



Chapter 9: Arrays

Objectives

Students should

- Be able to define, initialize, and use one-dimensional as well as multidimensional arrays correctly.
- Be able to use arrays as well as their elements as parameters to methods.
- Be able to write code to sort array elements in any orders desired.
- Be able to write code to search for elements in an array.
- Be able to use arrays in problem solving using computer programs.

Requirement for a List of Values

Suppose we want to write a program that counts the number (frequency) of each digit from 0 to 9 in a *String* entered by the user, the program might be written using what we have studied so far be like this:

```
import java.io.*;
public class CountDigitFrequency
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader stdin
            = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter string:");
        String s = stdin.readLine();
        int freq0 = 0; int freq1 = 0;
        int freq2 = 0; int freq3 = 0;
        int freq4 = 0; int freq5 = 0;
        int freq6 = 0; int freq7 = 0;
        int freq8 = 0; int freq9 = 0;
        for(int i=0;i<s.length();i++){
            char c = s.charAt(i);
            if(c >= '0' && c <= '9'){
                switch(c){
                    case '0': freq0++; break;
                    case '1': freq1++; break;
                    case '2': freq2++; break;
                    case '3': freq3++; break;
                    case '4': freq4++; break;
                    case '5': freq5++; break;
                    case '6': freq6++; break;
                    case '7': freq7++; break;
                    case '8': freq8++; break;
                    case '9': freq9++; break;
                    default:
                }
            }
        }
        System.out.println("Number of 0 = "+freq0);
        System.out.println("Number of 1 = "+freq1);
        System.out.println("Number of 2 = "+freq2);
        System.out.println("Number of 3 = "+freq3);
        System.out.println("Number of 4 = "+freq4);
        System.out.println("Number of 5 = "+freq5);
        System.out.println("Number of 6 = "+freq6);
        System.out.println("Number of 7 = "+freq7);
        System.out.println("Number of 8 = "+freq8);
        System.out.println("Number of 9 = "+freq9);
    }
}
```

From the code, you should be able to notice that ten variables are used for storing the frequencies of the digits. Also, separate instances of *System.out.println()* are called for printing



the resulting frequencies. Although the names of the variables storing the frequencies are rather similar, they are independent from one another. Therefore, we cannot make use of iterative constructs learned previously, which would significantly reduce the size of the code. Not being able to use iterative constructs in this case also leads to clumsy and error-prone source code.

What we require is a mechanism that enables us to store a list of related values in a single structure, which can be referred to using a single identifier. Java provides this mechanism through the use of *arrays*.

Specifically to the above example, we need a list of ten elements, in which each element is used for storing the frequency of each digit.

One-dimensional Array

An array variable is used for storing a list of element. Similar to a variable of other data types, an array variable needs to be declared. The declaration of an array takes a rather similar syntax to the declaration of a variable of other data type, which is:

```
ElementType [] a;
```

`ElementType` is the data type of each element in the array. The square bracket `[]` identifies that this is an array declaration. `a` is the identifier used for referring to this array. Note that identifier naming rules applies here also.

Below are some examples when arrays of various data types are declared.

```
int [] freq;  
int [] numICE, numADME;  
double [] scores;  
char [] charList;  
string [] studentNames;
```

Note that more than one arrays of the same type can be declared in a single statement such as the one shown in the second example. Also, it is perfectly correct and common to create an array of non-primitive data type such as an array of *String* shown in the last example above.

To initialize an array, the keyword *new* is used in the following syntax.

```
a = new ElementType[n];
```

`n` is a non-negative integer specifying the length of `a`. Here are some examples of array initialization associated with the arrays declared in the above example.

```
freq = new int[10];  
numICE = new int[5];  
int n = 5;  
numADME = new int[n];  
scores = new double[50];  
charList = new char[2*n];  
studentNames = new String[40];
```

Notice that we can use any expression that is evaluated to an `int` value, as well as an explicit `int` value, to specify the length.

Declaration and initialization can be done in the same statement. For example,

```
int [] freq = new int[10];  
int n = 5;  
int [] numADME = new int[n];
```



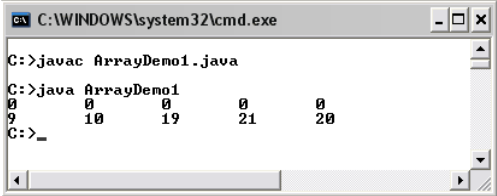
Whenever an array is initialized using the keyword *new*, every of its element is set to a numeric zero if the elements are of numeric types, to the *boolean false* if the elements are of *boolean* type, and to *null* (a reference to nowhere) if the elements are of non-primitive data types.

Accessing Array Elements

Elements in an array can be referred to using its associated *index*. For an array of length *n*, the corresponding indexes run from 0 to *n*-1. An element of an array *a* with index *k* can be referred to in the program using *a[k]*.

Consider the following program. Note that the length of an array *a* can be found using *a.length*.

```
public class ArrayDemo1                                1
{                                                        2
    public static void main(String[] args)              3
    {                                                    4
        int [] a = new int[5];                          5
        for(int i=0; i<a.length; i++){                  6
            System.out.print(a[i]+"\\t");                7
        }                                                8
        System.out.print("\\n");                        9
        a[0] = 9;                                       10
        a[1] = 10;                                     11
        a[2] = a[0]+a[1];                              12
        a[3] = 20;                                     13
        a[4] = a[3]++;                                  14
        for(int i=0; i<a.length; i++){                  15
            System.out.print(a[i]+"\\t");                16
        }                                                17
    }                                                    18
}                                                        19
```



An array of *int* of length 5 is declared and initialized on line 5. Then, a *for* loop is used to iterate through *a[i]* from *i* equals 0 to *a.length*-1. We can see that the program prints 0 0 0 0 0 out on the screen. The each element of the array *a* is assigned with different *int* value from line 10 to line 14. The *for* loop on line 15 makes the program prints the values of every element, which are 9, 10, 19, 21, and 20, on screen. Recall that the post fix increment operator on line 14 makes assign the value of *a[3]* to *a[4]* prior to increasing *a[3]* by 1.

Example

Here we can modify *CountDigitFrequency.java* by storing the frequency of each digit occurrence in a single array. Appropriate iterative constructs are also used. Notice that the code is much more compact.

```
import java.io.*;                                      1
public class CountDigitFrequency2                      2
{                                                        3
    public static void main(String[] args) throws IOException  4
    {                                                    5
        BufferedReader stdin
```



```
        = new BufferedReader(new InputStreamReader(System.in));           6
    System.out.print("Enter string:");                                   7
    String s = stdin.readLine();                                       8
    int [] freq = new int[10];                                         9
    for(int i=0;i<s.length();i++){                                    10
        char c = s.charAt(i);                                         11
        if(c >= '0' && c <= '9'){                                     12
            freq[c-'0']++;                                           13
        }                                                            14
    }                                                                  15
    for(int i=0;i<freq.length;i++){                                   16
        System.out.println("Number of "+i+" = "+freq[i]);           17
    }                                                                  18
    }                                                                  19
}                                                                      20
```

```
C:\>javac CountDigitFrequency2.java
C:\>java CountDigitFrequency2
Enter string:08091211234
Number of 0 = 2
Number of 1 = 3
Number of 2 = 2
Number of 3 = 1
Number of 4 = 1
Number of 5 = 0
Number of 6 = 0
Number of 7 = 0
Number of 8 = 1
Number of 9 = 1
C:\>_
```

Explicit Initialization

If we would like each element in an array to have the value other than the default value (zero for numeric types and false for `boolean`) during its initialization, we can use an *initializer list*. An initializer list is a list of values, each of which is separated by a comma, enclosed in a pair curly bracket. For example:

```
int [] a = {1,2,3,4,5};
String [] programs = {"ADME", "AERO", "ICE", "NANO"};
boolean [] allTrue = {true, true, true};
```

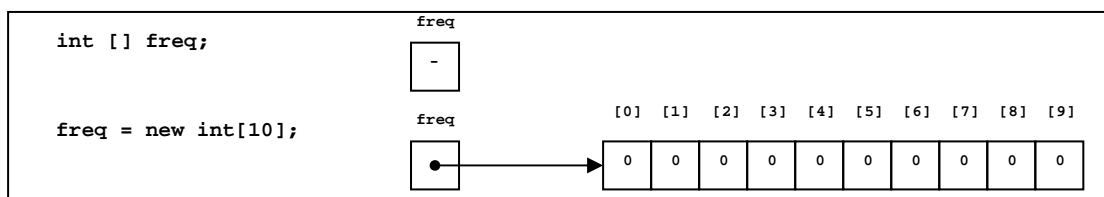
Initializer lists have to be used in the statements in which array variables are declared. Therefore, the following statements are invalid.

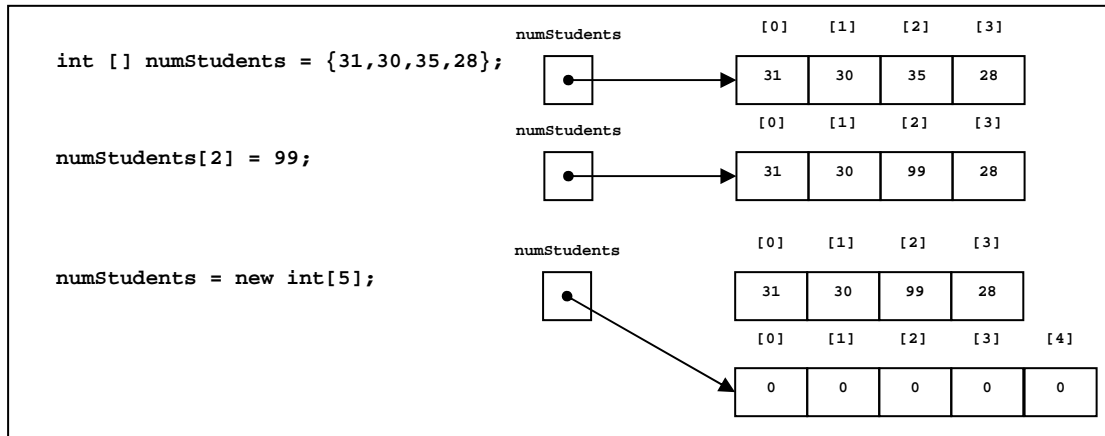
```
int [] a;
a = {1,2,3,4,5}; // This is invalid.
```

Array Variables are Reference Variables.

Just like variables storing values of non-primitive data types (such as *String*), array variables are reference variables. The value that is actually stored in an array variable is a reference to the real array object locating somewhere in the memory.

The following pictures show illustrations of how memory is arranged when array-related statements are executed.





An array variable can be assigned with the value of another array variable of the same type. For example, the following code segment is valid.

```

char [] a = {'A','B','C'};
char [] b = {'X','Y'};
b = a;
    
```

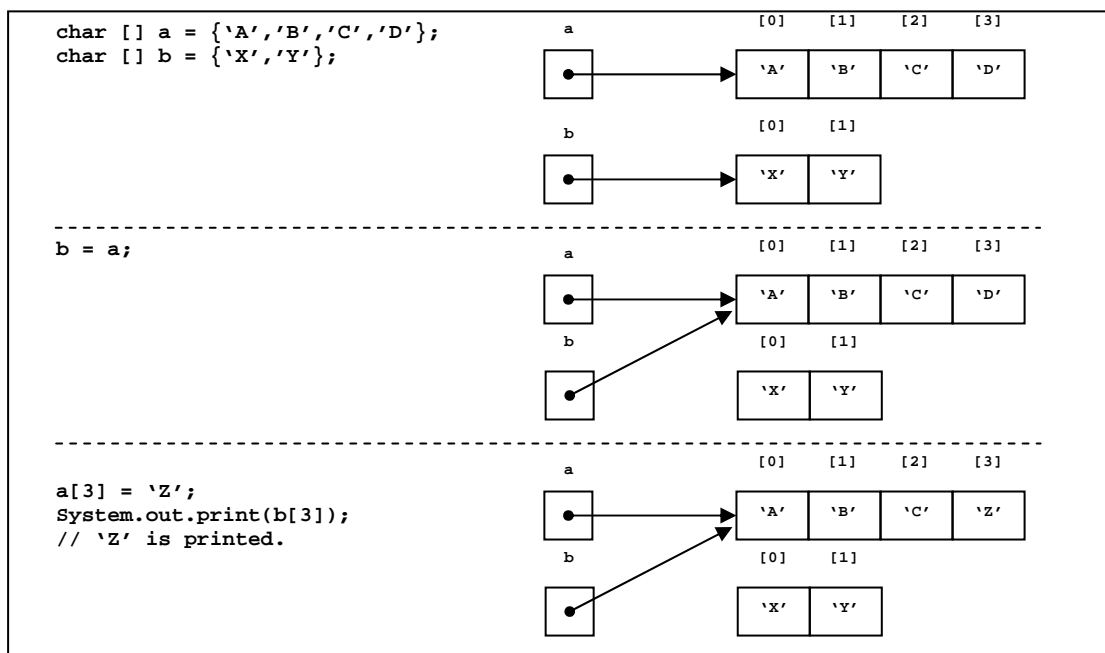
Both **a** and **b** are variables referring to arrays of **char**. On the last line, **b** is made to point to the same array as **a** using the assignment operator. Notice that the fact that **a** and **b** used to refer to an array of **char** with different lengths does not matter as long as they are both arrays of the same data type.

The following code segment is invalid due to the conflicting array data types.

```

int [] k = {1,2};
double [] j = {1.0,2.0,3.0};
j = k;
    
```

Assigning an array variable with another array variable makes the former array variable refer to the same array as the later one. Consider the following picture.



In the above example, the statement **b=a;** makes **b** refers to the same array as **a**. Therefore, **b[3]** is actually the same value as **a[3]**.



Arrays and Methods

Just like variables of other data types, array variables can be used as a parameter in methods' argument lists. One thing, you need to keep in mind is that the actual value that is kept in an array variable is the reference to its associated array stored somewhere in the memory. Therefore, when an array variable is used as a input parameter to a method, that reference is copied to the corresponding array variable defined in the head of that method's definition.

Consider the following example.

```
public class ArraysAndMethods
{
    public static void main(String[] args)
    {
        int [] a = {100,101,102,103};
        int k = 100;
        printArrayValues(a);
        System.out.println("k = "+k);
        someMethod(k,a);
        printArrayValues(a);
        System.out.println("k = "+k);
    }
    public static void someMethod(int k,int [] b){
        System.out.println("-----In the method.");
        k = 0;
        for(int i=0;i<b.length;i++) b[i]=0;
        printArrayValues(b);
        System.out.println("k = "+k);
        System.out.println("---Going out of the method.");
    }
    public static void printArrayValues(int [] a){
        for(int i=0;i<a.length;i++)
            System.out.print(a[i]+",");
        System.out.println();
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:>javac ArraysAndMethods.java
C:>java ArraysAndMethods
100,101,102,103,
k = 100
0,0,0,0,
k = 0
---Going out of the method.
0,0,0,0,
k = 100
C:>
```

From the above example, notice the values of the array variable **a** and the **int** variable **k** before and after *someMethod()* is called. As we have discussed in the last chapter, the value of **k** in *main()* does not change when the value of the variable **k** in *someMethod()* is changed. (If this is not clear to you, go back and consult Chapter 8.) However, the value of the integers in the array associated with **a** are changed since the array variable **b** in *someMethod()* refers to exactly the same array as **a**. Therefore, **b[i]**, for each **i** equals 0 to 3, is the same value as **a[i]** in *main()*.

An array can also be used as a returned value from a method. To do so, the return type must be specified in the method header appropriately as it is done with the values of other data types. To indicate that an array of a specific data type will be returned from a method, the return type must be the data type of the array element followed by square brackets.

For example,

```
public static int [] f(){
    // Body of f()
}
```

indicates that *f()* returns an array of **int**.



The program listed below shows an example of a method that returns an array of `double`. The method receives two input parameters and returns an array containing d^n , where d is the first input parameter and n are integers running from 0 to one less than the `int` value of the second input parameter.

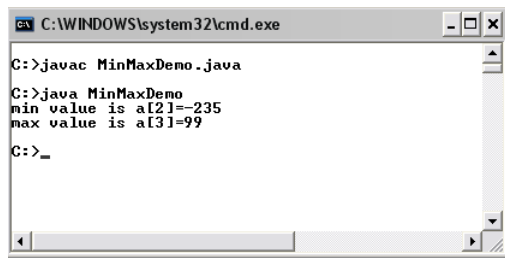
```
public class PowerSeries
{
    public static void main(String[] args)
    {
        double d = 2.0;
        int k = 5;
        double [] dn = genPowerSeries(d,k);
        for(int i=0;i<dn.length;i++)
            System.out.println("dn["+i+"]="+dn[i]);
    }
    public static double [] genPowerSeries(double d,int k){
        double [] dn = new double[k-1];
        dn[0] = 1;
        for(int i=1;i<k-1;i++)
            dn[i] = dn[i-1]*d;
        return dn;
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:>javac PowerSeries.java
C:>java PowerSeries
dn[0]=1.0
dn[1]=2.0
dn[2]=4.0
dn[3]=8.0
C:>
```

Example: Finding Maximum and Minimum Values

The following Java program shows an example of methods that find the maximum and minimum values of an array.

```
public class MinMaxDemo
{
    public static void main(String[] args)
    {
        int [] a = {-128,65,-235,99,0,26};
        int minIdx = findMinIdx(a);
        int maxIdx = findMaxIdx(a);
        System.out.println("min value is a["+minIdx+"]="+a[minIdx]);
        System.out.println("max value is a["+maxIdx+"]="+a[maxIdx]);
    }
    public static int findMinIdx(int [] a){
        int k, minIdx=0;
        for(k=1;k<a.length;k++){
            if(a[k]<a[minIdx])
                minIdx = k;
        }
        return minIdx;
    }
    public static int findMaxIdx(int [] a){
        int k, maxIdx=0;
        for(k=1;k<a.length;k++){
            if(a[k]>a[maxIdx])
                maxIdx = k;
        }
        return maxIdx;
    }
}
```



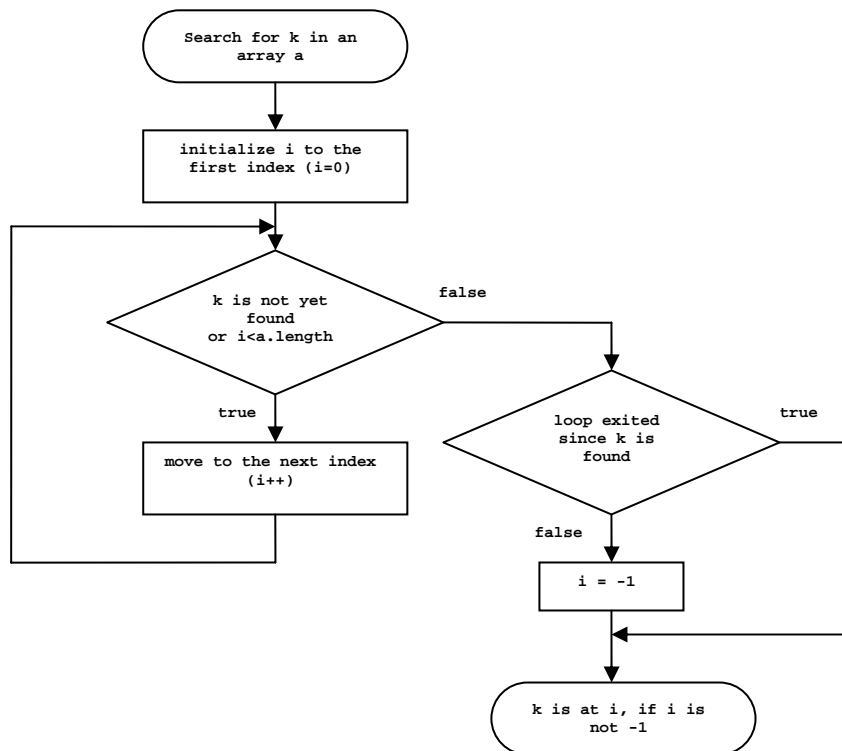
findMinIdx() and *findMaxIdx()* defined on line 10 and line 18 are used for finding the index of the minimum and the maximum values in an array input to the methods. The minimum value is located by first assuming that the element at the 0th index is the minimum value, then iterating through each element and compare each value with the minimum value. If any element is smaller than the current minimum value, replace the minimum value with that element. This way, when the comparison has been performed with every element in an array, the minimum value obtained in the end is the actual minimum value in that array. The maximum value is located using a rather similar procedure.

Both the minimum and maximum values can be found using a single method, if we let the method return an array containing the indexes of the two values. Each index is placed in each position of the returned array. This is shown in the program listed below. In this program, *findMinMax()* returns an array of two integers in which the first position contains the index of the minimum value and the second position contains the index of the maximum value.

```
public class MinMaxDemo2
{
    public static void main(String[] args)
    {
        int [] a = {-128,65,-235,99,0,26};
        int [] idx = new int[2];
        idx = findMinMaxIdx(a);
        System.out.println("min value is a["+idx[0]+"="+a[idx[0]]);
        System.out.println("max value is a["+idx[1]+"="+a[idx[1]]);
    }
    public static int [] findMinMaxIdx(int [] a){
        int k, minIdx=0, maxIdx=0;
        for(k=1;k<a.length;k++){
            if(a[k]<a[minIdx])minIdx = k;
            if(a[k]>a[maxIdx])maxIdx = k;
        }
        int [] idx = {minIdx,maxIdx};
        return idx;
    }
}
```

Sequential Search

We usually need an ability to search for data of a particular value in an array. Sequential search is a search algorithm that operates by checking every element of an array one at a time in sequence until a match is found. A flow diagram of sequential search can be shown in the picture below.



Suppose we want to search for a value k in an array a , we start checking at $a[i]$ when $i=0$. If $a[i]$ is not k , we increase i by one and repeat the checking. The process is iterated until $a[i] = k$, or until there is no more element in a (i.e. i exceeds $a.length-1$). Once the loop is exited, we have to check whether it is exited because k is found or there are no more elements to search. If the former case happens, i will still be less than $a.length$. If the latter case happens, i will be greater than or equal to $a.length$. In this case, we assign -1 to i to indicate that k is not found in a .

Below is an example of sequential search implementation in a method.

```

public class SeqSearchDemo
{
    public static void main(String[] args)
    {
        int [] a = {99,105,86,34,108,25,11,96};
        System.out.println("a={99,105,86,34,108,25,11,96}");
        System.out.println("86 is found at a["+seqSearch(a,86)+"]");
        System.out.println("96 is found at a["+seqSearch(a,96)+"]");
        System.out.println("0 is found at a["+seqSearch(a,0)+"]");
    }
    public static int seqSearch(int [] a, int k){
        int i = 0;
        int len = a.length;
        while(i<len && a[i]!=k){
            i++;
        }
        if(i>=len) i=-1;
        return i;
    }
}
    
```

```

C:\WINDOWS\system32\cmd.exe
C:>javac SeqSearchDemo.java
C:>java SeqSearchDemo
a={99,105,86,34,108,25,11,96}
86 is found at a[2]
96 is found at a[7]
0 is found at a[-1]
C:>
    
```

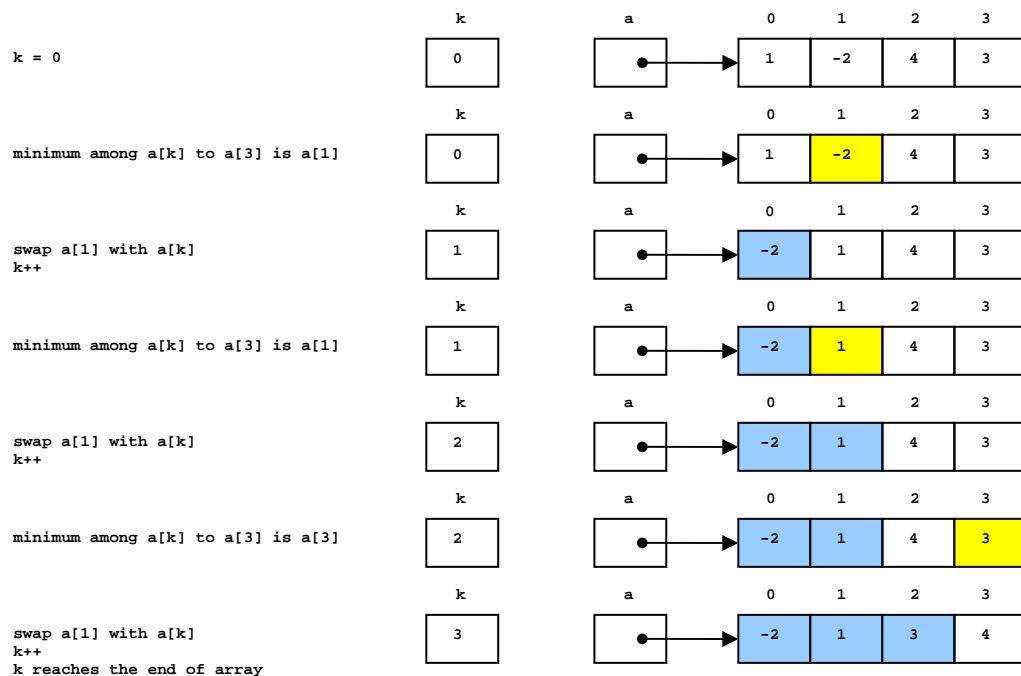


Selection Sort

Sometimes we need to sort the values appeared in an array. Algorithms performing the task are called sorting algorithms. Probably the most intuitive sorting algorithm is selection sort. To sort the values in an array increasingly, selection sort works as follows:

1. Let k be the first position of the array (i.e. $k = 0$).
2. Find the minimum value from the k^{th} position to the last position.
3. Swap the minimum value with the value in the k^{th} position.
4. Increase k by one.
5. Repeat step 2 to step 4 until k reaches the end of the array.

Let's look at an illustrated example of the sorting of an array $a = \{1, -2, 4, 3\}$.



Below is an example of selection sort implementation in a method.

```
public class SelectionSortDemo
{
    public static void main(String[] args)
    {
        double [] a = {6.0,5.9,-10.5,-8,1.3};
        for(int i=0;i<a.length;i++)
            System.out.print(a[i]+" ");
        System.out.println();
        selectionSort(a);
        for(int i=0;i<a.length;i++)
            System.out.print(a[i]+" ");
        System.out.println();
    }
    public static void selectionSort(double [] a){
        int k = 0, minIdx;
        while(k<a.length-1){
            minIdx = findMinIdx(a,k);
            swapElement(a,k,minIdx);
            // continue on the next page
        }
    }
}
```

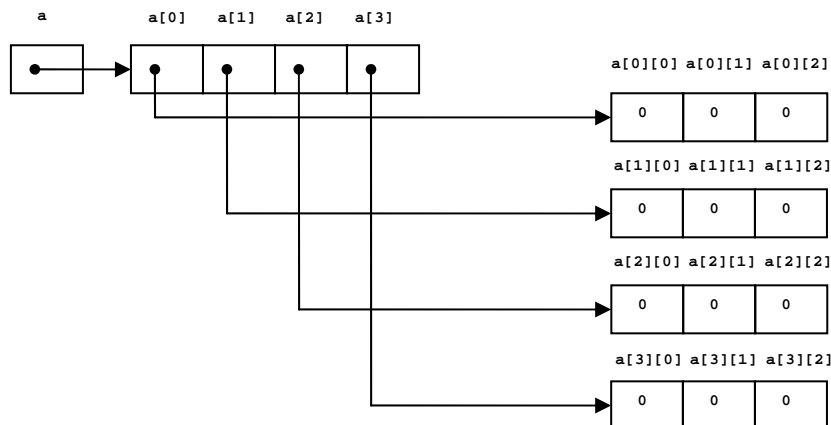


```

        k++;
    }
}
public static int findMinIdx(double [] a,int k){
    //This method finds the index of the minimum value
    //among a[k] to a[a.length-1]
    int minIdx = k;
    for(int i=k+1;i<a.length;i++){
        if(a[i]<a[minIdx]) minIdx = i;
    }
    return minIdx;
}
public static void swapElement(double [] a,int i,int j){
    double temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
}
    
```

Multi-dimensional Arrays

An elements contained in an array can be another array itself. An array in which each element is another array is called *multi-dimensional array*. The picture shown below illustrates an array of arrays of integers, or a two-dimensional array of integers.



A multi-dimensional array of a certain data type is declared by inserting pairs of [] after the data type specification. The number of [] equals the dimension of the array. For examples,

```

String [][] arrayOfString;
double [][][] a3DArray;
    
```

The first statement declares a variable named `arrayOfString` as a two-dimensional array of `String`. The other statement declares another variable named `a3DArray` as a three-dimensional array of `double`.



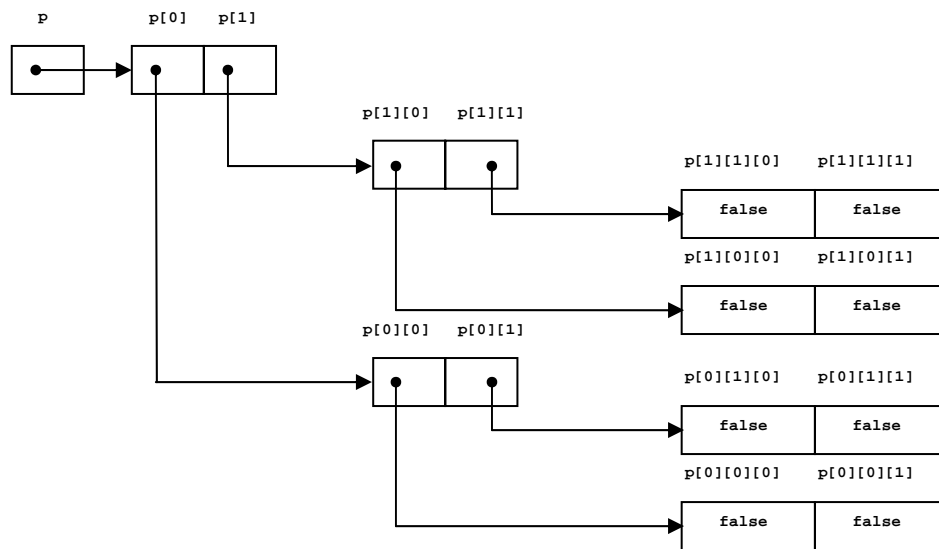
The following statements show how to declare and initialize multi-dimensional arrays with default values according to their data types.

```
int [][] k = new int[3][5];
boolean [][][] p = new boolean[2][2][2];
```

The first statement declares a variable **k** and assigns to it a reference to an array of length 3, each of whose elements is a reference to an array of five integers. All elements in the arrays of integers are initialized to 0.

The second statement declares a variable **p** and assigns to it a reference to an array of length 2, each of whose elements is a reference to a two-dimensional array of **boolean** value of the size 2 x 2. All **boolean** values are initialized to **false**.

The picture below shows an illustration of the variable **p**.



Initializer Lists for Multi-dimensional Arrays

Nested initializer lists can be used to initialize a multi-dimensional array with values other than the default value. For example, the statement:

```
int [][] k = {{1,2},{3,4,5},{8,10,12,14}};
```

initialize makes the variable **k** to refer to an array of three elements, where the first element is an array of two **int** values {1,2}, the second element is an array of three **int** values {3,4,5}, and the last element is an array of four **int** values {8,10,12,14}.

To access elements in **k**, we use these indexing mechanisms:

k : refers to the whole two-dimensional array of **int**.
k[0] : refers to the array {1,2}.
k[1] : refers to the array {3,4,5}.
k[2] : refers to the array {8,10,12,14}.
k[i][j] : refers to the j^{th} element of **k[i]**.

E.g. **k[0][1]** equals 2, **k[1][0]** equals 3, and **k[2][3]** equals 14 etc.



The following statement shows an example of using an initializer list to initialize a three-dimensional array of *String*.

```
String [][][] s = {
    {{ "Hamburg", "Berlin", "Munich"}, {"Paris", "Dijon"}},
    {{ "Hanoi", {"Bangkok", "Chiang Mai"}}
};
```

Being initialized this way,

s : refers to the whole three-dimensional array of *String*.
s[0] : refers to the two-dimensional array {{ "Hamburg", "Berlin", "Munich"}, {"Paris", "Dijon"}}.
s[1] : refers the two-dimensional array {{ "Hanoi", {"Bangkok", "Chiang Mai"}}}
s[0][0] : refers to the array { "Hamburg", "Berlin", "Munich".
s[0][1] : refers to the array { "Paris", "Dijon".
s[1][0] : refers to the array { "Hanoi".
s[1][1] : refers to the array { "Bangkok", "Chiang Mai".
s[i][j][k] : refers to the k^{th} element of **s[i][j]**.

E.g. **s[0][1][1]** equals "Dijon", **s[1][1][1]** equals "Bangkok", and etc.

Let's look at an example program demonstrating the indexing of multi-dimensional array. Nested *for* loops are used for browsing through each level of the three-dimensional array of *int* named **a**. Pay attention to the length of **a[i]**, **a[i][j]** and the value of each *int* value **a[i][j][k]**.

```
public class ArrayLengthDemo
{
    public static void main(String[] args)
    {
        int [][][] a
        = {{{1,2,3},{4,5},{6}},{{7,8},{9,10,11,12,13}}};
        System.out.println("a.length = "+a.length);
        for(int i=0;i<a.length;i++){
            System.out.println("a["+i+"].length = "+a[i].length);
            for(int j=0;j<a[i].length;j++){
                System.out.print("a["+i+"]["+j+"].length = ");
                System.out.println(a[i][j].length);
                for(int k=0;k<a[i][j].length;k++){
                    System.out.print("a["+i+"]["+j+"]["+k+"]=");
                    System.out.println(a[i][j][k]);
                }
            }
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac ArrayLengthDemo.java
C:\>java ArrayLengthDemo
a.length = 2
a[0].length = 3
a[0][0].length = 3
a[0][0][0]=1
a[0][0][1]=2
a[0][0][2]=3
a[0][1].length = 2
a[0][1][0]=4
a[0][1][1]=5
a[0][2].length = 1
a[0][2][0]=6
a[1].length = 2
a[1][0].length = 2
a[1][0][0]=7
a[1][0][1]=8
a[1][1].length = 5
a[1][1][0]=9
a[1][1][1]=10
a[1][1][2]=11
a[1][1][3]=12
a[1][1][4]=13
C:\>
```



Example

We would like to write a program that can calculate the result of A^n where A is a square matrix and n is a positive integer.

Problem definition: The program needs to calculate the n^{th} power of a matrix whose elements, as well as the value of n , are specified by the user.

Analysis: Elements of A and the power n should be read from keyboard. The result of the calculation should be shown as the output on screen.

Design:

- The user must specify the size of the square matrix A via keyboard. The dimension will be kept in an `int` variable named `dim`.
- Elements of A will be kept in a two-dimensional array named `a`. With the size of A known, the program should iteratively prompt the user to input elements of A one by one. Each element will be stored in `a`.
- The program prompts the user to enter n via keyboard. The input will be kept in `n`.
- A^k can be calculated from $A^{k-1} \times A$ for $k = 2, 3, 4, \dots, n$. That means A^n can be calculated by iteratively multiply A with the result of the multiplication prior to the current iteration. After each iteration of the multiplication, use another two-dimensional array named `b` to store A^{k-1} . Also, use another two-dimensional array named `c` to store the result.
- To calculate $C = B \times A$. Use the relation:
$$c_{ij} = \sum_{k=1}^n b_{ik} a_{kj}$$
- Show each element of A^n on screen.

Implementation:

```
import java.io.*;
public class MatrixPower
{
    public static void main(String[] args) throws IOException
    {
        // Declare variables
        double [][] a, b, c;
        int dim, n;
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        // Read matrix size
        System.out.print("Enter matrix size:");
        dim = Integer.parseInt(stdin.readLine());
        // Create a, b, c and read each element of a
        a = new double[dim][dim];
        b = new double[dim][dim];
        c = new double[dim][dim];
        for(int i=0; i<dim; i++){
            for(int j=0; j<dim; j++){
                System.out.print("a"+(i+1)+"(j+1)+"=");
                a[i][j] = Double.parseDouble(stdin.readLine());
            }
        }
        // Continue on the next page
    }
}
```



```
// Read power n
System.out.print("Enter n:");
n = Integer.parseInt(stdin.readLine());
// Perform raising a to the n th power
b = a;
for(int k=1;k<n;k++){
    c = multSqMatrices(b,a);
    b = c;
}
// Show the result on screen
showMatrix(c);
}

public static double [][] multSqMatrices(

    double [][] b,double [][] a){
    int dim = b.length;
    double [][] c = new double[dim][dim];
    for(int i=0;i<dim;i++)
        for(int j=0;j<dim;j++){
            for(int k=0;k<dim;k++){
                c[i][j] = c[i][j]+b[i][k]*a[k][j];
            }
        }
    return c;
}

public static void showMatrix(double [][] c){
    int nRows = c.length;
    int nCols = c[0].length;
    for(int i=0;i<nRows;i++){
        for(int j=0;j<nCols;j++){
            System.out.print(c[i][j]+"\\t");
        }
        System.out.println();
    }
}
}
```

```
C:\WINDOWS\system32\cmd.exe
C:\>javac MatrixPower.java
C:\>java MatrixPower
Enter matrix size:3
a11=2
a12=1
a13=0
a21=1
a22=3
a23=1
a31=0
a32=2
a33=4
Enter n:3
15.0    22.0    9.0
22.0    55.0    40.0
18.0    80.0    86.0
C:\>_
```



Exercise

1. Show how to declare variables corresponding to the following:
 - a. An array of `int`.
 - b. An array of `boolean`.
 - c. An array of `String`.
 - d. An array of arrays of `double`.
 - e. A two-dimensional array of *Rectangle*.
 - f. A three-dimensional array of `char`.
2. Declare and initialize arrays corresponding to the following:
 - a. An array of 20 `int` values.
 - b. An array of `double` where its length equals the length of an array of `int` called `b`.
 - c. An array of `boolean` where its first three values are `true` and the other two are `false`.
 - d. An array of *String* containing the names of the seven days in a week.
 - e. An array containing an array of 1.0, 2.5, 3.0, and another array of 2.5, 3.0, 4.5.
 - f. A two-dimensional array suitable for representing a identity matrix of the size 3×3

3. Is it valid to create an array where each element is an array whose length is different from the lengths of other elements in the same array.

4. Determine the output of the following code segment.

```
int [] a = new int[10];
a[1] = 2;
a[a.length-1]=8;
for(int i=0;i<a.length;i++){
    System.out.print(a[i]+"\\t");
}
```

5. Determine the output of the following code segment.

```
int [] a = new int[10];
for(int i=0;i<a.length-1;i++){
    a[i] = a[+i]+i;
}
for(int i=0;i<a.length;i++){
    System.out.print(a[i]+"\\t");
}
```

6. Determine the output after *main()* is executed.

```
public static void main(String[] args) {
    int k = 1;
    int [] a = {10,11,12,13,14};
    f(k,a);
    System.out.println(k);
    showArrayContent(a);
}
public static void f(int k,int [] b){
    if (k >= b.length) return;
    for(int i=k;i<b.length;i++){
        b[i]=b[b.length-i];
    }
    k = 0;
}
public static void showArrayContent(int [] a){
    for(int i=0;i<a.length;i++) System.out.println(a[i]);
}
```




7. Write a method that receives an array of `int` and returns the sum of every elements in the array.

8. Write a method that evaluates the value of a polynomial function $p(x)$ at given values of x . The function is of the form $c_n x^n + c_{n-1} x^{n-1} + \dots + c_0$. The method header is given as:

```
public static double [] p(double [] x, double [] coeff)
```

The k^{th} element in `coeff` is corresponding to the c_k . Each element in the returned array of `double` is the value of $p(x)$ evaluated at x being the value of the element in `x` at the same index.

9. Write a method called `findRep()` whose header follows:

```
public static int findRep(int [] a, int target, int nRep)
```

The method finds whether `a` contains `nRep` consecutive elements whose values equal `target` or not. If so, it returns the position of the first element whose value equals `target`. Otherwise, it returns -1. For example, if `a` contains 6, 8, 9, 9, 9, 3, 9, 2, 0, `findRep(a,9,2)` and `findRep(a,9,3)` return 2, while `findRep(a,9,4)` and `findRep(a,10,1)` return -1.

10. Write a method that sorts an input array of `int` in place (i.e. the elements in the original array are sorted. There is no new array resulted from the sorting). There must be another parameter determining whether to sort this array increasingly or decreasingly.

11. Every methods in this problem receives two arrays of `int` as their input parameters.

- Write `combine()` which returns a new array whose elements are taken from both input arrays and their orders are preserved starting from the elements from the first input array followed by the ones from the second.
- Write `union()` which returns a new array whose elements are unique elements taken from both input arrays. The elements of the output array should be sorted increasingly.
- Write `intersect()` which returns a new array where every elements in the array must be unique and appear in both input arrays. The elements of the output array should be sorted increasingly.
- Write `subtract()` which returns a new array whose elements are unique and appear in the first input array but not in the second one. The order of the elements of the output array must follow the order of the first input array.
- Write `xor()` which returns a new array whose elements are unique and appear in either one of the input arrays but not both. The elements of the output array should be sorted increasingly.

12. Write a method that receives an array of `double` and returns an array of `double` in the form $\{a,b,c\}$ where a is the biggest value of the input array, b is the second biggest value and c is the third biggest value. The values of a , b , and c must not repeat one another. If the input array contains less than three elements, returns a new array whose elements represent the decreasingly sorted version of the input array.

13. Suppose that two arrays are said to be equal if they have similar lengths and every elements in the same positions of the two arrays are equal. Write a method called `isEqual()` which returns `true` if its two input arrays equal and `false` otherwise.



14. Repeat the previous problem in the case where positions do not matter, i.e. the two arrays are said to be equal if their elements form sets with similar members.
15. Write a method that receives an array of *String*, together with a *String* and returns **true** if there is at least one element of the input array that contains or equals the other *String* input. Otherwise, it returns **false**.
16. Explain why the following code segment lead to a failed compilation.

```
final double RANGE = 200;
int step = 12, k=6;
int [][] a = new int[(int)(RANGE/step)][k];
for(int i=0;i<a.length;i++)
    a[i] = new int[2][2];
```

17. What is the output of the following code segment?

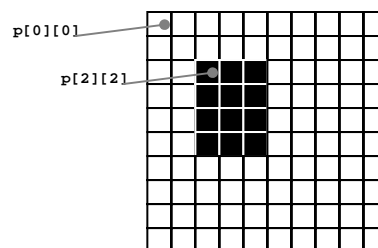
```
String [][][] x = new String[5][6][7];
System.out.println(x.length*x[2].length);
```

18. One way to represent a black-and-white image is to store **boolean** values in a two-dimensional array, **p**. **p[i][j]** is **true** if the pixel in the i^{th} row and j^{th} column is black. Similarly, it is **false** if the corresponding pixel is white.

Write a method:

```
public static boolean drawRect(boolean p, int x, int y,
                               int width, int height, int ink)
```

If **ink** equals 0, the method draws a white rectangle whose topleft corner locating at **p[x][y]**. Its width and height are the values of **width** and **height** respectively. If **ink** equals 1, the method draws a black rectangle instead. If **ink** equals -1, the drawing is done in a way that every pixels of the rectangle drawn by the method are toggled from white to black, or black to white. The following array demonstrate an example of **p** after performing **drawRect(p,2,2,3,4,1)** on an all-white array **p** whose size is 10×10.



The method returns **true** if the drawing is performed successfully. It returns **false** and does not perform any drawing when at least one of the following situations take place:

- The value specified by **ink** is not -1, 0, or 1.
- The specified topleft corner does not fall in the vicinity of **p**, i.e. the value of **x**, **y** or both is not in the range of the array.
- The rectangle exceeds the vicinity of **p**.

19. Repeat the previous problem. However, this time, the method should attempt to draw the specified rectangle even though the topleft corner does not fall in the vicinity of **p** or the whole rectangle does fit in **p**.



For example, if **p** and **q** are of the size 8×8 and initially all zeros. **drawRect(p,5,5,2,5,1)** and **drawRect(q,-2,-2,4,4,1)** would result in the following arrays.



20. In a seat reservation program of an airline, the seating chart of an airplane is represented using a two-dimensional array, **seats**. The array **seats[i][j]** contains the name of the passenger who has reserved the seat number **j** in the **i**th row, where **j** is 0 for the seat number A, **j** is 1 for the seat number B, and so on. **seats[i][j]** stores **null** if the seat is vacant. Note that different airplanes have different numbers of rows. However, assume that the seats in a row always set in the following setting.

Write methods, that have one of their input arguments being the array **seats**, for performing the tasks in the following items. Decide on the names, their input arguments, and their returned values appropriately.

- Adding a passenger name to a selected seat. It returns **true** if the operation is successful, and return **false** if the selected seat is not empty. Given that the selected seat is specified in the form of a *String* in the form: "[row number]-[seat number]", such as "1-A", "25-E", and "36-I".
- Removing the passenger at a specified seats.
- Searching for the seat reserved by a passenger by his/her name. The method returns the *String* representing the seat location or "**not found**" if there is no passenger of the given name.
- Counting the number of seats available in each row.
- Searching for available *n* consecutive seats in a row. The method returns the *String* representing the left-most seat location of the available *n* consecutive seats in the front-most row that has such an availability. The user must have a choice whether seats across an aisle are considered adjacent or not.
- Randomly relocating passengers in the seating chart. Each passenger must be assigned a seat not conflicting with other passengers. (This method is not going to be useful for any functioning airlines!)