Arrays

- Arrays and Methods
- Searching
- Sorting Arrays

Reading:

=> Continue with section 2.1
As noted previously, an array element can be used in any context expecting an expression of the base type of the array:

- **Assignment**
  
  ```
  x = a[i];
  ```

- **Expression**
  
  ```
  x = 3 * a[i] + 5;
  ```

- **Actual parameter**
  
  ```
  x = largest(a[5],a[6]);
  ```

- **Returned by a method**
  
  ```
  return a[i];
  ```

An example with array elements as actual parameters:

[my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/parallelArrays1.txt](http://my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/parallelArrays1.txt)
Note that, just like a variable of type String, a variable of an array type is actually a pointer.

The fact that a variable of an array type is a pointer has several implications:
- copying arrays
- passing arrays as parameters
- comparing arrays for equality

Copying arrays:
www.cs.fit.edu/~pbernhar/teaching/cse1001/array1.txt
A method can have an array type as a formal parameter.

An array must be provided as an actual parameter in the method call.

- Use just the array name and no brackets.
- Note that a pointer to the array is passed, and not the whole array.

```java
public static void showArray(char[] a) {
    for(int i = 0; i <= a.length-1; i++)
        System.out.println(a[i]);
}

public static void main(String[] args) {
    char[] grades = new char[45];
    Random generator = new Random();

    for (int i=0; i<=grades.length-1; i++)
        grades[i] = (char)('A' + generator.nextInt(5));

    showArray(grades);
}
```
The length of the array passed can be different for each call.

- When you define the method you do not need to know the length of the array.
- Use the `length` attribute inside the method.

```java
public static void showArray(char[] a) {
    for(int i = 0; i <= a.length-1; i++)
        System.out.println(a[i]);
}

public static void main(String[] args) {
    char[] grades = new char[45];
    char[] status = new char[20];
    int[] myInts = new int[100];
    :
    // Generate random data for grades, status and myInts
    :
    showArray(grades);
    showArray(status);
    showArray(myInts); // What does this do?
}
What is output by the following?

```java
int[] a = new int[3];
int[] b = new int[3];

for(int i=0; i <= a.length-1; i++)
    a[i] = 5;

for(int i=0; i <= b.length-1; i++)
    b[i] = 5;

if(b == a)
    System.out.println("a equals b");
else
    System.out.println("a does not equal b");
```
A special method is required to test two arrays for equality.

This method returns true if and only if the arrays have the same length and all corresponding values are equal.

```java
public static boolean equals (int[] a, int[] b) {
    if (a.length != b.length)
        return false;
    else {
        for (int i=0; i<a.length-1; i++)
            if (a[i] != b[i])
                return false;
        return true;
    }
}
```
The `else` clause is not needed in this case:

```java
public static boolean equals (int[] a, int[] b) {
    if (a.length != b.length)
        return false;

    for (int i=0; i<=a.length-1; i++)
        if (a[i] != b[i])
            return false;

    return true; // Note the multiple points of exit
}
```
An example showing how the method is called:

```java
int[] A1 = {10, 20, 30};
int[] A2 = {10, 28, 30};
int[] A3 = {10, 28, 30};
int[] A4 = {5, 11, 31, 76, 90};

if (equals(A1,A2))
    System.out.println("yes they are equal");
else
    System.out.println("no they are not equal");

if (equals(A2,A3))
    System.out.println("yes they are equal");
else
    System.out.println("no they are not equal");

if (equals(A1,A4))
    System.out.println("yes they are equal");
else
    System.out.println("no they are not equal");
```
Another version, preferred by many, because of the single-point-of-exit:

```java
public static boolean equals(int[] a, int[] b) {
    int i;
    boolean match;

    match = true;
    if (a.length != b.length)
        match = false;
    else {
        i = 0;
        while (match && (i <= a.length-1)) {
            if (a[i] != b[i])
                match = false;
            i++;
        }
    }

    return match;        // Single point of return
}
```
As noted previously, whenever an array is passed as a parameter, the *address of* (or rather, a pointer to) the array is copied, not the array itself.

```java
public static void modifyArray(int[] a) {
    for(int i = 0; i <= a.length - 1; i++)
        a[i] = a[i] + 1;
}

public static void main(String[] args) {
    int[] scores = {10, 25, 9, -16};

    modifyArray(scores);
    for (int j = 0; j <= scores.length - 1; j++)
        System.out.println(scores[j]);
}
```

Array parameters are thus said to be *call by reference* parameters.
Parameters of a primitive type are called *call by value* parameters, because the *value* of the actual parameter is copied, and not a *pointer* to the actual parameter.

```java
public static void modInt(int x) {
    x = x + 1;
}

public static void main(String[] args) {
    int myInt;

    myInt = 0;
    modInt(myInt);
    System.out.println(myInt);
}
```

Parameters of most other types in Java are call by reference parameters.
A method can return an array:

```java
public static char[] vowels() {  
    char[] newArray = new char[5];
    newArray[0] = 'a';
    newArray[1] = 'e';
    newArray[2] = 'i';
    newArray[3] = 'o';
    newArray[4] = 'u';
    return newArray;
}

public static void main(String arg[]) {  
    char[] c;
    c = vowels();
    for (int i = 0; i <= c.length-1; i++)
        System.out.println(c[i]);
}
```
More Array Examples

- Lets “methodize” the previous examples:
  - my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/arrayTest1.txt
  - my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/parallelArrays2.txt
    (what does the array allocation and call to getGrades remind you of?)
  - my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/parallelArrays0.txt
    (bad example, doesn’t work)

- Another example:
  - www.cs.fit.edu/~pbernhar/teaching/cse1001/arrayadd.txt
For each of the following, give a Java method that (write them *by hand first*, type in second):

And also, for each of the exercises give sample code that calls the method.

**Easy:**
1. Has two integer arrays as parameters and *returns a third array* that contains the product of the corresponding elements in the two given arrays.
2. Has two integer arrays as parameters and returns a third array that contains the maximum of the corresponding elements of the two given arrays.
3. Has an int array and a single integer as parameters, and sets all locations in the array to 0 that contain the given integer.
4. Has two int arrays as parameters and returns *true* if the two arrays are identical and *false* if they are not.
5. Has one integer array as a parameter, and returns *true* if the array is sorted and *false* otherwise.

**Challenging:**
6. Has one int array as a parameter and *returns another array* that contains the contents of the first array in reverse order.
7. Has one int array as a parameter and reverses the contents of that array (how is this different from the previous one?).
8. Has a single int array as a parameter and returns the largest value in the array.
9. Same as the previous exercise, but where the method returns a position where the maximum value was located.
10. Has two integer arrays as parameters, and returns *true* if the first array is a subset of the second, and *false* otherwise.
11. Has two integer arrays as parameters, and returns true if the two arrays contain the same set of integers (in any order).
Array & Method Exercises

Difficult:

12. Has one int array as a parameter, and an integer \textit{pos}, where \(0 \leq \textit{pos} \leq \textit{a.length}-1\). The method should construct and return an array that is identical to given array, but with the value in position \textit{pos} deleted (Note that the resulting array should have length 1 less than the given array).

13. Has one int array as a parameter, and two integers \textit{pos1} and \textit{pos2}, where \(0 \leq \textit{pos1} \leq \textit{pos2} \leq \textit{a.length}-1\). The method should construct and return an array that is identical to given array, but with the values in positions \textit{pos1} though \textit{pos2} deleted.

14. Given an int array, and a single integer, construct an array that is identical to the given array, but with all occurrences of the given integer deleted.

15. Has one int array as a parameter, and perform a “left-shift” of the array, i.e., each element in the array moves one position to the left. Note that the element in the first position “drops off” of the array, and the last element stays as is.

16. Modify your solution from the previous exercise to do a “circular-left-shift” of the array, i.e., each element of the array moves one position to the left, with the element in the first position moving to the last position.

17. Has one int array as a parameter, plus two integers \textit{pos1} and \textit{pos2}, where \(0 \leq \textit{pos1}, \textit{pos2} \leq \textit{a.length}-1\) and \textit{pos1} \(\leq \textit{pos2}\). The method should perform a circular shift all the values in the array between positions \textit{pos1} and \textit{pos2}.

18. Has one int array as a parameter, plus three integers \textit{pos1}, \textit{pos2} and \textit{k}, where \(0 \leq \textit{pos1}, \textit{pos2} \leq \textit{a.length}-1\), \textit{pos1} \(\leq \textit{pos2}\) and \(\textit{k} \geq 0\). The method should perform a circular shift of all the values in the array \textit{k} times, between positions \textit{pos1} and \textit{pos2}.

19. Redo the last four exercises, but for a “right-shift.”

20. Has one integer array as a parameter, and returns \textit{true} if the array contains no duplicates and \textit{false} otherwise.

21. Has two arrays of integers as parameters, \textit{intArray1} and \textit{intArray2}, each containing no duplicates. The method should construct and return another array that contains the set-difference of the two given arrays. In other words, all the integers that appear in \textit{intArray1} but not \textit{intArray2}.

22. Same as the last one, but for set-intersection, and set-union.
Note that for the methods performing set operations (union, intersection, set-difference, and removing duplicates), determining the size of the resulting array is itself non-trivial, assuming the resulting array is to have length exactly equal to the number of elements in the result of the operation.

Redo any of the exercises, but for arrays of strings.
Give a Java method that has a single 2-D arrays as a parameter. The method should return true if the array is square and false otherwise.

```java
public static boolean square(int[][] a) {
    return (a.length == a[0].length);
}
```
Similarly, for each of the following, give a Java method that:

- **Easy:**
  1. Has one 2-D array of strings as a parameter, plus a string `str`, a (valid) row number `i`, and a (valid) column number `j`. The method should set the value in row `i` and column `j` of the array to the string `str`.
  2. Has one 2-D array of strings as a parameter, plus a string `str`, and a (valid) row number `i`. The method should set all of the values in row `i` of the array to the string `str`.
  3. Same as the last exercise, but where the integer `i` is a (valid) column number.

- **Challenging:**
  4. Has two 2-D arrays of ints as parameters. The method should return `true` if the arrays are identical (i.e., the same number of rows and columns, plus the same contents), and `false` otherwise.
  5. Has two 2-D arrays of ints as parameters that have the same number of rows and columns. The method should construct and return a third array that contains the maximum of the corresponding elements of the two given arrays.
  6. Has one 2-D array of ints as a parameter, plus two (valid) row numbers `i` and `j`. The method should swap the contents of rows `i` and `j`.
  7. Same as the previous exercise, but where `i` and `j` are valid column numbers.
2D Array Exercises

- Difficult:
  8. Has one 2-D int array as a parameter. The method should perform a vertical mirror-transformation of the array.
  9. Has one 2-D int array as a parameter. The method should perform a horizontal mirror-transformation of the array.
  10. Has one square 2-D int array as a parameter. The method should perform a top-left, bottom-right, diagonal-transformation of the array.
  11. Has one square 2-D int array as a parameter. The method should perform a bottom-left, top-right, diagonal-transformation of the array.
  12. Has two 2-D int arrays as parameters. The method should determine if the first array is a sub-array of the second.
  13. Given one 2-D integer array, performs a circular, horizontal shift of the array, i.e., each row shifts one-position to the right, but where the element at the end of row $i$ shifts to the first location of row $i+1$; also, the last element of the last row shifts to the first position of the first row.
  14. Same as the last exercise, but do a circular, vertical shift of the array.
  15. Has one 2-D array of ints as a parameter. The method should perform a circular right-shift of the outside “boarder” of the 2-D array, i.e., shift the top row right one position, shift the far right column down one position, shift the bottom row left one position, and shift the far left column up one position (similar to the lights on a marquee sign). Don’t forget to account for corner values.

- For each of the above, write the code by hand first, then type it in, compile it, and run it.

- For all of the above, give sample code that calls the method.
Given an array $A$ of integers, and an integer $x$, determine:

- If $x$ is in the array $A$
- The position of $x$ in $A$, if it exists

There are many algorithms for searching an array.

- Searching is a “classic” problem in computer science.

Sequential/linear search - search the array, from beginning to end (or from end to beginning).

- Not the most efficient search algorithm.
- Conceptually simple and easy to program.
Sequential Search

First example:

```java
public boolean inList(int x; int[] a) {
    for (int i=0; i<=a.length-1; i++)
        if (x == a[i])
            return true;
    return false;
}

// In the main
int[] myArray = {44, 19, 2, 10, 342, ...};
if (inList(24,myArray))
    System.out.println("Yes");
else
    System.out.println("no");
```

Note the distinction between “best-case,” “worst-case,” and “average-case” performance.
Another version (single point of exit):

```java
public boolean inList(int x; int[] a) {
    boolean found = false;
    int i = 0;
    while (!found && (i <= a.length-1)) {
        if (x == a[i])
            found = true;
        else
            i++;
    }
    return found;
}
```

Exercise - Modify `inList` so that it returns the position of the element, or -1 if it doesn’t appear in the array.
Binary Search

- Binary search:
  - More efficient than sequential search in the worst case (substantially)
  - Assumes the array is sorted

- Suppose we are searching for:
  - 45
  - -35
  - 23
  - 98
  - 21
  - 2
  - 29
  - 83
  - 31

- Give both informal, and formal traces of binary search.
Summary of binary search:

- At each step there is a segment of the array being searched, which is indicated by a “low” position and a “high” position.
- Look at the middle element between the “low” and “high.”
- If the middle element is the one you are looking for, stop.
- If not, repeat the search on the upper or lower half of the current segment of the array being searched.

See the program at:

- [http://cs.fit.edu/~pbernhar/teaching/cse1001/binarySearch.txt](http://cs.fit.edu/~pbernhar/teaching/cse1001/binarySearch.txt)

Binary search is much more efficient than sequential search.

- $O(\log n)$ vs. $O(n)$
Sorting a list of elements is another very common problem:

- sort numbers in ascending order (non-decreasing)
- sort numbers in descending order (non-increasing)
- sort strings in alphabetic order (lexicographic)

There are many algorithms for sorting (a partial list follows):

- Selection sort \( O(n^2) \)
- Bubble sort \( O(n^2) \)
- Insertion sort \( O(n^2) \)
- Quick sort \( O(n^2) \)
- Heap sort* \( O(n \log n) \)
- Merge sort*, etc. \( O(n \log n) \)
Selection Sort

- Sorting Algorithm #1: Selection sort
  - One of the easiest sorting algorithms.
  - Not particularly efficient.
  - Still useful – efficient algorithms are typically more complicated.

- To sort an array of integers in ascending order:
  - Find the smallest number in the array.
  - Put this number in the *first* position of the array.
  - Find the next smallest number in the array.
  - Put this number in the *second* position of the array.
  - Find the third smallest number in the array.
  - Put this number in the *third* position of the array.
  - Continue until the entire array is sorted.
Selection Sort

Original array:

| 7 | 6 | 11 | 17 | 3 | 15 | 5 | 19 | 30 | 14 |

1st iteration: smallest value is 3, its index is 4, swap a[0] with a[4]

| 7 | 6 | 11 | 17 | 3 | 15 | 5 | 19 | 30 | 14 |

| 3 | 6 | 11 | 17 | 7 | 15 | 5 | 19 | 30 | 14 |
Selection Sort

2nd iteration: smallest value in remaining list is 5, its index is 6, swap a[1] with a[6]

<table>
<thead>
<tr>
<th>3</th>
<th>6</th>
<th>11</th>
<th>17</th>
<th>7</th>
<th>15</th>
<th>5</th>
<th>19</th>
<th>30</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>11</td>
<td>17</td>
<td>7</td>
<td>15</td>
<td>6</td>
<td>19</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>11</td>
<td>17</td>
<td>7</td>
<td>15</td>
<td>6</td>
<td>19</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>17</td>
<td>7</td>
<td>15</td>
<td>11</td>
<td>19</td>
<td>30</td>
<td>14</td>
</tr>
</tbody>
</table>

How many iterations are needed?
Note that, in general, the algorithm repeats the following procedure:

- Find the $i^{th}$ smallest number in the array.
- Put this number in the $i^{th}$ position of the array.

Note that when looking for the number that goes in position $i$:

- The numbers in position 0 thru $i-1$ are already sorted.
- The number for position $i$ must be somewhere in positions $i$ thru $n-1$.

Also note that when you put the number in position $i$, the number already there must be put somewhere else.

- Swap it with the element to be moved into position $i$. 
// Selection sort
public static void sort(int[] a)
{
    int minPos, temp;

    for (int i=0; i<=a.length-2; i++) // Why a.length-2 ?
    {
        // Find the position of the value that belongs in position i
        minPos = i;
        for (int j=i+1; j<=a.length-1; j++)
            if (a[j] < a[minPos])
                minPos = j;

        // Swap the values in positions i and min
        temp = a[i];
        a[i] = a[minPos];
        a[minPos] = temp;
    }
}
More Generally…

On the $i^{th}$ iteration of the outer loop:
- Values in positions 0 through $i-1$ are already sorted and in their correct locations
- The value that belongs in position $i$, is somewhere in positions $i$ through $n-1$

During this $i^{th}$ iteration:
- The algorithm scans positions $i$ through $n-1$ to find the position, called $minpos$, of the smallest value
- The algorithm swaps the values in position $i$ and $minpos$
There are two levels of understanding here:
  - Understanding the algorithm.
  - Understanding the code.

This implies two types of hand-traces:
  - Array-level trace.
  - Code-level trace.

Exercise – do a trace on an array containing 50, 40, 30, 20, 10
Selection Sort

- Another version:

```java
public static void sort(int[] a) {
    int pos;
    for (int i = 0; i <= a.length - 2; i++) {
        pos = indexOfSmallest(i, a);
        interchange(i, pos, a);
    }
}
```

- Exercise – code methods `indexOfSmallest` and `interchange`. 
In some contexts an array variable can contain the “null” pointer; frequently this is considered first, otherwise errors might occur.

```java
public static boolean equals (int[] a, int[] b) {
    if (a == null) && (b == null)
        return true;
    else if (a == null) || (b == null)
        return false;
    else if (a.length != b.length)
        return false;
    else {
        for (i=0; i<=a.length-1; i++)
            if (a[i] != b[i])
                return false;
        return true;
    }
}
```

According to Dr. Stansifer (Java expert) this is not traditional Java style.
Another Version (without `else` clauses; again, not Java style):

```java
public static boolean equals (int[] a, int[] b)
{
    if (a == null) && (b == null)
        return true;

    if (a == null) || (b == null)
        return false;

    if (a.length != b.length)
        return false;

    for (i=0; i<=a.length-1; i++)
        if (a[i] != b[i])
            return false;

    return true;
}
```
Insertion Sort

- Sorting Algorithm #2: Insertion sort
  - Also very easy to understand and implement.
  - Also not very efficient.

- Basic idea:
  - Start with the first integer in the array as the “sorted portion.”
  - Keeping expanding the sorted portion of the array by one.
  - During each expansion step, insert the next integer into the correct spot in the sorted portion.
  - More specifically, during the \(i\)th iteration of the outer loop (where \(i\) is between 1 and \(n-1\)), insert the value in position \(i\) into the sorted portion of the array (which is in positions 0 through \(i-1\)).
Insertion Sort: An example

- First iteration:
  - [5], 3, 4, 9, 2
  - [3, 5], 4, 9, 2

- Second iteration:
  - [3, 5], 4, 9, 2
  - [3, 4, 5], 9, 2

- Third iteration:
  - [3, 4, 5], 9, 2
  - [3, 4, 5, 9], 2

- Fourth iteration:
  - [3, 4, 5, 9], 2
  - [2, 3, 4, 5, 9]
Initial version:

```java
// Insertion sort
public static void insertionSort(int[] a) {
    int j;

    for (int i=1; i<=a.length-1; i++) {
        j=i;
        while (j>=1) {
            if (a[j] < a[j-1]) {
                int temp=a[j-1];
                a[j-1]=a[j];
                a[j]=temp;
            }
            j=j-1;
        }
    }
}
```
Second version:

// This version is slightly more efficient
public static void insertionSort(int[] a) {
    int j;
    boolean done;

    for (int i=1; i<=a.length-1; i++) {
        j=i;
        done = false;
        while ((j>=1) && (!done))
            if (a[j] < a[j-1]) {
                temp=a[j-1];
                a[j-1]=a[j];
                a[j]=temp;
                j = j - 1;
            }
        else
            done = true;
    }
}
Third version:

// This version has a cosmetic improvement
public static void insertionSort(int[] a) {
    int j;
    boolean done;

    for (int i=1; i<=a.length-1; i++) {
        j=i;
        done = false;
        while ((j>=1) && (!done))
            if (a[j] >= a[j-1])
                done = true;
            else {
                int temp=a[j-1];
                a[j-1]=a[j];
                a[j]=temp;
                j = j - 1;
            }
    }
}
This one eliminates the boolean variable

```java
public static void insertionSort(int[] a) {
    int j;

    for (int i=1; i<=a.length-1; i++) {
        j=i;
        while ((j>=1) && (a[j]<a[j-1])) {
            temp=a[j-1];
            a[j-1]=a[j];
            a[j]=temp;
            j = j - 1;
        }
    }
}
```

Forth version:
Fifth version:

// Another slight improvement in efficiency
public static void insertionSort(int[] a)
{
    int j, v;

    for (int i=1; i<=a.length-1; i++) {
        j=i;
        v = a[j];
        while ((j>=1) && (v<a[j-1])) {
            a[j]=a[j-1];
            j=j-1;
        }
        a[j] = v;
    }
}
Sorting Algorithm #3: Bubble sort
- Also very easy to understand and implement.
- Also not very efficient.
- Several minor variations and enhancements are possible.

Basic Idea:
- Sorted portion of the array is at the upper end.
- Each iteration of the main loop expands the sorted portion by one.
- Each iteration of the main loop “pushes” the largest element in the unsorted portion to the sorted portion.
- Larger values “bubble up.”

Algorithm:
- Repeatedly compare adjacent values.
- Swap values that are out of order.
Bubble Sort: An example

First Iteration:
- \([5, 3], 4, 9, 2 \rightarrow [3, 5], 4, 9, 2\)
- \(3, [5, 4], 9, 2 \rightarrow 3, [4, 5], 9, 2\)
- \(3, 4, [5, 9], 2 \rightarrow 3, 4, [5, 9], 2\)
- \(3, 4, 5, [9, 2] \rightarrow 3, 4, 5, [2, 9]\)

Second Iteration:
- \([3, 4], 5, 2, 9 \rightarrow [3, 4], 5, 2, 9\)
- \(3, [4, 5], 2, 9 \rightarrow 3, [4, 5], 2, 9\)
- \(3, 4, [5, 2], 9 \rightarrow 3, 4, [2, 5], 9\)

Third Iteration:
- \([3, 4], 2, 5, 9 \rightarrow [3, 4], 2, 5, 9\)
- \(3, [4, 2], 5, 9 \rightarrow 3, [2, 4], 5, 9\)

Fourth Iteration:
- \([3, 2], 4, 5, 9 \rightarrow [2, 3], 4, 5, 9\)

=> last value is not compared

=> last two values are not compared

=> last three values are not compared
Bubble Sort

- Initial version:

```java
// Bubble sort
public static void bubbleSort1(int[] a)
{
    int temp;

    for (int i=1; i<=a.length-1; i++) {
        for (int j=0; j<a.length-i; j++) {
            if (a[j] > a[j+1]) {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```
Second version: (fewer bubbles)

// This version stops when a pass occurs with no swaps.
public static void bubbleSort1(int[] a) {
    int i, temp;
    boolean doMore;

    i = 1;
    doMore = true;
    while ((i<=a.length-1) && (doMore)) {
        doMore = false;
        for (int j=0; j<a.length-i; j++)
            if (a[j] > a[j+1]) {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
                doMore = true;
            }
        i = i + 1;
    }
}
Third version:

// Same as the second, but rearranged slightly.
public static void bubbleSort2(int[] a) {
    int k, temp;
    boolean doMore;

    k = a.length - 1;
    doMore = true;
    while (doMore) {
        doMore = false;
        for (int j = 0; j < k; j++) {
            if (a[j] > a[j + 1]) {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                doMore = true;
            }
        }
        k = k - 1;
    }
}
The forth version exploits the fact that the range that needs to be scanned on each iteration only needs to extend:

- From the position just before the left-most swap.
- To the position just after the right-most swap.

This range is captured by the variables `newLowest` and `newHighest`.

Note that although the forth version will correctly sort an array, it does not work as described above (you should fix it so that it does).
Forth version: (even fewer bubbles!)

```java
public static void bubbleSort2(int[] a) {
    int lowest, highest, newLowest, newHighest, temp;

    newLowest = 0;
    newHighest = a.length - 1;
    while (newLowest < newHighest) {
        lowest = newLowest;
        highest = newHighest;
        newLowest = a.length; // An arbitrarily high value
        for (int i = lowest; i < highest; i++)
            if (a[i] > a[i + 1]) {
                temp = a[i]; a[i] = a[i + 1]; a[i + 1] = temp;
                if (i < newLowest) {
                    newLowest = i - 1;
                    if (newLowest < 0)
                        newLowest = 0;
                }
                else if (i > newHighest)
                    newHighest = i + 1;
            }
    }
}
```
Question: why is the variable doMore missing?

Even more versions are possible:

- Alternately scan in both directions.

Google search for more versions.
How to Compare Algorithms in Efficiency

- Empirical Analysis:
  - Experimentation:
    - Wall-clock time
    - CPU time
    - Requires many different inputs
  - Can you predict performance before implementing the algorithm?

- Theoretical Analysis:
  - Approximation by counting important operations
  - Mathematical functions based on input size \( N \)
### How Fast/Slow Can It Get?

(10G Hz, assume $10^{10}$ operations/sec)

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N \log_2 N$</th>
<th>$N^2$</th>
<th>$2^N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>33</td>
<td>100</td>
<td>1,024</td>
</tr>
<tr>
<td>100</td>
<td>664</td>
<td>10,000</td>
<td>$1.3 \times 10^{30}$ (4 x10^{12} years)</td>
</tr>
<tr>
<td>(10^{-8} sec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>9,966</td>
<td>1,000,000</td>
<td>Forever??</td>
</tr>
<tr>
<td>10,000</td>
<td>132,877</td>
<td>100,000,000</td>
<td>Eternity??</td>
</tr>
</tbody>
</table>
Suppose we wish to determine if an array forms a palindrome:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>F</td>
<td>A</td>
<td>D</td>
<td>H</td>
<td>H</td>
<td>D</td>
<td>A</td>
<td>F</td>
<td>C</td>
</tr>
</tbody>
</table>

The following locations must be compared:

\[
\begin{align*}
A[0] & \quad - \quad A[9] \\
\end{align*}
\]
Consider the generic case:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>i</th>
<th>...</th>
<th>n-3</th>
<th>n-2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>F</td>
<td>A</td>
<td>D</td>
<td>H</td>
<td>H</td>
<td>D</td>
<td>A</td>
<td>F</td>
</tr>
</tbody>
</table>

The following locations must be compared:

- $A[0]$ and $A[n-1]$
- ...
- $A[i]$ and $A[?]$

In general, position $i$ (on the left-hand side) must be compared to position $n-(i+1)$. 

---

55
Initial solution:

```java
public static boolean palindrome(char[] A) {
    for (int i=0; i<=A.length-1; i++)
        if (A[i] != A[A.length-(i+1)])
            return false;
    return true;
}

// In main
char[] myChars = {'C','H',...,'C'};

if (palindrome(myChars))
    System.out.println("yup");
else
    System.out.println("nope");
```
Although correct, the previous version goes further than needed:

```java
public static boolean palindrome(char[] A) {
    for (int i=0; i<=(A.length/2)-1; i++)
        if (A[i] != A[A.length-(i+1)])
            return false;
    return true;
}

// In main
char[] myChars = {'C','H',...,'C'};

if (palindrome(myChars))
    System.out.println("yup");
else
    System.out.println("nope");
```
By default, an int array will be initialized to contain all 0’s.

- Explicit initialization is still recommended, however.

An array can be initialized in its declaration.

- No explicit allocation is required.
- Length of the array is determined automatically.

Example:

```java
double[] reading = {5.1, 3.02, 9.65};
System.out.println(reading.length);
```

Outputs 3, the length of the array reading
But it Even Gets Worse…

Do the following attempts to “copy” work correctly?

```java
String[] A1 = {“dog”, “cat”, “cow”};
String[] A2, A3;

A2 = new String[A1.length];
for (int i=0; i<=A1.length-1; i++)
    A2[i] = A1[i];

A3 = new String[A1.length];
System.arraycopy(A1, 0, A3, 0, A1.length);
```

Because strings are “immutable,” however, in most contexts the above copy works fine…