Lecture Notes
Chapter #6
Arrays

1. Array
   - solves the problem of storing a large number of values and manipulating them
   - is a data structure designed to store a fixed-size sequential collection of elements of the same type, i.e., it is a 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. Array Creation
   - Specify the Array Size, i.e., Determine the Array Length
   - Allocate Memory for the Array &
   - Assign a Reference to that Memory Location

   \[
   \text{temperature} = \text{new double}[24];
   \]
   which allocates 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>
</tbody>
</table>
```

where each cell can be referenced by its index, e.g., `temperature[0] = 54.7;` which stores the value 54.7 in cell 0

```
<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>54.7</td>
<td>453.9</td>
<td>657.7</td>
<td>342.2</td>
<td>564.9</td>
<td>769.8</td>
</tr>
</tbody>
</table>
```

and
```
double x = temperature[0];
```
which retrieves the value 54.7 from cell 0 and places it in the variable x.

6. Array Size

- is determined when the array is created, i.e., when memory is allocated
- size of an existing array, e.g., `temperature`, can be determined by `temperature.length`
  which for the above example, returns the value 6
- once the array is allocated, the length cannot be modified

7. Array Indexed Variables

```
temperature[0] = temperature[0] + 5.9;
```
```
       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

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

Initializing an array – all variable values set to zero

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

Initializing an array – variable values set to random numbers such that $0\leq \text{temperature} < 100$

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

Printing all the values in an array

```java
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

```java
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];
}
```

Compute the maximum and minimum values of all the variables in the array

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

Compute the smallest index of the, possibly multiple, maximum value of the array

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

Shifting the variable values from the left to the right
10. **For-Each Loops**

```java
for( <data-type> element: temperature) 
    System.out.println(element);
```

**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

```java
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 & assigns the reference to that array.

The reference variable **temperature1** can be visualized as either a

- **handle for the array** or
- **a pointer to the array**.

Thus the statements

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

creates a new reference variable **temperature2** and making it refer to the same array as the reference variable **temperature1**.

```
<table>
<thead>
<tr>
<th>temperature1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array of Temperature Values</td>
</tr>
<tr>
<td>Temperature2</td>
</tr>
</tbody>
</table>
```

**a. Copying Arrays Using 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. Copying an Array using the Static `arraycopy` Method

java.lang.System contains the method `arraycopy`

`arraycopy` violates the java naming convention!
syntax
`arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);`

```
source starting position   target starting position   number of elements to be copied
```

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. Copying an Array using the Clone Method (to be discussed in Chapter 10)

12. Passing Arrays to Methods

```
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 });
```

```
Definition of printArray method
```

```
Invocation of printArray method
```

Anonymous array, i.e., there is no explicit reference variable holding the array; hence the array does not exist outside of the parameter list of the printArray method.

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); // x is 1
        System.out.println("y[0] is " + y[0]); // y[0] is 5555
    }

    public static void m(int number, int [ ] numbers)
    {
        number = 1001;
        numbers[0] = 5555;
    }
}

JVM uses an area of memory called the **heap** for dynamic memory allocation, where blocks of memory are allocated and freed in an arbitrary order, to store arrays.
13. Passing Array Elements

```java
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

```java
int[] a = {1, 2};
swap(a[0], a[1]);  // swap(1, 2); thus after swap a == {1, 2}
swapFirstTwoInArray(a);  // after swap a == {2, 1}
```

There is no way for the swap method to send the values back to the array `a`, i.e., pass by value

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 in an Array**

**Listing 6.4**

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');
}
```

Listing 6.4

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;
}
```

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 lower case letter in an array
16. **Variable-Length Argument Lists**

- are used to pass a variable number of arguments to a method
- specify a variable-length argument in the method declaration, 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 an array by comparing the value of a 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, and then 4 after which the search would terminate.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></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 i = 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 ¼ 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 \( k \)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.

Worst case search on a list of 1024 elements
- Linear Search \( \Rightarrow \) 1023 comparisons
- Binary Search \( \Rightarrow \) 11 comparisons

9300% improvement

Defining a Binary Search Algorithm

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 is contained in the array
- the value \(-1*\) (insertion point -1), if the item is not in the array

```java
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 in the text!!
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,

\[
\begin{array}{cccccccc}
2 & 9 & 5 & 4 & 8 & 1 & 6 \\
2 & 6 & 5 & 4 & 8 & 1 & 9 \\
2 & 6 & 5 & 4 & 1 & 8 & 9 \\
2 & 1 & 5 & 4 & 6 & 8 & 9 \\
2 & 1 & 4 & 5 & 6 & 8 & 9 \\
2 & 1 & 4 & 5 & 6 & 8 & 9 \\
1 & 2 & 4 & 5 & 6 & 8 & 9 \\
\end{array}
\]

```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)**

Moving on cell at a time across the array, place the selected value in its correct location, shifting the values of all the other cells to make the necessary room, e.g., forming a military line based on the decreasing/increasing height of the solders.

**Example**

```
2  9  5  4  8  1  6 // no move
  ↓   ↓
2  9  5  4  8  1  6
  ↓   ↓
2  9  5  4  8  1  6
  ↓   ↓
2  5  9  4  8  1  6
  ↓   ↓
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
```

```java
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, i.e., [4, 17).

```
<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th></th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.8</td>
<td>...</td>
<td>657.7</td>
<td>...</td>
<td>564.9</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
  ```java
  int [] list1 = { 1, 2, 3, 5 };
  int [] list2 = { 1, 2, 3, 5 };
  int [] list3 = { 1, 2, 3, 4 };
  ```
  then
  ```java
  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.,
```
```
```java
matrix[3][7]
```
```java
matrix[3], i.e., vector
```
```
matrix[0].length \rightarrow number of columns
```
23. Ragged Arrays

```java
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  8  9
0
7  6  35  29  190 -19  57  209  41
987  8796  1982
```

```java
int [][] messyArrayB = new int [5][ ];
messyArrayB[0] = new int[3];
messyArrayB[1] = new int[6];
messyArrayB[2] = new int[1];
messyArrayB[3] = new int[9];
messyArrayB[4] = new int[3];
```

produces

```
messyArrayB
```

```java
```
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[0].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[0].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);
}
```

- `matrix.length == 5`
- `matrix[row].length == 10`
i.e., `matrix[0].length == 10`
25. Multidimensional Arrays

a. Two-Dimensional Arrays ➔ Tables, Matrices
   Consists of an Array of One-Dimensional Arrays, i.e., Vectors

b. Three-Dimensional Arrays ➔ Cubes, e.g., Rubik'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.,
   temperature(hours[24], days[7], weeks[52], years[500])
   defined by
   double [ ] [ ] [ ] [ ] temperature = new double [24] [7] [52] [500];
   Consists of …
   i. an Array of Three-Dimensional Arrays, i.e., Tables or Matrices, or
   ii. an Array of an Array of an Array of One-Dimensional Arrays, i.e., Vectors

   The statement
   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

   For a general four-dimensional array, e.g., double votive[10][10][10][10];
   the meaning of votive[0][0][0][0] must be established by the context of the
   encompassing program.

26. Programs to Study, Trace, & Understand
   (programs that you will be expected to understand on the midterm exam)
   - Listing 6.11
   - Listing 6.12
   - Listing 6.13
   - Listing 6.14

27. Potential Midterm Examination Questions
   - Any Material Covered in Liang textbook Chapters 1 through 6 inclusively
   - Review Questions (any)
   - Program Listings in Liang (any)
   - Projects Assigned and Collected
   - Lectures 1 thru 6 (see Lecture Notes)

28. How to Study
   - Using the resources in #27 above, make an exhaustive set of notes of those points that you might
     be asked questions in the Exam, but which you think that you might not be able to recall
   - Reduce the notes created above to a set of 3x5 cards, listing only those points that you now think
     that you might not be able to recall
   - Repeat this process until you only need one 3x5 card
   - Reduce this 3x5 card to a 2x3 card; bring the card with you to look at 10 minutes before the exam
     starts; put the card away and take the exam!