrate a common pattern of iterating through every array element. A simple counting loop is used, and the loop counter is used as the index. Many variations on this pattern are possible, but this simple version that iterates through **all** elements is the most common. Notice how the **length property** of the array is used in the loop test with the < operator.

Let's restate the information about data types:

- The array variable **numbers** has data type **int[ ]**.
- The element **numbers[i]** has data type ***int***.

1.1 Declaring Array Variables and Constructing Array Objects

Declaring an array variable creates the reference variable, but **not** the array. This statement:

```
double[ ] profits;
```

declares a variable named **profits**. It is a reference variable with no reference.

This statement:

```
profits = new double[6];
```

constructs a new double[ ] object with length 6 and binds a reference to the variable **profits**. This array has 6 elements, numbered 0 to 5, and all are initialized to the value 0.0.

This statement:

```
double[ ] x = profits;
```

declares a new variable, but it does **not** create a new array. Instead it creates an **alias** -- a second variable referring to the same array object.

It is possible to create a new array object and at the same time initialize its contents. This *can only be done* in the same statement where the array variable is declared, like this:

```
double[ ] rates = { 2.75, 3.25, 3.5, 3.675 };
```

This statement not only declares a new variable (**rates**), but also constructs a new double[ ] object initialized with the values provided. The length of the array is implied by the number of values listed in the { } -- it is 4 in this example. Notice that there is no keyword ***new*** in this statement. Nevertheless, this statement **does** create a new array object.

Be aware that an array with length 0 is not the same as no array at all. These two statements:

```
int[ ] a1;
int[ ] a2 = new int[0];
```

produce entirely different situations. For example, **a2.length** is a legal expression (its value is 0), but **a1.length** will throw an exception.

1.2 Arrays of Objects

An array may be formed of any data type. If the data type is a reference type, then the array holds object references only -- not the actual objects -- even though we might refer to it as an array of objects. Here's an example:

```
String[ ] names = new String[12];
names[3] = "Tai";
String[ ] simpsons = { "Marge", "Homer", "Lisa", "Bart", "Maggie" };
Color[ ] favColors = { Color.BLUE, new Color( 255, 102, 255 ), null };
myBox.setColor( favColors[1] );
```

Notice one detail in this example: the array **favColors** has been forced to have length 3, but the last element has been set to ***null***. This leaves a place in the array to add another color later.

Think about the data types in this last example. Fill in the following table:

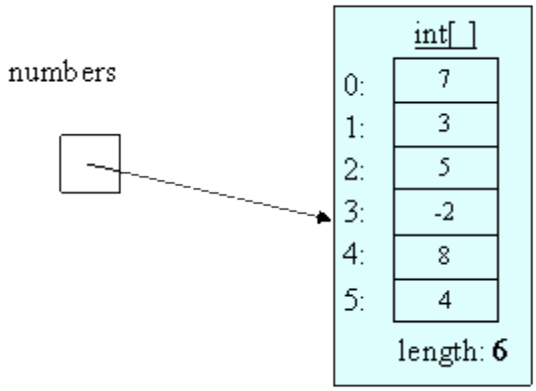
variable	data type
names	
names[3]	
favColors	
favColors[1]	

1.3 Array Indexes

As we saw above, array elements can be accessed by using a literal number as the index. Yet, an array usually holds a collection of related information, so it is common to process this data with a loop. In this logic, the loop control variable is used to identify the particular element to access. Here is a sample program that shows all of the elements in the array numbers in the terminal window by using a loop to count through the index values:

```
int[ ] numbers = { 7, 3, 5, -2, 8, 4 };
for ( int ix = 0; ix < numbers.length; ix++ )
       System.out.println( numbers[ ix ] );

/* Test results:
7
3
5
-2
8
4
*/
```



Notice several things about this program:

- In this counting loop, the variable 'ix' is the loop counter. Convince yourself that as the loop executes, this variable will start at 0, then go to 1, then to 2, 3, 4, 5, and stop when it reaches 6.
- The value of ix is **initialized to '0'** -- the lowest array index.
- The loop continues as long as the value of 'ix' is **less than the length** of the array. In an array, the index values range from 0 up to one less than the length of the array. The array in this example has a length of 6 -- it has 6 elements, numbered 0 through 5.
- The loop counter 'ix' is used as the array subscript: 'numbers[ix]'. With each pass through the loop, the value of 'ix' changes, and a different element of the array is shown on the web page.
- In this example, there are only 6 elements in the array, but this logic would work for any size array.

How would you rewrite this loop so that it printed the contents of the array in reverse order? That is left as a practice activity.

1.3.1.1 *Arrays as Method Parameters and Return Values*

An array type may be specified as the data type for a formal parameter or a return value in a method definition. In these situations, the array is treated like any other reference type -- just remain aware of that the data type of each variable in the program. Here's an example:

```
public static void testVectorSum() {
   double[ ] x = { 1.3, 4.5, 2.0 };
   double[ ] y = { 2.7, 1.5, 3.0 };
   double[ ] z;
   z = vectorSum( x, y );
}
public static double[ ] vectorSum( double[ ] a, double[ ] b ) {
   if ( a.length != b.length )
      throw new IllegalArgumentException();
   double[ ] sum = new double[a.length];
   for ( int i = 0; i < a.length; i++ )
      sum[i] = a[i] + b[i];
   return sum;
}
```

Here are some things to notice in this example:

- The data type of each parameter in the vectorSum method is **double[ ]**.
- This data type says nothing about the lengths of the arrays. Arrays of any length might be used as the actual parameters of a method call. If needed (as in this case), a validation test should be included.
- The variables **x** and **y** used as the arguments in the method call are type **double[ ]**. This matches the data type expected. Notice that **no [ ] are used in the method call**. **x** is type **double[ ]**, but **x[0]** would be type ***double***, and x[ ] would be a syntax error.
- The return type for the vectorSum method is **double[ ]**. The local variable **sum**, used in the return statement, also has this data type.

1.4 Two-Dimensional Arrays

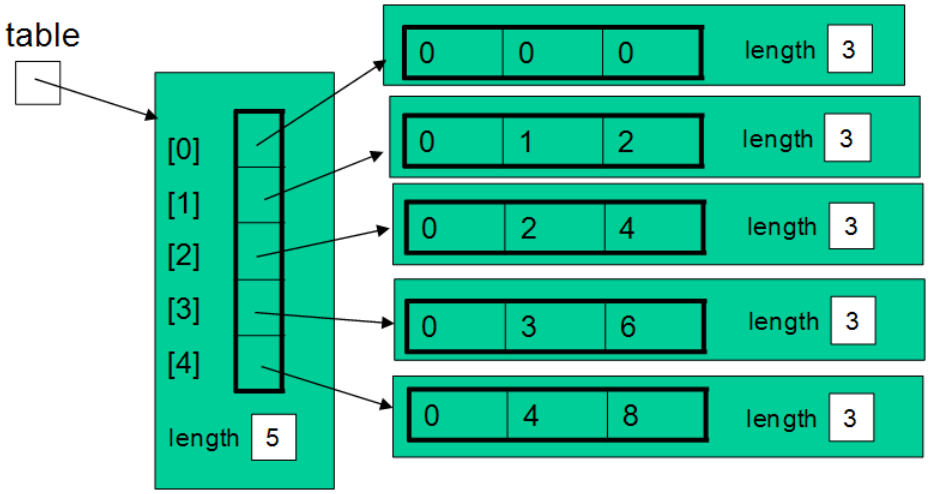
When you consider of a table of information, you can think of it of as a group of rows, with each row being a group of cells. This analogy is a good way to think of 2-D arrays in Java. There is really no such thing as a 2-D array specifically. Rather, we take advantage of the fact that **any data type can be used as the basis for an array -- even another array type**. So, a 2-D array of integers is really an array of *int* arrays. Here's an example:

```
int[ ][ ] table = new int[5][3]; // 5 rows, 3 cells in each row (i.e. 3 columns)
// fill in values
for ( int r = 0; r < table.length; r++ )
   for ( int c = 0; c < table[0].length; c++ )
      table[r][c] = r * c;
```

Notice the following:

- The data type of **table** is **int[ ][ ]** -- "int array array"
- The data type of **table[0]** is **int[ ]** -- "int array"
- The data type of **table[r][c]** is ***int***
- The value of **table.length** is 5 -- the is the length of the array of *int* arrays
- The value of **table[0].length** is 3 -- the is the length of each *int* arrays
- Each subscript in a 2-D (or 3-D or ...) array has its own set of [ ], and they are listed in sequence. This emphasizes the 'array of arrays' structure.

An object diagram is very helpful in understanding what is going on:

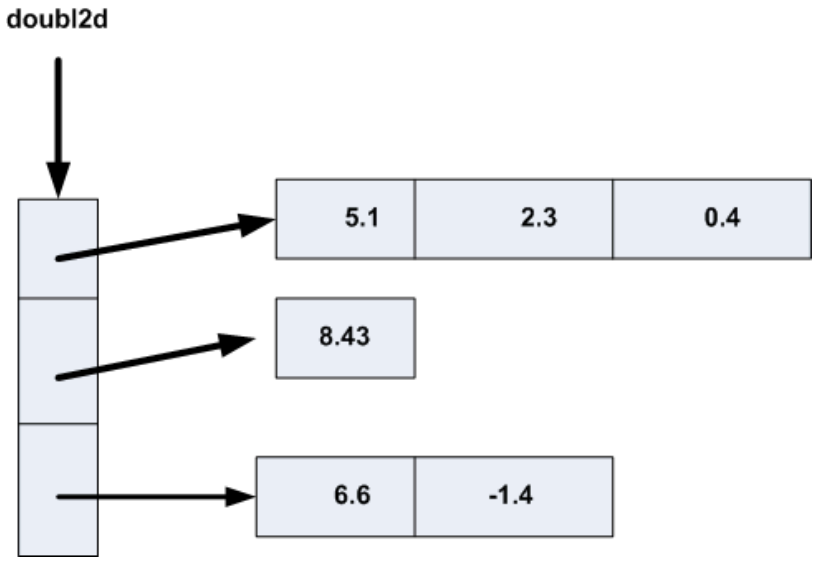


Array initialization is also possible with a 2-D array. Here's an example:

```
double[ ][ ] doubl2d = { {5.1, 2.3, 0.4}, {8.43}, {6.6, -1.4} };
String[ ][ ] families = {
          {"Marge", "Homer", "Lisa", "Bart", "Maggie"},
          {"Archie", "Edith", "Gloria"},
          {"Mike", "Carol", "Greg", "Marcia", "Jan", "Peter", "Bobby", "Cindy"}
};
```

One interesting thing to notice about these examples: the lengths of the 'sub-arrays' are not the same. Indeed, there is no requirement that they be the same, and array initialization provides a way to create a **jagged array** (a 2-D array where the sub-arrays have different lengths) like this. The syntax for array initialization also reflects the 'array of arrays' structure.

Here is an object diagram for the doubl2d array (I have omitted the length field for clarity):



2 Array Algorithms

Maintaining a collection of related data makes it especially easy to perform analytical operations on that data. This note describes some of the most basic and most common operations. You should master all of these techniques -- these are tools you need to have at your fingertips!

The algorithms in this note are demonstrated using arrays, but other types of collections could be used with only minor modifications. Focus on the fundamental principles of each algorithm as much as on the array-specific syntax.

2.1 Basic Iteration

Iterating through a collection -- examining each element in sequence -- is the most basic operation and is at the core of many algorithms. The following short bit of code illustrates the use of a counting loop to vary the array index over the full range of valid values.

```
int[ ] grades = { 95, 88, 99, 74, Math.round(77.8f), 65, 96 };
for ( int i = 0; i < grades.length; i++ ) {
   System.out.println( grades[i] );
}
```

Notice the pattern used in the *for* loop. This is a pattern you should memorize. The counting loop begins with the number 0 and counts as long as the count is less than the length of the array. Because this pattern uses the array length property rather than a magic number, it will work with an array of any length. This example also illustrates array initialization.

2.2 Sums and Averages

Adding things up or finding the average value are related operations. Both require a complete pass through the collection, 'touching' every element. Here is a method that takes an int array as a parameter and returns the average of all the values in the array, rounded to the nearest integer value:

```
public static int averageInts( int[ ] x ) // assumes length > 0
{
   int sum = 0;
   for ( int i = 0; i < x.length; i++ ) {
      sum += x[i];
   }
   return Math.round( sum / (double)x.length );
}
```

Notice the following:

- This method is declared ***static***. When can you do that? The simple rule is that any method that does not make use of instance variables may be declared *static*, and there is really no reason not to do so.
- Notice the data type of the parameter: **int[ ]** This means that any *int* array may be provided, and it says nothing about the length of the array. You always need to check the length of an array parameter -- you cannot make any assumptions about what it will be.
- The accumulator **sum** must be **initialized** before the beginning of the loop.
- The standard pattern for iterating through all elements in an array is used.

2.3 Lowest or Highest Value (or its Index)

Another very common algorithm determines the lowest (or highest) value in a collection. Here's an example:

```
public static int findLowestInt( int[ ] x ) // assumes length > 0
{
   int lowest = x[0]; //initialization -- assume first element is lowest
   //notice initialization on next line
   for ( int i = 1; i < x.length; i++ ) {
      if ( x[i] < lowest )
      lowest = x[i];
   }
   return lowest;
}
```

Notice the following:

- This algorithm also requires an accumulator, and like all accumulators, it must be initialized. However, the initialization here can be a little bit trickier. If you know that the array provided will always contain at least one element, you may initialize to the value of the first element, as has been done here. Otherwise, you need to use another strategy.
- The counting loop skips the first element of the array, since this has been handled with the initialization.
- An *if* statement inside the body of the loop determines whether or not the accumulator should be updated.
- Apart from changing the names of identifiers so they are still meaningful, only one character in this program needs to be changed to make it find the highest value instead of the lowest. What is it?

Many variations on any of these algorithms are possible. Suppose you wanted to find the element in an array of String objects that contains the fewest characters. Or perhaps you want to find the element in a String array that begins with the "highest" character (i.e. the letter closest to the end of the alphabet if the String values are all words). How would you adapt the algorithms to solve these problems? Why not try it and post your solutions in the forum!

One very important variation involves finding the **index** of the lowest (or highest) value instead of the value itself. This is especially important when working with an array of objects, as we will soon learn. By finding the index rather than the value itself, it becomes possible to retrieve a reference to the object itself, which can then be used in any way desired. We will soon discover that finding the lowest/highest index is also an important part of some sorting algorithms.

Here is the method above modified to return the index of the lowest element:

```
public static int findLowestIntIndex( int[ ] x ) // assumes length > 0
{
   int lowestIndex = 0; //initialization -- assume first element is lowest
   //notice initialization on next line
   for ( int i = 1; i < x.length; i++ ) {
      if ( x[i] < x[lowestIndex] )
      lowestIndex = i;
   }
   return lowestIndex;
}
```

Make a point of noticing the several ways in which this method differs from the previous version.

2.4 Sequential Search

Another common operation is a simple, or sequential, search. Here's an example:

```
public static boolean findIntValue( int[ ] x, int target )
{
      for ( int i = 0; i < x.length; i++ ) {
         if ( x[i] == target )
            return true; // we don't need to search any more
      }
      return false; // we only get here if search failed
}
```

Notice the following:

- No extra initialization required; standard iteration loop used.
- The method terminates (with a return statement) as soon as a match for the target is found -- there is no need to continue searching.
- If the loop finishes normally, the search must have failed.
- A variation on this might return the index of the first element that matches the target, or some special code (frequently -1) if the target is not present. What would you need to change to make this happen? Give it a try!

2.5 Counting Things

A few changes to this search method yields a method that counts the occurrences of the target:

```
public static int findIntValueCount( int[ ] x, int target )
{
   int count = 0; // initialize the count
   for ( int i = 0; i < x.length; i++ ) {
      if ( x[i] == target )
         count++; // found one -- count it
   }
   return count;
}
```

Notice that no early exit is possible. We need to examine every element in order to ensure that the count is complete.

2.6 Sifting

Another variation on a simple search is a sifting operation. The key difference is that a sifting operation can find more than one match -- its job is to find all values that match the specified criteria. Here's an example:

```
/**
* This method takes an array of ints and a threshold value.
* Prints to system.out the values of all elements from
* the array that are greater than or equal to the threshold.
*/
public static void printHighValues( int[ ] x, int threshold )
{
   System.out.print( "Values >= " + threshold + ": " );
   for ( int i = 0; i < x.length; i++ ) {
      if ( x[i] >= threshold )
         System.out.print( x[i] + " " );
   }
   System.out.println(); // go to next line
}
```

As with counting, no early exit is possible. The standard pattern for iterating through an entire array is used.

2.7 Methods that Return an Array

Many algorithms that operate on collections produce a new collection as a result. In these cases, the method needs to return an array as its result. Here's an example:

```
/**
* This method takes 2 double array parameters and returns
* a double array result. Each element of the result is the
* sum of the corresponding elements of the parameters. If
* the parameter arrays are of different lengths, the result
* has the length of the longer of the two.
*/
public static double[ ] addArrays ( double[ ] x, double[ ] y ) {

   int size = Math.max( x.length, y.length );
   int smaller = Math.min( x.length, y.length );
   double[ ] result = new double[size];

   double[ ] longer;
   if ( x.length > y.length ) {
      longer = x;
   } else {
      longer = y;
   }

   for ( int i = 0; i < smaller; i++ ) {
      result[i] = x[i] + y[i];
   }
   for ( int i = smaller; i < size; i++ ) {
      result[i] = longer[i];
   }
   return result;
}
```

Notice the following:

- The method's return type is **double[ ]** -- it returns an array. Nothing in this data type specifies the length of the array to be returned. That is determined by the arrays provided as parameters. However, the length needs to be known before the array to be returned can be constructed, because arrays are fixed-length collections.
- Another array variable named **longer** is used as an **alias** for the longer of the two array parameters. This makes the code that fills in the elements at the end of the **result** array easier to write.
- The two loops operating in sequence, one after the other, fill in every element in the **result** array.
- The *return* statement simply uses the variable **result**. This variable represents the entire array. Notice that **result** is declared with data type **double[ ]**, which matches the method's return type.

2.8 Swap Two Elements

An important basic array operation is swapping, or exchanging, two array elements. This simple operation is used in a number of different sorting algorithms. Here's a method that performs a swap:

```
public static void swap( int[ ] x, int a, int b )
{
   if ( a < 0 || a >= x.length || b < 0 || b >= x.length )
      throw new IndexOutOfBoundsException( "Illegal index" );

   int temp = x[a];
   x[a] = x[b];
   x[b] = temp;
}
```

Notice the following:

- The parameters **a** and **b** represent **index values** -- the indexes of the elements to be swapped. You may wish to **validate** the indexes and do something appropriate if they are not valid, as illustrated.
- A temporary variable is always required to perform a swap.
- The change takes place inside the array object provided as the actual parameter of the method call. No return value is required because the array parameter itself is modified.

2.9 Using Computed Array Indexes

In every example so far, each array index is either the value of a variable or a literal number. However, many situations require the array index to be **calculated** in some way. One common type of calculation is an **offset** -- the index used is the value of a counter offset by some amount.

The following example rotates the elements in the array one position to the left. This is accomplished in the following way:

1. Copy element 0 into a temporary variable.
2. Shift each of the other elements one position to the left.
3. Copy the value from the temporary variable into the last element of the array.

For example, here's an array and the result after a rotate-left operation:

{ 32, 18, 27, 12, 56, 15, 49 } // original array
{ 18, 27, 12, 56, 15, 49, 32 } // result after rotate-left

The following method implements this operation:

```
public static void rotateLeft( int[ ] x ) // Assumes length > 0
{
   int temp = x[0]; // Initialization
   for ( int i = 1; i < x.length; i++ ) {
      x[i-1] = x[i];
   }
   x[x.length-1] = temp; // Finalization
}
```

Notice the following:

- The value of **x[0]** is stored in the variable **temp**.
- Shifting the remaining elements to the left requires **length-1** operations, so the loop count is started at 1.
- **x[i-1]** is an example of a subscripted variable with a computed index value.
- A finalization step is needed to get the last value in place.
- This algorithm is a cousin of the swap algorithm we examined above. Can you see the similarities? Like the swap method, this is a *void* method -- the changes are made right in the array itself.
- The comment says that the method assumes the length of the array is > 0. Take a minute to trace through this method in the case where the length of the array is 1. Convince yourself that the method works properly in this 'degenerate' case.

What changes would you need to make to this method to produce a rotateRight method? This is a little bit tricky. To figure it out, sketch an array on paper and then plan out what you would need to do in order to rotate the elements to the right. Hint: the loop will need to count down. Try writing and testing the code!

3 Partially-Filled Arrays

Java arrays are fixed-capacity collections. The capacity must be specified when an array is constructed, and it can never be changed. However, many applications where arrays are used involve sets of data whose size may vary. The solution: partially-filled arrays.

The basic idea is this: construct an array that is initially larger than needed, and then use only some of its elements. One critical issue: the number of elements actually used needs to be stored somewhere. So, a partially-filled array always has a companion *int* variable that holds the 'size' of the array (the number of elements actually used).

Here's an example:

```
double[ ] grades = new double[100]; // length (or 'capacity') is 100
int gradesSize = 0; // size is initially 0 -- no elements are used

// add some grades
grades[gradesSize++] = 3.8;
grades[gradesSize++] = 4.0;
grades[gradesSize++] = 3.9;
grades[gradesSize++] = 4.0;

// calculate the average grade
double sum = 0;
for ( int i = 0; i < gradesSize; i++ )
   sum += grades[i];
double average = sum / gradesSize;
```

Notice the following things about this example:

- The variable **gradesSize** holds the number of elements actually being used in the partially-filled array **grades**. This also happens to be the index number for the next element to be filled. All elements actually being used are clustered at the lowest index values.
- Adding a new value to the partially-filled array involves assigning the value to the appropriate element (grades[gradesSize]) and also incrementing the gradesSize. This is handled very conveniently by the statement shown. When ++ is used in the postfix position, it means "use the current value of the variable in the expression, and then increment its value." Actually, the statements shown are not "safe" in general, because it is possible that the array is already full. Here is a safe version:
  ```
  if ( gradesSize < grades.length )
  grades[gradesSize++] = whatever;
  ```

- The variable **gradesSize** is used in place of **grades.length** in most situations -- for example, whenever you need to iterate through the array or otherwise need to know its 'size' (e.g. for calculating the average).

Partially-filled arrays are a useful tool to know!