Lecture #8-10
Arrays

1. Array
data structure designed to store a fixed-size sequential collection of
elements of the same type
collection of variables of the same type

2. Array Declarations – Creates a Storage Location for a Reference to an Array,
i.e., Creating a Reference Variable for an Array
double[ ] temperature; -- preferred notation
double temperature[ ]; -- inherited from the C programming language

3. Creating Arrays – Allocating Memory for the Array,
   Specifying the Array Size, i.e., Determining the Array Length &
   Assigning a Reference to that Memory Location
   
   ```
   temperature = new double[24];
   ```
   Allocating Memory for an Array of doubles
   assigns sufficient memory to store 24 different temperature readings

4. Creating a Reference Variable, Allocating Memory & Assigning a Reference to that Memory Location
   
   ```
   double[ ] temperature = new double[24];
   ```

5. Assigning & Using Values Stored in an Array
   
   The statement
   ```
   double[ ] temperature = new double[6];
   ```
   creates an array
   
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.8</td>
<td>453.9</td>
<td>657.7</td>
<td>342.2</td>
<td>564.9</td>
<td>769.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
   where each cell can be referenced by its index, i.e.,
   ```
   temperature[0] = 54.7; // stores the value 54.7 in cell 0
   ```
   allocates sufficient memory to store 24 different temperature readings

   ```
   54.7 | 453.9 | 657.7 | 342.2 | 564.9 | 769.8 |
   ```
   index values

   ```
   double x = temperature[0] // retrieves the value 54.7 from cell 0
   ```
6. Array Size
   a. determined when the Array is created, i.e., when memory is allocated
   b. size of an array, e.g., temperature, can be determined by
      `temperature.length`
      which for the above example, returns the value 6
   c. once the array is allocated, the length cannot be modified

7. Array Indexed Variables
   ```
   temperature[0] = temperature[0] + 5.9;
   temperature[0] ← 54.7 + 5.9
   60.6
   ```

8. Array Initialization
   ```
   double[ ] temperature = {54.7, 453.9, 657.7, 342.2, 564.9, 769.8};
   ```

9. Processing Arrays
   ```
   for( i=0; i < temperature.length; i++)
   {
       temperature[i] = 0;
   }
   ```

   ```
   for( i=0; i < temperature.length; i++)
   {
       temperature[i] = Math.random()*100;
   }
   ```

   ```
   for( i=0; i < temperature.length; i++)
       System.out.println(temperature[i]);
   ```

```
   double total = 0;
   for( i=0; i < temperature.length; i++)
   {
       total += temperature[i];
   }```

   Compute the sum of the values of all variables in the array

   Initializing an array – all variable values set to zero

   Initializing an array – variable values set to random numbers such that
   0<= temperature < 100
double max = temperature[0];
double min = temperature[0];
for( i=0; i < temperature.length; i++)
{
    if( temperature[i] > max) max = temperature[i];
    if( temperature[i] < min) min = temperature[i];
}

double max = temperature[0];
int indexOfMax = 0;
for( i=0; i < temperature.length; i++)
{
    if( temperature[i] > max)
    {
        max = temperature[i];
        indexOfMax = i;
    }
}

double temp = temperature[0];
for( i=0; i < temperature.length; i++)
{
    temperature[i - 1] = temperature[i];
}

10. For-Each Loops

for( double element: temperature)
    System.out.println(element);

Compute the maximum and minimum values of all the variables in the array
Compute the smallest index of the, possibly multiple, maximum value of the array
Shifting the variable values from the left to the right

element must be the same type as the elements in temperature
For-Each Loops cannot be used to traverse the array in a different order nor for modifying the values of the array
11. Copying Arrays

Remember that `double[] temperature` creates a Reference Variable, i.e., creates storage space for a reference to an array, but does not allocate storage space for an array.

The statement

```java
double[] temperature1 = new double[24];
```

creates a reference variable for an array, allocates memory for an array and assigns the reference to that array. The reference variable `temperature1` can be visualized as a handle for the array or a pointer to the array.

Thus the statements

```java
double[] temperature2;
temperature2 = temperature1;
```

results in creating a new reference variable `temperature2` and making it refer to the same array as the reference variable `temperature1`.

a. Use a Loop Statement

```java
double[] temperature1 = {54.7, 453.9, 657.7, 342.2, 564.9, 769.8};
double[] temperature2 = new double[temperature1.length];
for (i=0; i < temperature1.length; i++)
    temperature2[i] = temperature1[i];
```
b. Use the Static `arraycopy` Method

Java's `System` class contains the method `arraycopy`, which violates the Java naming convention! The correct syntax is:

```java
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

- **source starting position**
- **target starting position**
- **number of elements to be copied**

```java
double[] temperature1 = {54.7, 453.9, 657.7, 342.2, 564.9, 769.8};
double[] temperature2 = new double[temperature1.length];
System.arraycopy(temperature1, 0, temperature2, 0, temperature1.length);
```

c. Use the Clone Method (Chapter 10 to be discussed later)

12. Passing Arrays to Methods

```java
public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
}

printArray(new int[] { 3, 1, 2, 6, 2 });
```

Java uses **pass-by-value** to pass arguments to a method.

For an argument of a primitive type, the arguments value is passed. For an argument of an array type, the value of the argument is a reference to an array; the value of the reference variable is passed. The effect is that arrays are **passed by reference**, i.e., the method has access to the values stored in the array.
public class Test
{
    public static void main(String[] args)
    {
        int x = 1;
        int[] y = new int[10];
        m(x, y);
        System.out.println("x is " + x);
        System.out.println("y[0] is " + y[0]);
    }
    public static void m(int number, int[] numbers)
    {
        number = 1001;
        numbers[0] = 5555;
    }
}
13. Passing Array Elements

public static void swap(int n1, n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}

public static void swapFirstTwoInArray(int [] array)
{
    int temp = array[0];
    array[0] = array[1];
    array[1] = temp;
}

Invocation Statements

int [] a = {1, 2};
swap( a[0], a[1]);
swapFirstTwoInArray(a);
14. Returning an Array from a Method

```java
public static int[] reverse(int[] list)
{
    int[] result = new int[list.length];
    for (int i = 0, j = result.length - 1; i < list.length; i++, j--)
        result[j] = list[i];
    return result;
}
```

```java
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
list2 == {6, 5, 4, 3, 2, 1}
```

15. Counting Letter Occurrences

Use the methods

```java
public static char getRandomCharacter(char ch1, char ch2)
{
    return (char) (ch1 + Math.random() * (ch2 - ch1 + 1));
}
```

```java
public static char getRandomLowerCaseLetter()
{
    return getRandomCharacter('a', 'z');
}
```

to build the array `chars` (see Liang pages 193-194 for details)

Use the method

```java
public static int[] countLetters(char[] chars)
{
    int[] counts = new int[26];
    for (int i = 0; i < chars.length; i++)
        counts[chars[i] - 'a']++;
    return counts;
}
```
to record the count of each lowercase letter in an array
16. Variable-Length Argument Lists

Used to pass a variable number of arguments to a method; in the method declaration, specify a variable-length argument by stating a data type followed by an ellipsis, i.e., “...”, followed by a variable name. A parameter list can contain only one variable-length argument and it must be placed at the end of the parameter list.

```java
public static void printMinAndMax(double ... numbers)
{
    if ( numbers.length == 0)
        return;
    double max = numbers[0];
    double min = numbers[0];
    for ( int i = 0;  i < numbers.length;  i++ )
        if ( numbers[ i ]  > result )
            max = numbers[ i ];
        else
            min = numbers[ i ];
    System.out.println( “Maximum: “ + max );
    System.out.println( “Minimum: “ + min );
}
```

Invocation of a Variable-Length Method

```java
printMinAndMax( ); // produces no output

printMinAndMax( 34, 3.2, 9, 15.1, 47, 106, 2.9); // produces
    Maximum: 106.0
    Minimum: 2.9

printMinAndMax(new double [] { 1, 54, -54 }); // produces
    Maximum: 54.0
    Minimum: -54.0
```
17. Linear Search (of an Array)

Searches array by comparing value of search item with each element in the array one-by-one; e.g., for the array a, below, the search for the value 564.9 would search, in order, cells 0, 1, 2, 3, & 4 where the search would terminate.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>123.8</td>
<td>453.9</td>
<td>657.7</td>
<td>342.2</td>
<td>564.9</td>
<td>769.8</td>
</tr>
</tbody>
</table>

A linear search method can be written is as follows:

```java
public class linearSearch( int [ ] list, int key ) // key is the search item
{
    for ( int = 0; i < list.length; i++ )
        if (key == list[ i ])
            { return i; }
    return -1;
}
```

It returns the index of the desired item as specified in the parameter key if such an item is found in the list, otherwise it returns -1 to signal failure to find such an item.

The execution time of a linear search increases linearly as n increases, hence it is said to be of the order n, i.e., O(n). On average it takes n/2 tries to find the item if it is in the array and n tries if it is not in the array. The linear search algorithm is usually avoided for searches in arrays with large n values.
18. Binary Search (of an Array)

In order to use this algorithm, the array must be sorted. Assume that it has been sorted in an ascending order, e.g., such as a telephone book. Think of how you search a phonebook; if you are searching for a last name that begins with the letter ‘K’, you generally open the book in the middle and determine whether the letter ‘K’ falls in the first half or the second half of the book. In some areas, it will fall in the first and in others it will fall in the second. In either case, you then ignore, i.e., “throw away” the other half of the phonebook as being irrelevant to the current search. You then repeat the process with the relevant section of the phonebook until you locate the desired name or determine that no such person is listed in that phonebook. This is a binary search.

In the first pass, you only need to search ½ of the original list; in the second pass you only search ½ of ½ of the original list; etc., etc., ... .

Assume n is a power of 2, i.e., n == 2^m for some m. After the first pass there n/2 elements left to search, after the second pass there are (n/2)/2 elements left to search and after the k\textsuperscript{th} search there are n/2^k elements to search.

When n == log_2 n, only one more element is left in the array and only one more search is necessary to determine whether the desired item is in the array. Hence, in the worst case, binary search requires at most log_2 n + 1 comparisons.

Remember that the linear search on an array of length n required n comparisons in the worst case.

Let low denote the first index and high denote the last index of the array segment that is currently being searched.
Initially low = 0 and high = list.length-1;

Let mid denote the index of the middle element of the array segment that is currently being searched. Hence mid = (low + high)/2; remember that the indexes are all integers so that the computation of mid uses integer division, e.g, mid = ( 15 + 42 )/2 yields 28.
The binary search algorithm given below, returns the index of the desired item, if such an item exists in the array, otherwise it returns the value 

\(-1 \times (insertion\ point + 1))

That is the index of the negation of the insertion point for the item or the index of the point at which the item should be inserted to maintain the sorted order of the array. If the item is not in the array the algorithm returns the value \((-low -1)\).

public class BinarySearch
{
    public static int binarySearch(int[] list, int key)
    {
        int low = 0;
        int high = list.length - 1;
        while (high >= low)
        {
            int mid = (low + high)/2;
            if (key < list[mid]) high = mid -1;
            else if (key == list[mid]) return mid;
            else low = mid + 1;
        }
        Return - low -1; // in this case we have high < low
    }
}

Homework: page 198 trace the algorithm as suggested
19. Selection Sort (of an Array)

Locate largest element in the array, swap that element with the element in the last place in the array. Locate second largest element in the array, swap that element with the element in the next to last place in the array. Continue this process until the array is sorted. For example,

```
2  9  5  4  8  1  6
2  6  5  4  8  1  9
2  5  4  1  8  9
2  1  5  4  6  8  9
2  1  4  5  6  8  9
2  1  4  5  6  8  9
1  2  4  5  6  8  9
```

```java
public class SelectionSort {
    public static void selectionSort(double[] list) {
        for (int i = list.length - 1; i >= 1; i--)
            {
                double currentMax = list[0];
                int currentMaxIndex = 0;

                for (int j = 1; j <= i; j--)
                    {
                        if (currentMax < list[j])
                            {
                                currentMax = list[j];
                                currentMaxIndex = j;
                            }
                    }

                if (currentMax != i)
                    {
                        list[currentMaxIndex] = list[i];
                        list[i] = currentMax;
                    }
    }
}
```

Homework: page 200 trace the algorithm as suggested
20. Insertion Sort (of an Array)

Example

\[ \begin{array}{cccccccc}
2 & 9 & 5 & 4 & 8 & 1 & 6 & \quad // \text{no move} \\
2 & 5 & 9 & 4 & 8 & 1 & 6 \\
2 & 4 & 5 & 9 & 8 & 1 & 6 \\
2 & 4 & 5 & 8 & 9 & 1 & 6 \\
1 & 2 & 4 & 5 & 8 & 9 & 6 \\
1 & 2 & 4 & 5 & 6 & 8 & 9 \\
\end{array} \]

public class Insertionsort
{
    public static void insertionSort(double[] list)
    {
        for ( int i = 1; i < list.length; i++ )
        {
            double currentElement = list[i];
            int k;
            for ( k = i-1; k >= 0 && list[k] > currentElement; k -- )
            {
                list[k + 1] = list[k];
            }
            list[k + 1] = currentElement;
        }
    }
}

Homework: page 202 trace the algorithm as suggested
21. Arrays Class

`java.util.Arrays`

- contains various static methods dealing with arrays
- overloaded for the primitive types
- `java.util.Arrays.sort(numbers); // sorts the array numbers`
- `java.util.Arrays.sort(numbers, 4, 17); // sorts the portion of the array between numbers[4] and numbers[16], inclusive.`

numbers

<table>
<thead>
<tr>
<th>123.8</th>
<th>...</th>
<th>657.7</th>
<th>...</th>
<th>564.9</th>
<th>769.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>16</td>
<td>17</td>
<td>123.8</td>
</tr>
</tbody>
</table>

- `java.util.Arrays.binarySearch(<array_name, search_target>);`
  e.g., `java.util.Arrays.binarySearch(numbers, 657.7);`

- `java.util.Arrays.equals(list1, list2);` e.g.,
  if list1, list2 and list3 are defined as
  ```
  int [ ] list1 = { 1, 2, 3, 5 };
  int [ ] list2 = { 1, 2, 3, 5 };
  int [ ] list3 = { 1, 2, 3, 4 };
  ```
  then
  ```
  java.util.Arrays.equals(list1, list2) returns TRUE
  and
  java.util.Arrays.equals(list1, list3) returns FALSE
  ```

- `java.util.Arrays.fill(list1, 0) → list1 contains { 0, 0, 0, 0 }`
  and
- `java.util.Arrays.fill(list2, 1, 2, 0) → list2 contains { 1, 0, 0, 5 }`
22. Two-Dimensional Arrays

```java
int[] matrix = new int[5][10]; // produces the two-dimensional array
```

where matrix[2][5] = 29 places the value 29 in the cell at row 2, column 5

```java
java.util.Arrays.fill(matrix[0], 1); // produces
```

A two-dimensional array is, in actual fact, an array of arrays, i.e.,

```
matrix[3][7] = 29 places the value 29 in the cell at row 3, column 7
```
23. Ragged Arrays

    int [ ] [ ] messyArray = {
        {1, 2, 3},
        {4, 5, 6, 7, 8, 9},
        {0},
        {7, 6, 35, 29, 190, -19, 57, 209, 41},
        {987, 8796, 1982}
    };

    produces

    | 1 | 2 | 3 |
    |---|---|---|
    | 4 | 5 | 6 |
    | 7 | 6 | 35| 29| 190| -19| 57| 209| 41|
    | 987| 8796| 1982 |

    int [ ] [ ] messyArray = new int [5][ ];
messyArray[0] = new int[3];
messyArray[1] = new int[6];
messyArray[2] = new int[1];
messyArray[3] = new int[9];
messyArray[4] = new int[3];

    produces
24. Processing Two-Dimensional Arrays

```java
int[][] matrix = new int[5][10];
int total = 0;

for (int row = 0; row < matrix.length; row++)
{
    for (int column = 0; column < matrix[row].length; column++)
    {
        matrix[row][column] = (int)(Math.random() * 100);
    }
}

for (int row = 0; row < matrix.length; row++)
{
    for (int column = 0; column < matrix[row].length; column++)
    {
        System.out.print(matrix[row][column] + " ");
    }
    System.out.println();
}

for (int row = 0; row < matrix.length; row++)
{
    for (int column = 0; column < matrix[row].length; column++)
    {
        total += matrix[row][column];
    }
}

for (int column = 0; column < matrix[0].length; column++)
{
    int total = 0;
    for (int row = 0; row < matrix.length; row++)
    {
        total += matrix[row][column];
        System.out.println("Column: " + column + "Total: " + total);
    }
}
```
25. Multidimensional Arrays

a. Two-Dimensional Arrays \(\rightarrow\) Tables, Matrices
   Consists of an Array of One-Dimensional Arrays, i.e., Vectors

b. Three-Dimensional Arrays \(\rightarrow\) Cubes, e.g., Rubic’s Cube
   Consists of
   i. an Array of Two-Dimensional Arrays, i.e., Tables or Matrices, or
   ii. an Array of an Array of One-Dimensional Arrays, i.e., Vectors

c. Four-Dimensional Arrays, e.g.,
   \(\text{temperature(hours[24], days[7], weeks[52] years[500])}\)
   defined by
   \(\text{double [ ] [ ] [ ] [ ] temperature = new double [24] [7] [52] [500];}\)
   Consists of ...

   The statement
   \(\text{temperature[11][2][4][5] = 97.4;}\)
   specifies that the temperature at recorded for the
   hour 11, day 2, week 4, year 5 was 97.4
   the absolute meaning of temperature[0][0][0][0] must be established
   by the context of the encompassing program.