Chapter 7

Arrays
<table>
<thead>
<tr>
<th>num1</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>num2</td>
<td>20</td>
</tr>
<tr>
<td>num3</td>
<td>30</td>
</tr>
</tbody>
</table>
A simple variable stores a single value

```
int num1 = 10;
int num2 = 20;
int num3 = 30;
```

A simple variable (int, char, double, bool etc) can store only one value at any given time.

```
int age = 30;  This initializes the variable age with the value 30, but this value can change during program execution.
```
If you need to store more than one value, use an array. An array can be thought of as a group of variables, of the same type, all sharing the same name.

Although the array is a multivariable entity, the entire group is known by only one name, thus simplifying the management of array variables.
An array is a contiguous (adjacent) block of memory locations. The block is addressed as a whole with the array name, whereas each element of the array is addressed using a subscript.
Array objects can hold any type of object

```java
int[] number = new int[100]; // stores 100 numbers

double[] salesTax = new double[10]; // stores 10 tax rates

char[] alphabet = new char[26]; // stores 26 characters
```

The size of the array determines the number of elements in the array.

Arrays may be of any type, such as a simple type offered by Java, or a built in object type, or even a user defined object type created by the programmer.

Although arrays may be of any type, each element of the array must be of the same type, i.e. all int or all char or some other type.

When declaring an array, make sure that you specify the size of the array within square brackets. The size indicates how many elements the array will have.
Arrays are initialized automatically by Java as a gesture of good will. Each type has an assigned default value as shown below:

Default values are: integral types (0), floating types (0.0), bool (false), string (empty), char (0).
You may initialize an array explicitly

```java
int[] numbers = {1, 2, 3, 4, 5};
```

Most often it is clearer to initialize your arrays explicitly during declaration. Note here that the size of the array is determined by the count of initial values, 5 in the above example.
The subscript refers to the offset from the first element of the array to any other element. So, the first element is offset 0 elements from the beginning of the array, while the second element is offset 1, the third is at offset 2 and so on. The last element of the array is always at an offset of n - 1, where n is the size of the array.
This requirement is based on the fact that each index or subscript is an offset from the start of the array. So, in order for the proper address to be computed the system must perform simple arithmetic like so.

Assume you have an array of chars. Each character is allocated one byte of memory and thus an array that has 10 elements will occupy 10 elements * 1 byte each = 10 bytes of memory. The first element will be offset 0 bytes from the start, the second element 1 byte and the tenth element 9 bytes from the start.

Now if the array is named arr, then a reference to the fourth element will be arr[3], and thus its offset will be 3 * 1 = 3 bytes from the start of the array. Its exact address will be <address of the start of the array> + 3 (the element’s offset).

As you can see the above computations involve integral values which leads to the requirement.
// (1) Declare an array to store the number of children
// living in each of the 100 houses in a city subdivision.
int[] numberOfChildrenPerHouse = new int[100];

// (2) Declare an array to store the number of children
// living in each of the 100 houses in a city subdivision.
final int NUMBER_OF_HOUSES_IN_SUBDIVISION = 100;
int[] numberOfChildrenPerHouse =
    new int[NUMBER_OF_HOUSES_IN_SUBDIVISION ];

The first statement used the value 100 directly in the array declaration. Using a literal constant in this way could create maintenance headaches later on, if we decide to change the array size.

The second statement uses a constant instead. Using constants makes the code a lot more readable and more importantly more maintainable. If we decide to change the size of the array, all we need to do is change the constant value.

Keep in mind that the program will of course have additional code where the array will be accessed and its size used in loops. If you use the literal constant (100) in all of this code, you will have to make all those changes later on, if the size were to change.

Using a constant variable, this is avoided, and only one change needs to be made to the constant itself.
// Read from the keyboard 100 values into the array.
for (int house = 1; house <= NUMBER_OF_HOUSES_IN_SUBDIVISION; house++) {
    System.out.print("Enter number of children for house "+
            house + ": ");
    numberOfChildrenPerHouse[house - 1] = keyboard.nextInt();
} // end for

Enter number of children for house 1: 3
Enter number of children for house 2: 2
...
Enter number of children for house 100: 1

The for statement cycles through all 100 homes, so the starting index is specified at 1.

However, arrays are 0 based, with the first element at index 0, so we must make the adjustment in the input statement.
// Display to the screen the array.
for (int house = 1;
    house <= NUMBER_OF_HOUSES_IN_SUBDIVISION;
    house++) {
    System.out.print("Number of children for house " +
    house +
    ": ") +
    numberOfChildrenPerHouse[house - 1]);
} // end for

Number of children for house 1: 3
Number of children for house 2: 2
... Number of children for house 100: 1
You want to track the daily high and low temperatures for the month of July.

The temperature values will be entered by the user at the keyboard.

Once all the temperatures have been entered, the program will compute the daily temperature difference and store that as well.

Finally, all three values are to be displayed to the screen.
Arrays in Classes and Methods
Arrays can be local variables

public void getHourlyEmployees() {
    Employee[] hourlyEmployee;
    ...
}

Arrays can be parameters

```java
public static void main(String[] args) {
    ...
}
```
```java
public void readIntegerNumbers(int[] number) {
    System.out.print("Enter " + number.length + " integers: ");

    Scanner keyboard = new Scanner(System.in);
    for (int i = 0; i <= number.length - 1; i++) {
        number[i] = keyboard.nextInt();
    } // end for
} // end readIntegerNumbers()
```
Arrays can be return values

```java
public Employee[] getEmployees() {
    ...
}
```
public int[] getIntegerArray(int elements, int initValue) {
    int[] array = new int[elements];

    for (int i = 0; i <= array.length – 1; i++) {
        array[i] = initValue;
    }   // end for

    return array;
}   // end readIntegerNumbers()
Arrays are objects, thus

```java
type] a = {1, 2, 3};
type] b;

b = a;  // makes b and a refer to the same
         // memory location
```
public void copyArray(int[] source, int[] target) {
    // Both arrays must be the same size.
    for (int i = 0; i <= source.length - 1; i++) {
        target[i] = source[i];
    }
    // end for
}
// end readIntegerNumbers()
Arrays are objects, thus

```java
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};

if (a == b) {...} // evaluates to false
    // since a and b refer to two
    // different memory locations
```
public boolean areEqual(int[] array1, int[] array2) {
    boolean equal = true;
    int i = 0;
    if (array1.length != array2.length) {
        equal = false;
    } else {
        while (equal && i <= array1.length - 1) {
            equal = (array1[i] == array2[i])? true : false;
            i++;
        }
    }
    return equal;
} // end readIntegerNumbers()
You want to use a class to store some personal information on the 5 people you just met at a conference.

- You want to save their name, and address, along with their cell phone number.

- You decide to write a small program that inputs the records and displays them to the screen.
Programming with Arrays and Classes
package noduplicateslistdemo;

public class NoDuplicatesListDemo {

    public static void main(String[] args) {
        NoDuplicatesList nflTeams = new NoDuplicatesList(4);
        String team;

        nflTeams.addItem("Chicago Bears");
        nflTeams.addItem("Miami Dolphins");
        nflTeams.addItem("St. Louis Rams");
        nflTeams.addItem("Chicago Bears");
        nflTeams.addItem("Dallas Cowboys");
        nflTeams.addItem("Arizona Cardinals");
    }
}
Application Deconstructed

<NoDuplicatesListDemo.java>

```java
nflTeams.moveFirst();
    team = nflTeams.getNext();
    while (team != null) {
        System.out.println("Team: "+ team);
        team = nflTeams.getNext();
    } // end while
} // end main()
} // end NoDuplicatesListDemo
```

![Output - NoDuplicatesListDemo (run)](image)

List is full.
- Team: Chicago Bears
- Team: Miami Dolphins
- Team: St. Louis Rams
- Team: Dallas Cowboys
BUILD SUCCESSFUL (total time: 0 seconds)
package noduplicateslistdemo;

public class NoDuplicatesList {
    private static final int DEFAULT_LIST_SIZE = 100;
    private String[] items;
    private int numberOfItems;
    private int indexOfCurrentItem;

    // Allocates the array and updates the numberOfItems.
    private void set(int listSize) {
        items = new String[listSize];
        numberOfItems = 0;
    }
    // end set()
// Returns true if item already exists on the list.
private boolean isOnList(String item) {
    boolean found = false;
    int i = 0;

    while (!found && i <= numberOfItems - 1) {
        if (items[i].equalsIgnoreCase(item)) {
            found = true;
            break; // end if
        } else {
            i++;
        } // end if
    } // end while

    return found;
} // end isOnList()
// Allocates an array of default size.
public NoDuplicatesList() {
    this(DEFAULT_LIST_SIZE);
} // end NoDuplicatesList()

public NoDuplicatesList(int initialListSize) {
    set(initialListSize);
} // end NoDuplicatesList()

// Sets the indexOfCurrentItem to 0.
public void moveFirst() {
    indexOfCurrentItem = 0;
} // end moveFirst()

// Returns the first item.
public String getFirst() {
    return items[0];
} // end getFirst()
// Returns the last item.
public String getLast() {
    return items[numberOfItems - 1];
} // end getLast()

// Returns the next item in the list. If the current item
// happens to be the last one, then it returns null.
public String getNext() {
    String item = null;
    if (indexOfCurrentItem <= numberOfItems - 1) {
        item = items[indexOfCurrentItem];
        indexOfCurrentItem++;
    } // end if

    return item;
} // end getNext()
// Returns true if list is empty, false otherwise.
public boolean isEmpty() {
    return numberOfItems == 0;
} // end isEmpty()

// Returns true is list is full, false otherwise.
public boolean isFull() {
    return numberOfItems == items.length;
} // end isFull()
// Adds the item to the end of the list, but without duplicates. Updates the indexOfCurrentItem to point to this item.
public void addItem(String item) {
    if (isFull()) {
        System.out.println("List is full.");
    } else if (!isOnList(item)) {
        items[numberOfItems] = item;
        indexOfCurrentItem = numberOfItems;
        numberOfItems++;
    } // end if
} // end addItem()
} // end NoDuplicatesList
Sorting and Searching Arrays
The basic idea behind linear search is that you start searching with the first element of the array and continue until you either a) find the value you are searching, or b) the value is not contained in the array. At that point all elements would have been searched.
public boolean contains(String[] items, String item) {
    boolean found = false;
    int i = 0;
    while (!found && i <= items.length - 1) {
        if (items[i].equalsIgnoreCase(item)) {
            found = true;
        } else {
            i++;
        }
    }
    return found;
} // end contains()
You can easily find the min/max value

Finding the minimum value in a list starts off with one basic assumption, that the first value could be the smallest.

Once the assumption has been made, then it becomes a simple matter of comparing all the other numbers against the current min, until another min is found.

This algorithm works equally well when searching for the maximum value.

One notable distinction the actual code will make is to keep track of the index of the min value, rather than the value itself. Having the index you can always refer to the value, but having the value does not give you access to the index.
// Returns the index of the next smallest value in the array.
private int indexOfNextSmallest(int startAt, int[] list) {
    // Assume smallest value is at startAt.
    int minIndex = startAt;

    // Now, let's try and find another value that is less.
    for (int i = startAt; i <= list.length - 1; i++) {
        minIndex = (list[i] < list[minIndex])? i : minIndex;
    } // end for

    return minIndex;
} // end indexOfNextSmallest()
Selection Sort Deconstructed

- Go through the array n - 1 times, where n is array size.

- In each pass, select next value in order (smallest, largest)

- Exchange with top value in unordered portion

- Continue with next value until no more values left

Selection sort basically stands up to its name. It selects a value from a select group of values, and places it in its final ordered position.

The array starts out as two regions, an empty region containing all the values that have been placed in their correct position by value, and a second region that includes all the values in the array that are still out of range.

The algorithm runs through the array n - 1 times, n being the number of elements in the array. During each pass, the next value - smallest or largest - is found and exchanged with the first value in the unordered portion of the array.

This process is repeated until all values have been placed in order.
Pass 1: The smallest value -6 is swapped with the top value of the unordered region, -4. At the end of the pass, the -6 is in its final position in the ascending order being performed.
Pass 2: The next smallest value in the remaining region [1] to [4], -4, is now swapped with 7.
Selection Sort
<Pass 3>

<table>
<thead>
<tr>
<th>MEMORY</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] -6</td>
<td>[0] -6</td>
</tr>
</tbody>
</table>

Find the next value in order

Pass 3 of 4

The process continues with 7 swapping with 9.
And finally the 9 swaps with itself to complete the sort in 4 passes.
### Selection Sort
<Array sorted>

#### After 4 passes the array is in order

<table>
<thead>
<tr>
<th>[0]</th>
<th>-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>-4</td>
</tr>
<tr>
<td>[2]</td>
<td>7</td>
</tr>
<tr>
<td>[3]</td>
<td>9</td>
</tr>
<tr>
<td>[4]</td>
<td>12</td>
</tr>
</tbody>
</table>
// Sorts the array in ascending order.
private void selectionSort(int[] array) {
    int minIndex;
    for (int topIndex = 0; topIndex <= array.length - 2; topIndex++) {
        // Find the next smallest value.
        minIndex = indexOfNextSmallest(topIndex, array);
        // Exchange with top value.
        swap(topIndex, minIndex, array);
    }
}

// Swaps the two values identified by the top and minIndex parameters.
private void swap(int index1, int index2, int[] array) {
    int tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
}

The for statement in the selection sort method visits all the elements in the array, from the first (0) to the last (len - 1). For each element in the unordered section of the array, the next smallest value is found and swapped with this top value.

The swap function receives the array and the two indices of the elements to be swapped.
package selectionsortdemo;

public class SelectionSortDemo {
    public static void main(String[] args) {
        int[] number = {-4, 0, 5, 12, 9, 5, -20, 30};
        System.out.println("Original array: ");
        displayArray(number);
        selectionSort(number);
        System.out.println();
        System.out.println();
        System.out.println("Sorted array: ");
        displayArray(number);
        System.out.println();
    } // end main()
}
private static void displayArray(int[] number) {
    for (int i = 0; i <= number.length - 1; i++) {
        System.out.print(number[i] + " ");
    } // end for
} // end displayArray()

private static void selectionSort(int[] number) {
    for (int topIndex = 0; topIndex <= number.length - 2; topIndex++) {
        swap(topIndex, indexOfNextSmallest(topIndex, number), number);
    } // end for
} // end selectionSort()
private static int indexOfNextSmallest(int startIndex, int[] array) {
    int minIndex = startIndex;
    for (int i = startIndex; i <= array.length - 1; i++) {
        minIndex = (array[i] < array[minIndex]) ? i : minIndex;
    }
    return minIndex;
}

private static void swap(int index1, int index2, int[] array) {
    int tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
}

private static int selectionSortDemo(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int minIndex = indexOfNextSmallest(i, array);
        swap(i, minIndex, array);
    }
    return 0;
}

Code Deconstructed
<selectionSort>

Output - SelectionSortDemo (run)

run:
Original array: -4 0 5 12 3 5 -20 30
Sorted array: -20 -4 0 5 5 9 12 30
BUILD SUCCESSFUL (total time: 1 second)