Introduction to Computer Programming

Using the Java Programming Language

6. One Dimensional Arrays

Arrays are nicely organized lists of data. Think of a numbered list that starts at zero and extends one line every time you add something to the list. Arrays are useful for any number of situations because they're treated as a single hunk of data. For instance, if you wanted to store a bunch of high scores, you'd want to do that with an array. Initially, you might want to have a list of 10 items. You could in theory use the following code to store each score.

```
int score1;
int score2;
int score3;
int score4;
int score5;
int score6;
int score7;
int score8;
int score9;
int score10;
```

To make matters worse, if you needed to process each value, you'd need to create a set of code that deals with each variable by name. To check if `score2` is higher than `score1`, you'd need to write a function specifically to check those two variables before switching them. Thank goodness for arrays.

Declaring an Array

To declare an array in Java, we use the following syntax. Keep in mind that dataType may be any of the data types we have used up to this point (and we will extend upon this in the future).

```
dataType[] arr; or
dataType []arr; or
dataType arr[];
```

There is no *best* way to declare the array. All of the above are equally good ways to declare an array.

A declaration for an array to hold 10 scores would look like:

```
int scores[] = new int[10];
```

This statement declares an array `scores` of type integer, and then instantiates (or creates the actual storage in memory) the array to hold 10 values. These 10 values are numbered 0 through 9, just like Strings in Java are numbered 0 through the length of the string minus 1.

Initializing an Array

Initializing an array means to put values into the array. So, given our example of scores above, we need to load scores into the array. There are a few ways we can initialize an array with values.

First, we can place a value into each element of the array one by one. This would be done as follows:

```
int scores[] = new int[10];

scores[0]=85;
scores[1]=72;
scores[2]=99;
…
```

Clearly we can be more efficient given our knowledge of loops. Quite frankly, in almost all situations, we use arrays and loops hand-in-hand. Although the array we declared in our example held 10 scores, typically computers handle large amounts of data and we would be unable to use this method to fill each of the array elements.

To initialize an array with a loop, we could use the following code:

```
int scores[] = new int[10];

for (int i=0; i<10; i++) {
     System.out.println("Enter a score:");
     Scores[i] = in.nextInt();
     }
```

In the above example, the user would enter a score one by one to initialize the array.

Finally, a quick way of initialing an array can be done with the declaration statement. This accomplishes the task of declaring the array, instantiating the array (making room for it in memory), as well as assigning values to the array elements. An example of this method is shown here.

```
int scores[] = {75, 80, 85, 90, 95, 100, 60, 65, 75, 99};
```

This will assign 75 to `scores[0]`, 80 to `scores[1]`, etc.

Passing an Array to a Method

When discussing arguments/parameters and methods, we talked about *passing-by-value*. **Copies** of argument values are sent to the method, where the copy is manipulated and in certain cases, one value may be returned. While the copied values may change in the method, the original values in main did **not** change (unless purposely reassigned **after** the method).

The situation, when working with arrays, is somewhat different. If we were to make copies of arrays to be sent to methods, we could potentially be copying very large amounts of data. This would not be a very efficient thing to do in our programs and therefore Java does not do it!

Passing an array mimics a concept called "*passed-by-reference",* meaning that when an array is passed as an argument, its **memory address location** (its "reference") is used. In this way, the contents of an array CAN be changed inside of a method, since we are dealing directly with the actual array and not with a copy of the array. Note that in Java, we are not truly passing the array by reference, but for our purposes we can think that we are. The main point is to understand that if you pass an array to a method AND the method changes elements in the array, they will be changed when you return back to the calling method.

Let's look at an example of passing an array to a method. In this example we are passing an array of scores to a method and the method will return the lowest score.

```
public class Test{
      public static void main(String args[]){
            int scores[]={8,3,4,5};
            int minScore;

            minScore = GetMinimum(scores); //passing array to method
            System.out.println("The minimum score is: " + minScore);
      }

      public static int GetMinimum(int arr[]){
            int min=arr[0]; //Place the first score as min
            for(int i=1;i<arr.length;i++)
                  if(arr[i] < min)
                        min=arr[i];

            return min;
      }

}
```

Output from this program would yield:

```
The minimum score is: 3
```

Parallel Arrays

The following arrays represent data from a dog show. There are three separate arrays. Notice that the arrays do not all contain the same "type" of data. It is often necessary to represent data in a "table" form, as shown below. Such data, can be stored using parallel arrays. Parallel arrays are several arrays with the same number of elements that work in tandem to organize data. Keep in mind that it is up to the programmer to keep these arrays in sync and organized. Java does not know that the arrays are being used to hold data that is intended to be associated.

dogName

Wally	Skeeter	Corky	Jessie	Sadie

round1

18	22	12	17	15

round2

20	25	16	18	17

** One of the best features of parallel arrays is that the arrays do not have to hold the same types of data! **

In the data represented above, the first array is the dog's name, the second array is the dog's score in round 1 of the competition, and the third array is the dog's score in round 2 of the competition.

The arrays are parallel, in that the dog in the first element of the first array has the scores represented in the first elements of the second and third arrays.

If we were going to display the information from the rounds, we could do so with the following snippet of code:

```
System.out.println("Dog Name      Round 1 Score    Round 2 Score");
for(int i = 0; i < dogname.length; i++)
{
     System.out.println(dogname[i] + "   " + round1[i] + "   " + round2[i]);

}
```

Searching an Array

It is very common to want to look through an array of data to find a particular value. We say that we are *searching the array* for a key. The key is the value that you are looking for in the array. There are many ways to search an array, some of which are slower or faster than others.

Think of how important searching is in your day to day life. You type a word or phrase into Google and you hope that Google will return the exact information that you were looking for at the top of the search list. You also expect Google to do this as fast as it possibly can so that you are not waiting for a response.

Let's take a look at a couple of the more common ways we could search an array for a value.

Sequential Search

The simplest type of search is the sequential search. In the sequential search, each element of the array is compared to the key, in the order it appears in the array, until the desired element is found. If you are looking for an element that is near the front of the array, the sequential search will find it quickly. The more data that must be searched, the longer it will take to find the data that matches the key.

Consider this method which will search for a key integer value. If found, the index (subscript) of the first location of the key will be returned. If not found, a value of -1 will be returned.

```
public static int search(int[] numbers, int key)
{
       for (int i = 0; i < numbers.length; i++)
      {
           if ( numbers[i] = = key )
                 return i;  //Return the index where it was found.
      }
     // If we get to the end of the loop, a value has not yet
     // been returned.  We did not find the key in this array.
     return -1;
}
```

Now, suppose you are searching for the number of times a specific key value is included in an array:

```
//Find the number of times the name "Jones" appears in an array of name
public static void main(String[] args)
{
    String key = "Jones";
    int count;
    String[] list = new String [100];   // instantiate the array

    for (int i=0; i<100; i++) {     // fill the array
         System.out.println("Enter name: ");
         list [i]=input.nextLine();
     }

    count = search (list, key);    // invoke the method
    System.out.println("Count = " + count);
}

public static int search(String[] list, String key) {
     int i, count = 0;

     for(i = 0; i< list.length; i++)
     {
          if (list[i].equals(key))
             count = count+1;
     }
     return (count);
}
```

Another approach for searching is often done with a break command as well as a boolean to know when the key was found. Here is an example that utilizes a break and a boolean.

```
// Search for the number 31 in a set of integers
// This search takes place in main.
// This search could also have been placed in a method.
public class BreakBooleanDemo
{
     public static void main(String[ ] args)
     {
           int[] numbers = {12, 13, 2, 33, 23, 31, 22, 6, 87, 16};
           int key = 31;

           int i = 0;
           boolean found = false;

          for (i=0; i<numbers.length; i++)
          {
                if (numbers[i]==key)
                {
                    found = true;
                    break; // exits out of the for loop
                 }
           }

          if (found)
                System.out.println("Found "+key+" at index "+i+".");
          else
                System.out.println(key+"is not in this array.");
      }
}
```

Binary Search

Do you remember playing the game "Guess a Number", where the responses to the statement "I am thinking of a number between 1 and 100" are "Too High", "Too Low", or "You Got It!"? A strategy that is often used when playing this game is to divide the intervals between the guess and the ends of the range in half. This strategy helps you to quickly narrow in on the desired number.

When searching an array, the binary search process utilizes this same concept of splitting intervals in half as a means of finding the "key" value as quickly as possible. If the array that contains your data is in order (ascending or descending), you can search for the key item much more quickly by using a binary search algorithm ("divide and conquer").

Consider the following array of integers:

Array of integers, named num, arranged in "ascending order"!!

13	24	34	46	52	63	77	89	91	100
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]	num[6]	num[7]	num[8]	num[9]

We will be searching for the integer 77:

First, find the middle of the array by adding the array subscript of the first value to the subscript of the last value and dividing by two: (0 + 9) / 2 = 4 Integer division is used to arrive at the 4th subscript as the middle. (The actual mathematical middle would be between the elements 4th and 5th subscripts, but we must work with integer subscripts.)

The 4th subscript holds the integer 52, which comes before 77. We know that 77 will be in that portion of the array to the right of 52. We now find the middle of the right portion of the array by using the same approach. (5 + 9) / 2 = 7

The 7th subscript holds the integer 89, which comes after 77. Now find the middle of the portion of the array to the right of 52, but to the left of 89. (5 + 6) / 2 = 5

The 5th subscript holds the integer 63, which comes before 77, so we subdivide again
(6 + 6) / 2 = 6 and the 6th subscript holds the integer 77.

Remember: You must start with a pre-sorted array!!!

```
import java.util.Scanner;
public class BinarySearchExample
{
        public static void main(String[] args)
        {
          Scanner input = new Scanner (System.in);
          int key = 7;
          int[ ] num = new int [10];

          // Fill the array
          for (int i = 0; i < 10; i++) {
            System.out.println("Enter integer:");
            num[ i ]=input.nextInt();
          }
          input.close();
          //The binary search method
          binarySearch (num, 0, 9, key);
        }

        // Binary Search Method
        // This method accepts a pre-sorted array,
        // the subscript of the starting element for the search,
        // the subscript of the ending element for the search,
        // and the key number for which we are searching.
        public static void binarySearch(int[] array, int lowerbound, int upperbound, int
key)
        {
          int position;
          int comparisonCount = 1;   // counting the number of comparisons (optional)

          // To start, find the subscript of the middle position.
          position = ( lowerbound + upperbound) / 2;

          while((array[position] != key) && (lowerbound <= upperbound))
          {
            comparisonCount++;
            if (array[position] > key)// If the number is > key, decrease position by one.
            {
              upperbound = position - 1;
            }
            else
            {
              lowerbound = position + 1;    // else, increase position by one.
            }
            position = (lowerbound + upperbound) / 2;
          }
          if (lowerbound <= upperbound)
          {
            System.out.println("The number was found in array subscript " + position);
            System.out.println("The binary search found the number after " +
comparisonCount + " comparisons.");
            // printing the number of comparisons is optional
          }
          else
            System.out.println("Sorry, the number is not in this array. The binary search
made "                                +comparisonCount + " comparisons.");
        }
}
```

Binary search method:
binarySearch (num, 0, 9, key);

The arguments/parameters are:
array - the name of a **sorted** array
lowerbound – subscript (index) of first element to search, array [0]
upperbound – subscript (index) of last element to search, array[9]
key: item we wish to find.

This method simply prints the result of the search. You may wish to return the result of the binary search to be used in further investigations. To do so, return the "position" if the value is found and return a negative number (for example) if the result is not found.

Sorting an Array

There are many times when it is necessary to put an array in order from highest to lowest (descending) or vice versa (ascending). Because sorting arrays requires exchanging values, it is important that programmers use a technique to swap the positions within an array.

Let's look at a common mistake that beginning programmers make. Would the following code swap the values of x and y?

```
x = y;
y= x;
```

No. First, x would get replaced by y so that x and y contain the same value. When the second line was executed, y would be set to what it was originally set to, not to the original value of x.

We need to use a temporary holding place to successfully swap two values. Here's a sample of code to do it for a set of variables as well as two array locations.

```
//swap two variable values
temp = x;
x = y;
y = temp;


//swap two array locations
temp = a[i];
a[i] = a[j];
a[j] = temp;
```

There are many different ways to sort arrays. The basic goal of each method is to compare each array element to another array element and swap them if they are in the wrong position.

Some of the more common swapping algorithms include the bubble sort, the exchange sort, the selection sort, and the insertion sort. There are many different algorithms for sorting, but it is important that you understand a couple of the more popular ones.

Bubble Sort

In the bubble sort, as elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda. The bubble sort repeatedly compares adjacent elements of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two elements of the array are compared and swapped if out of order.

When the first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again. So, when does it stop? The bubble sort knows that it is finished when it examines the entire array and no "swaps" are needed (thus the list is in proper order). The bubble sort keeps track of occurring swaps by the use of a flag.

The table below follows an array of numbers before, during, and after a bubble sort for *descending* order. A "pass" is defined as one full trip through the array comparing and if necessary, swapping, adjacent elements. Several passes have to be made through the array before it is finally sorted.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	84	76	86	94	91	69
After Pass #2:	84	86	94	91	76	69
After Pass #3:	86	94	91	84	76	69
After Pass #4:	94	91	86	84	76	69
After Pass #5 (done):	94	91	86	84	76	69

```
// Bubble Sort Method for Descending Order
public static void BubbleSort(int[] num)
{
      int j;
      boolean swapped = true;
      int temp;   //holding variable
      while (swapped)
      {
          swapped= false;
          for(j=0; j<num.length-1; j++)
          {
              if (num[j] < num[j+1])
              {
                 temp = num[j];
                 num[j] = num[j+1];
                 num[j+1] = temp;
                 swapped = true;
              }
          }
      }
}
```

The bubble sort is an easy algorithm to program, but it is slower than many other sorts. With a bubble sort, it is always necessary to make one final "pass" through the array to check to see that no swaps are made to ensure that the process is finished. In actuality, the process is finished before this last pass.

Exchange Sort

The exchange sort is similar to its cousin, the bubble sort, in that it compares elements of the array and swaps those that are out of order. (Some people refer to the "exchange sort" as a "bubble sort".) The difference between the two sorts is the manner in which they compare the elements. The exchange sort compares the first element with each following element of the array, making any necessary swaps. When the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array swapping elements that are out of order. This sorting process continues until the entire array is ordered.

Let's examine our same table of elements again using an exchange sort for descending order. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	94	69	76	84	86	91
After Pass #2:	94	91	69	76	84	86
After Pass #3:	94	91	86	69	76	84
After Pass #4:	94	91	86	84	69	76
After Pass #5 (done):	94	91	86	84	76	69

The exchange sort, in some situations, is slightly more efficient than the bubble sort. It is not necessary for the exchange sort to make that final complete pass needed by the bubble sort to determine that it is finished.

```
//Exchange Sort Method for Descending Order (integers)
public static void ExchangeSort (int [] num)
{
     int i, j, temp;
     for (i=0; i<num.length-1; i++ )
     {
          for (j=i+1; j<num.length; j++)
          {
               if(num[i]<num[j])
               {
                   temp=num[i];   //swapping
                   num[i]=num[j];
                   num[j]=temp;
                }
          }
     }
}
```

When completed, the array will be in descending order and can be printed from main.