CHAPTER 8

Arrays and Strings

8.1 Introduction

Sometimes you need to deal with multiple number of similar data items. For instance, you might need to handle names and marks of 25 students. For such purposes, Java provides two reference data types namely *arrays* and *strings*.

In This Chapter

8.2 Arrays

To deal with a collection of similar data items, Java offers a reference data type called arrays. An array can hold several values of same type.

Arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Arrays are a way to group a number of items into a larger unit. Arrays can have data items of simple types like **int** or **float**, or even of user-defined types like structures and objects.

8.3 Types of Arrays

Arrays are of different types : (*i*) *one-dimensional arrays*, comprised of finite homogeneous elements. (*ii*) *multidimensional arrays*, comprised of elements, each of which is itself an array. A two-dimensional array is the simplest of multidimensional arrays. However, Java allows arrays of more than two dimensions.

An **Array** is a collection of variables of the same type that are referenced by a common name.

8.3.2 Two-Dimensional Arrays

The simplest form of a multidimensional array, the two-dimensional array, is an array having single-dimension arrays as its elements. The general form of a two-dimensional array declaration in Java is as follows :

```
type array-name [ ] [ ] = new type [row] [columns] ;
```

or as

```
type [ ] [ ] array-name = new type[rows] [columns] ;
```

where *type* is the base data type of the array having name *array-name* ; *rows*, the first index, refers to the number of rows in the array and *columns*, the second index, refers to the number of columns in the array.

Following declaration declares an **int** array *sales* of size 5, 12.

```
int sales [ ] [ ] = new int [5] [12] ;
```

or

```
int [ ] [ ] sales = new int [5] [12] ;
```

The array **sales** have 5 elements *viz*, sales [0], sales [1], sales [2], sales [3] and sales [4] ; each of these elements is itself an **int** array with 12 elements. The elements of **sales** are referred to as *sales* [0][0], *sales* [0][1], ..., *sales* [0][11], *sales* [1][0], *sales* [1][1],..... and so forth. The following program (8.2) reads sales of 5 salesmen in 12 months.

NOTE

While writing Java programs involving arrays, make sure to design test condition that ensures that only valid array-indices are used in the program. This will save you from array-index related runtime errors.

Program 8.2

Program to read sales of 5 salesmen in 12 months and to print total sales made by each salesman.

```
import java.util.Scanner ;
class Salesman  {
    public static void main(String args[ ]) {
        Scanner in = new Scanner(System.in) ;
        int sales[ ][ ] = new int[5][12] ;
        int i, j, total ;
        try {
            for(i = 0; i < 5 ; i++) {
                total = 0 ;
                System.out.println("Enter sales of salesman " + (i + 1)) ;
                for(j = 0; j < 12 ; j++)    {
                    System.out.print("Month " + (j + 1) + " : ") ;
                    sales[i][j] = in.nextInt( ) ;
                    total += sales[ i ][ j ] ;
                }
                System.out.println("Total sales of salesman " + (i + 1)  + " = " + total) ;
            }
            catch (Exception e) { }
    }
}
```

A sample program run for above program for 2 salesmen's sales in 4 months, is shown below (*i.e.*, we changed condition for *i loop* as i < 2 for *j loop* as j < 4) :

```
Enter sales of salesman 1
Month 1 : 5000
Month 2 : 6000
Month 3 : 7000
Month 4 : 4000
Total sales of salesman 1 = 22000
Enter sales of salesman 2
Month 1 : 5000
Month 2 : 4000
Month 3 : 4400
Month 4 : 6000
Total sales of salesman 2 = 19400
```

Memory Representation

Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second index indicates the column. This means the second index (*i.e., column*) changes faster than the first index (*i.e., row*) when accessing the elements in the array in the order in which they are actually stored in memory. If you have declared an array **pay** as follows :

```
short[ ] [ ] pay = new short [5] [7] ;
```

it will be having $5 \times 7 = 35$ elements which will be represented in memory as shown below in Fig. 8.2.

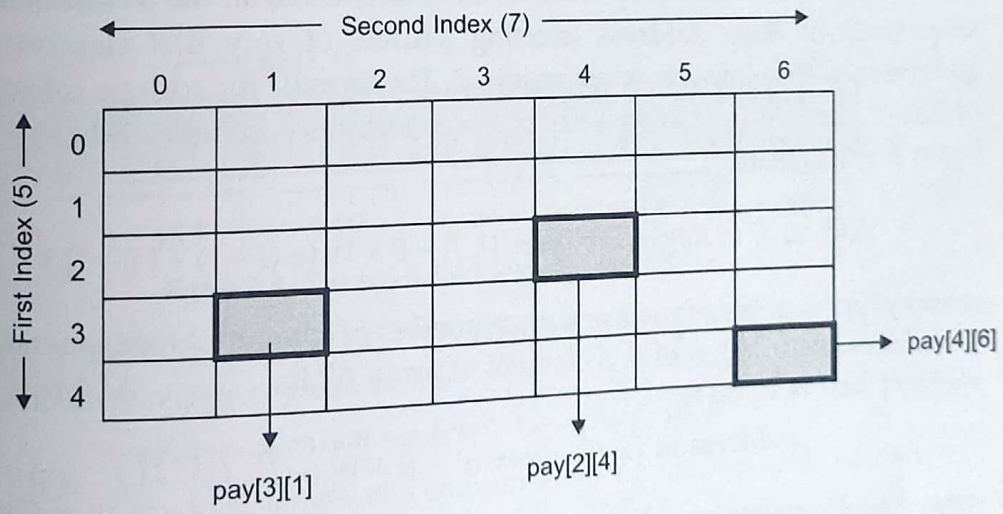


Figure 8.2 A two-dimensional array namely pay [5] [7] in memory.

The amount of storage required to hold a two-dimensional array is also dependent upon its base type, number of rows and number of columns. The formula to calculate total number of bytes required by a two-dimensional array is given below :

```
total bytes = number-of-rows × number of columns × size of base type
```

For instance, the above declared array **pay[5][7]** requires $5 \times 7 \times 2 = 70$ bytes as the base type *short*'s size is 2 bytes.

8.3.3 Implementation of Two-dimensional Array in Memory

While storing the elements of a 2-D arrays in memory, these are allocated contiguous locations. Therefore, a 2-D array must be *linearized* so as to enable their storage. There are two alternatives to achieve linearization *viz.*, *Row-major* and *column major*.

Row-Major Implementation

This linearization technique stores firstly the *first row* of the array, then the *second row* of the array, then the *third row*, and so forth. For example, array **Data** [1 : 5, 1 : 6] which logically appears as shown in Fig. 8.3(*a*), appears physically in row-major implementation as shown in Fig. 8.3(*b*).

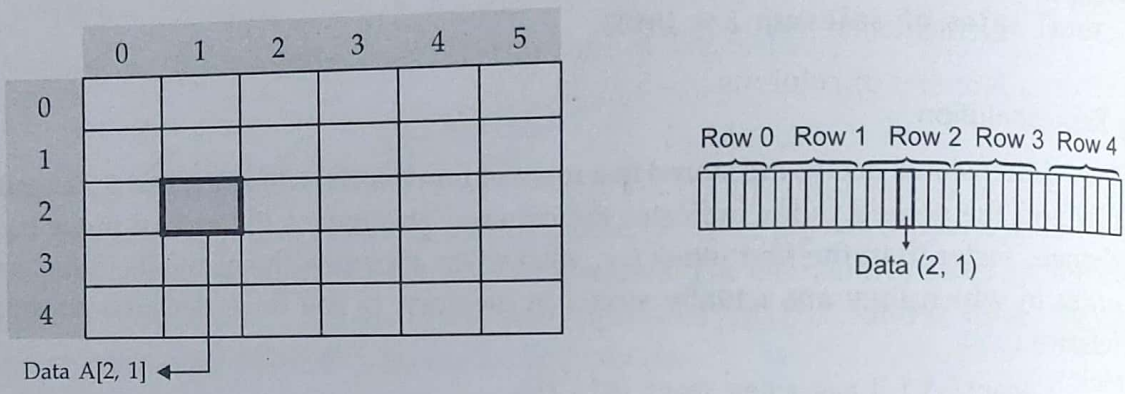


Figure 8.3 (*a*) Logical Implementation (*b*) Physical Row-major Implementation.

The computer does not keep track of the addresses of all the elements of the array but does keep track of **Base Address** (starting address of very first element) and calculates the addresses of the elements when required. The formula for address calculation (in Row-major order) of element [*I, J*] in an array of $m \times n$ elements with base address B and element size W bytes is given below :

$$\textit{Address of in Row-major order } [I, J] = B + W\,[n\,(I - 1) + [\,J - 1]\,]$$

where m are number of rows and n are number of columns. More generalised form of formula for address calculation of [*I, J*] element of array AR $[L_r : U_r, L_c : U_c]$ with base address B and element size W bytes is

$$\text{Address of } [I, J]^{\text{th}} \text{ element} = B + W\,(n(I - L_r]\ + [J - L_c]]$$

where n is number of columns *i.e.*, length of array $[L_c : U_c]$ *i.e.*, $U_c - L_c + 1$; L_r is first row number, L_c is first column number.

Example 8.1 *A 2-D array defined as A [4.. 7, −1 .. 3] requires 2 words of storage space for each element. If the array is stored in Row-major form, calculate the address of A[6, 2] given the base address as 100 (one hundred).*

Solution. Base address B = 100

Element size W = 2 bytes

$L_r = 4$; $L_c = -1$; $I = 6$; $J = 2$

r, number of rows $= U_r - L_r + 1 = 7 - 4 + 1 = 4$

c, number of columns $= U_c - L_c + 1 = 3 - (-1) + 1 = 5$

Address [I, J] $= B + W(c(I - L_r) + (J - L_c))$

$= 100 + 2(5(6 - 4) + 2 - (-1))$

$= 100 + 2(5 \times 2 + 3) = 100 + 2(13)$

$= 100 + 26 = 126.$

Example 8.2 *X is 2-D array [10 × 5]. Each element of the array is stored in 2 memory locations. If X[1, 1] begins at address 150, find the location of X[3, 4]. Use the formula for calculation. (The arrangement is in row-major).*

Solution. Base address $B = 150$

Element size $W = 2$

Number of columns $n = 5$

$L_r = L_c = 1$

Location X[3, 4] $= 150 + 2(5(3 - 1) + (4 - 1))$ [Using $B + W(n(I - L_r) + (J - L_c))$]

$= 150 + 2(10 + 3)$

$= 150 + 26 = 176$

Column Major

This linearization technique stores first the first column, then the second column, then the third column, and so forth. Using column-major implementation, DATA (1:5, 1:6) shown in Fig. 8.3(*a*) would appear in the memory as shown below :

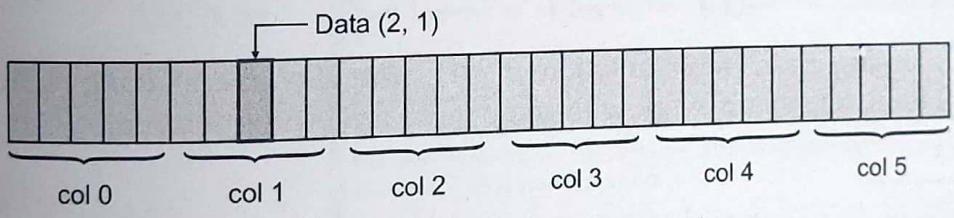


Figure 8.4 Column-Major Implementation.

The formula for address calculation of (I, J)th element in array $X[L_r : U_r, L_c : U_c]$ with base address B and element size w is given below :

$$Address\ (I, J) = B + w[r(J - L_c) + [I - L_r)]$$

where r is number of rows *i.e.*, length of a column array $L_r : U_r$ *i.e.*, $U_c - L_c + 1$.

Example 8.3 *Each element of an array A [−20..20, 10 ... 35] requires one byte of storage. If the array is stored in column major order beginning location 500, determine the location of A[0, 30].*

Solution. Base address B = 500

Element size W = 1 byte

r (number of rows) are $20 - (-20) + 1 = 41$

$L_r = -20 ; L_c = 10$

Address of A[0, 30] $= 500 + 1[(0 - (-20) + 41(30 - 10))]$

$= 500 + (20 + 41 \times 20) = 500 + 1[20 + 820] = 500 + 840 = 1340$

Example 8.4 *Each element of an array A [-15 ...20, 20 ...45] requires one byte of storage. If the array is stored in column major order beginning location 1000, determine the location of A[0, 40].*

Solution. Base address $B = 1000$

Element size W $= 1$ byte

Number of rows $= 20 - (-15) + 1 = 36$ (Using $U_r - L_r + 1$)

$L_r = -15,\ L_c = 20$

Address of $A[0, 40] = 1000 + 1[(0 - (-15)) + 36(40 - 20)]$

$= 1000 + 1[15 + 36 \times 20]$

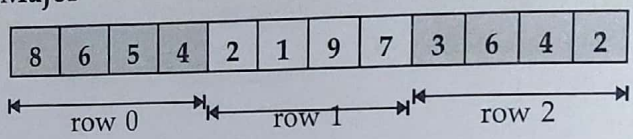
$= 1000 + 1[15 + 720] = 1000 + 735 = 1735.$

Example 8.5 *Given a two-dimensional array 3 × 4 as shown here :*

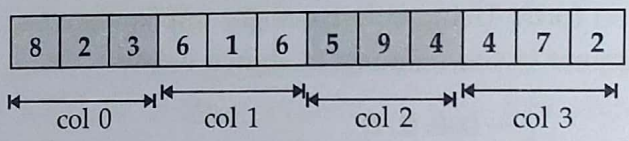
$$\begin{bmatrix} 8 & 6 & 5 & 4 \\ 2 & 1 & 9 & 7 \\ 3 & 6 & 4 & 2 \end{bmatrix}$$

How would array appear in memory, if : (i) Row-Major technique is implemented. (ii) Column-Major technique is implemented.

Solution. (*i*) **Row-Major**



(*ii*) **Column-Major**



After discussing basic representation of 1D and 2D arrays along with their memory representation, let us now work with some more array programming.

Program 8.3

You can generate random numbers in the range 0 to 1 using Math.random( ) method. If you want to generate random numbers in a range say 7(L) to 23(U) then you may use method as :

```
Math.random( ) * (U - L + 1) + L
```

where L is lower limit i.e., 7 ; U is upper limit i.e., 23

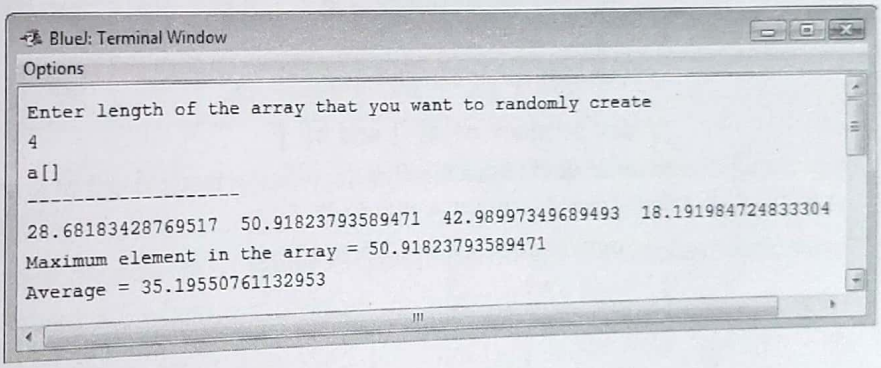
Using this formula, generate a 1D double type array in the range 10-50 after asking the user to specify its length. Then perform the following actions on the array

(i) display array elements (ii) find maximum element of the array (iii) compute array average

```
import java.util.Scanner ;
public class Arrays {
   public static void main(String[ ] args) {
      System.out.println("Enter length of the array that you want to randomly create") ;
      Scanner in = new Scanner(System.in) ;
      int N = in.nextInt() ;
      // initialize to random values between 10 and 50
      double[ ] a = new double[N] ;
```

```
        for (int i = 0 ; i < N ; i++) {
            a[i] = Math.random() * 41 + 10 ;
        }
        // print array values
        System.out.println("a[ ]") ;
        System.out.println("----------") ;
        for (int i = 0 ; i < N ; i++) {
            System.out.print(a[i]+" ") ;
        }
        System.out.println() ;

        // find the maximum
        double max = Double.NEGATIVE_INFINITY ; // it gives the min double value that can be stored
        for (int i = 0 ; i < N ; i++) {
            if (a[i] > max) max = a[i] ;
        }
        System.out.println("Maximum element in the array = " + max) ;

        //computing average
        double sum = 0.0 ;
        for (int i = 0 ; i < N ; i++) {
            sum += a[i] ;
        }
        System.out.println("Average = " + sum / N) ;
    }
}
```

The output produced would be



Program 8.4

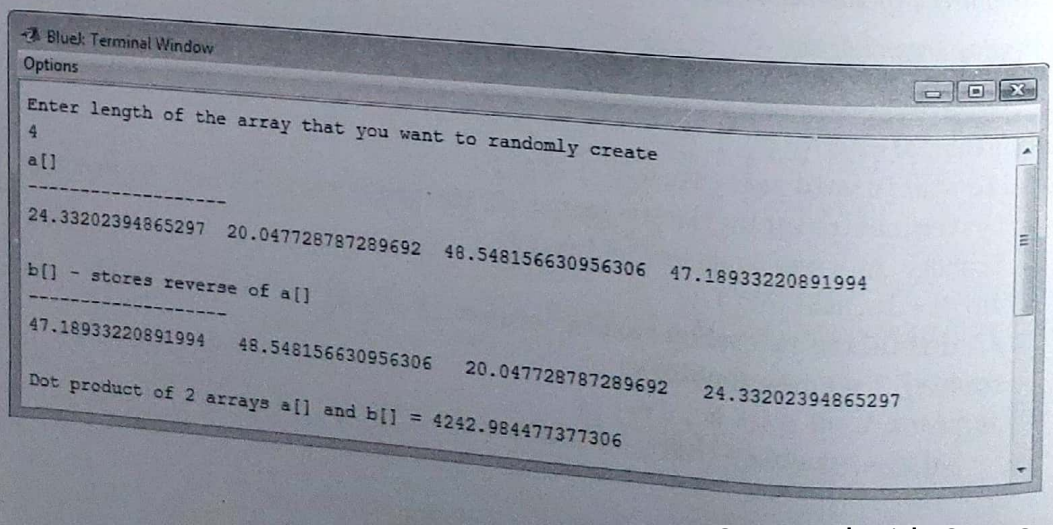
Create an array a in the same manner as you did in program 8.3. Then perform the following tasks on the array.

(i) copy array to another array b
(ii) reverse the second array
(iii) display second array
(iv) calculate dot product of two arrays as : $\Sigma a[i] * b[i]$

```
import java.util.Scanner ;
public class Arrays {
    public static void main(String[ ] args) {
        System.out.println("Enter length of the array that you want to randomly create") ;
        Scanner in = new Scanner(System.in) ;
        int N = in.nextInt() ;
        // initialize to random values between 10 and 50
        double[ ] a = new double[N] ;
        for (int i = 0 ; i < N ; i++) {
            a[i] = (double) (Math.random() * 41 + 10) ;
        }
```

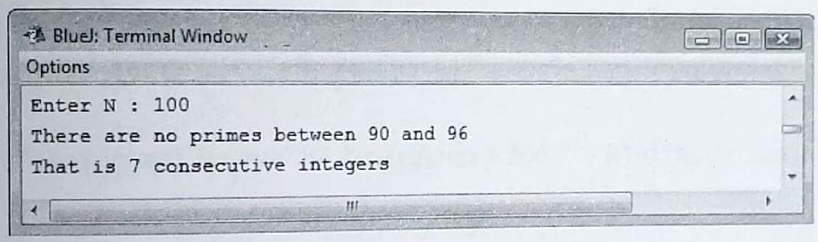
```
        // print array values
        System.out.println("a[ ]") ;
        System.out.println("————————-") ;
        for (int i = 0 ; i < N ; i++) {
            System.out.print(a[i]+"  ") ;
        }
        System.out.println() ;
        // copy to another array
        double[ ] b = new double[N] ;
        for (int i = 0 ; i < N ; i++) {
            b[i] = a[i] ;
        }
        // reverse the order of second array
        for (int i = 0 ; i < N/2 ; i++) {
            double temp = b[i] ;
            b[i] = b[N - i - 1] ;
            b[N - i - 1] = temp ;
        }
        // print array values of second array
        System.out.println() ;
        System.out.println("b[ ] - stores reverse of a[ ]") ;
        System.out.println("————————-") ;
        for (int i = 0 ; i < N ; i++) {
            System.out.print(b[i]+"  ") ;
        }
        System.out.println() ;
        // dot product of a[ ] and b[ ]
        double dotProduct = 0.0 ;
        for (int i = 0 ; i < N ; i++) {
            dotProduct += a[i] * b[i] ;
        }
        System.out.println("\nDot product of 2 arrays a[ ] and b[ ] = " + dotProduct) ;
    }
}
```

The output produced would be :



```
            if (isprime[i]) {
                for (int j = i ; i * j <= N ; j++)
                    isprime[i*j] = false ;
            }
        }
        // find longest consecutive sequence of integers with no primes
        int gap = 0 ;
        int bestgap = 0 ;
        int right = 0 ;
        for (int i = 2 ; i <= N ; i++) {
            if (isprime[i]) gap = 0 ;
            else gap++ ;
            if (gap > bestgap) { bestgap = gap ; right = i ; }
        }
        int left = right - bestgap + 1 ;
        System.out.println("There are no primes between " + left + " and " + right) ;
        System.out.println("That is " + bestgap + " consecutive integers") ;
    }
}
```

The output produced would be :



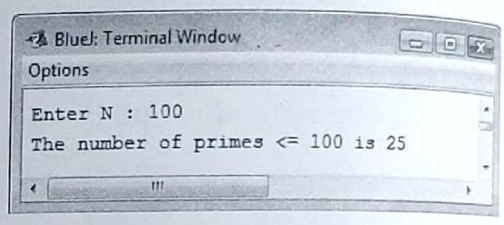
Program 8.7

Christian Goldback has conjectured for $N < 10^{14}$ that every even $N > 2$, can be represented as sum of two primes. For example, $18 = 5 + 13$.

Write a Java program that reads N which should be an even number > 2 and represents it as the sum of two primes.

```
import java.util.Scanner ;
public class Number {
  public static void main(String[ ] args) {
    Scanner in = new Scanner(System.in) ;
    System.out.print("Enter N : ") ;
    int N = in.nextInt() ;
    boolean[ ] isprime = new boolean[N] ;
    for (int i = 2 ; i < N ; i++)
       isprime[i] = true ;
    // determine primes < N using Sieve of Eratosthenes
    for (int i = 2 ; i * i < N ; i++) {
       if (isprime[i]) {
          for (int j = i ; i * j < N ; j++)
```

```
        // mark non-primes <= N using Sieve of Eratosthenes
        for (int i = 2 ; i*i <= N ; i++)  {
            // if i is prime, then mark multiples of i as nonprime
            // suffices to consider mutiples i, i + 1, ..., N/i
            if (isPrime[i]) {
                for (int j = i ; i*j <= N ; j++)  {
                    isPrime[i*j] = false ;
                }
            }
        }
        // count primes
        int primes = 0 ;
        for (int i = 2 ; i <= N ; i++) {
            if (isPrime[i]) primes++ ;
        }
        System.out.println("The number of primes <= " + N + " is " + primes) ;
    }
}
```



The output produced would be :

Program 8.6

Write a Java program to find the longest consecutive sequence of integers with no primes. The program should accept a number N and print out the largest block of integers between 2 and N with no primes.

Sample Input/Output :

```
Enter N      :   10
There are no primes between 8 and 10
That is 3 consecutive integers
Enter N      : 30
There are no prime between 24 and 28
That is 5 consecutive intgers.
```

NOTE

If in range 2...N, there are multiple longest sequences (each having same length), then you may print just one sequence.

Hint : *Use Sieve of Eratosthenes used in previous program.*

```
import java.util.Scanner ;
public class PrimeGap {
   public static void main(String[ ] args) {
       Scanner in = new Scanner(System.in) ;
       System.out.print("Enter N : ") ;
       int N = in.nextInt() ;
       boolean[ ] isprime = new boolean[N+1] ;
       for (int i = 2 ; i <= N ; i++)
           isprime[i] = true ;
       // determine primes < N using
       for (int i = 2 ; i * i <= N ; i++) {
```

Program 8.5

Write a program in Java to find the number of prime numbers <= N where accept N from user. Use the method Sieve of Eratosthenes (given below) to find prime numbers.

Sieve of Eratosthenes

To get the prime numbers in the range 2..N, you can use a simple and interesting method known as Sieve of Erotosthenes. It works as follows :

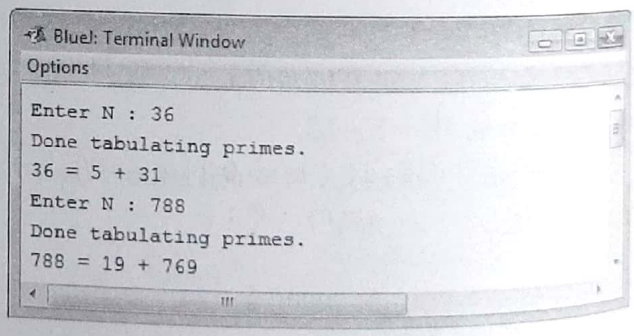
~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

Sequentially write down the integers from 2 to the highest number n you wish to include in the table. Cross out all numbers > 2 which are divisible by 2 (every second number). Find the smallest remaining number > 2. It is 3. So cross out all numbers > 3 which are divisible by 3 (every third number). Find the smallest remaining number > 3. It is 5. So cross out all numbers > 5 which are divisible by 5 (every fifth number). Continue until you have crossed out all numbers divisible by $\lfloor \sqrt{n} \rfloor$, where $\lfloor x \rfloor$ is the floor function. The numbers remaining are prime. This procedure is illustrated in the above diagram which sieves up to 50, and therefore crosses out composite numbers up to $\lfloor \sqrt{50} \rfloor = 7$.

```
import java.util.Scanner ;
public class PrimeSieve {
   public static void main(String[ ] args) {
      Scanner in = new Scanner(System.in) ;
      System.out.print("Enter N : ") ;
      int N = in.nextInt() ;
      // initially assume all integers are prime
      boolean[ ] isPrime = new boolean[N + 1] ;
      for (int i = 2 ; i <= N ; i++) {
         isPrime[i] = true ;
      }
```

```
                isprime[i*j] = false ;
            }
        }
        // count primes
        int primes = 0 ;
        for (int i = 2 ; i < N ; i++)
            if (isprime[i]) primes++ ;
        System.out.println("Done tabulating primes.") ;
        // store primes in list
        int[ ] list = new int[primes] ;
        int n = 0 ;
        for (int i = 0 ; i < N ; i++)
            if (isprime[i]) list[n++] = i ;
        // check if N can be expressed as sum of two primes
        int left = 0, right = primes -1 ;
        while (left <= right) {
            if  (list[left] + list[right] == N) break ;
            else if (list[left] + list[right] < N) left++ ;
            else right--  ;
        }
        if (list[left] + list[right] == N)
            System.out.println(N + " = " + list[left] + " + " + list[right]) ;
        else
            System.out.println(N + " not expressible as sum of two primes") ;
    }
}
```

The output produced would be :



Sometimes a problem that does not appear an array-based problem may also use arrays. For instance, consider the following program that performs binary addition but uses arrays to store binary numbers.

Program 8.8

Binary addition. Two binary digits are added as per following rules :

```
0 + 0 = 1
0 + 1 = 1
1 + 0 = 1
1 + 1 = 1 0      <-- carry bit
1 + 1 + 1 = 1 1  <-- carry bit
```

Write a Java program, that adds two binary numbers and returns their binary sum. You can use arrays to hold binary digits. The class specifications are :

```
class name : BinNumbers
Methods :
    long BinAdd(long A, long B) : adds two binary numbers received in A and B
```

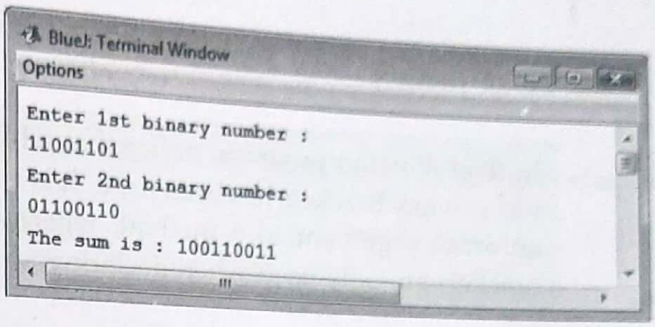
Write method main( ) that reads the two binary numbers and invokes BinAdd( ) to get their sum.

```
import java.util.Scanner ;
public class BinNumbers  {

  public long BinAdd(long A, long B)  {
     int[ ] R = new int[16] ;                    // To store the added bits' result
     for(int k = 0 ; k < 8 ; k++)
        R[k] = 0 ;
     int b1, b2, c, carry = 0, i = 15 ;          // i initialized to 15
     long a = A, b = B ;
     while (a > 0 && b > 0)  {
        b1 = (int)(a % 10) ;
        a /= 10 ;
        b2 = (int) (b % 10) ;
        b /= 10 ;
        c = b1 + b2 + carry ;                    // result bit stored in c
        if (c == 2)  {
            c = 0 ;
            carry = 1 ;
        }
        else if (c == 3)  {
            c = 1 ;
            carry = 1 ;
        }
        else if ( c == 0 || c == 1)
            carry = 0 ;
        else
            return -1 ;                          // invalid binary nos passed
        R[i-- ] = c ;                            // result bit stored in array R from right to left wards
  }

  if (a > 0)  {                                  // if first number has more bits
     b2 = 0 ;
     while(a > 0)  {
        b1 = (int)(a % 10) ;
        a /= 10 ;
        c = b1 + b2 + carry ;   // result bit
        if (c == 2)  {
            c = 0 ;
            carry = 1 ;
        }
```

```
            else if (c == 3) {
                c = 1 ;
                carry = 1 ;
            }
            else if ( c == 0 || c == 1)
                carry = 0 ;
            else
                return -1 ;                    // invalid binary nos passed
            R[i--] = c ;                       // result bit stored in array
        }
    }
    else if (b > 0)  {                         // if second number has more bits
        b1 = 0 ;
        while(b > 0) {
            b2 = (int)(b % 10) ;
            b /= 10 ;
            c = b1 + b2 + carry ;              // result bit
            if (c == 2) {
                c = 0 ;
                carry = 1 ;
            }
            else if (c == 3) {
                c = 1 ;
                carry = 1 ;
            }
            else if (c == 0 || c == 1)
                carry = 0 ;
            else
                return -1 ;                    // invalid binary nos passed
            R[i--] = c ;                       // result bit stored in array
        }
    }
    R[i--] = carry ;                           // overflow carry bit (carry of msb's) stored
    long sum = 0 ; int d ;
        for( int j = 0 ; j < 16 ; j++)
        {
            sum = sum * 10 + R[ j ] ;          // Result converted to numbers from result
        }                                      // bits stored in array R
        return sum ;
    }

public static void main(String[ ] args ) {
    Scanner in = new Scanner(System.in) ;
    long binNum1, binNum2, binNum3 ;
    System.out.println("Enter 1st binary number : ") ;
```

```
        binNum1= in.nextLong() ;
        System.out.println("Enter 2nd binary number : ") ;
        binNum2= in.nextLong() ;
        BinNumbers bns = new BinNumbers() ;
        binNum3 = bns.BinAdd(binNum1, binNum2) ;
        System.out.println("The sum is : "+binNum3) ;
    }
}
```

The output produced would be :



8.3.4 Operations on 1-D Arrays

You can use and manipulate 1-D arrays through many operations such as : searching, sorting etc. In this section, we are going to quickly brush up these operations.

8.3.4A Searching

For searching in 1-D arrays, you can apply one of the two popular searching techniques *viz.* linear search and binary search.

Linear Search

In linear search, each element of the array is compared with the given *Item* to be searched for, one by one. This method, which traverses the array sequentially to locate the given *Item*, is called *linear search* or *sequential search*.

The above search technique will prove the worst, if the element to be searched is one of the last elements of the array as so many comparisons would take place and the entire process would be time-consuming. *To save on time and number of comparisons, binary search is very useful.*

Program 8.9

To perform linear search in a given array of integers.

```
public class LinearSearch {
    public static void main(String[ ] args)    {
        int[ ] numbers = { 12, 13, 2, 33, 23, 31, 22, 6, 87, 16 };
        int key = 31;              // item to be searched for
        int i = 0 ;
        boolean found = false;  // set the boolean value to false until the key is found
                for ( i = 0; i < numbers.length; i++) {
            if (numbers[ i ]  == key)    {
                found = true;
                break;
            }
        }
        if (found) {//When found is true, the index of the location of key will be printed.
```

```
                System.out.println("Found " + key + " at index " + i + ".");
            }
            else {
            System.out.println(key + "is not in this array.") ;
            }
        }
    }
```

```
Found 31 at index 5
```

In the following program, notice that while passing array to the method, only array name without any brackets has been used. It is a syntax error to use empty brackets when passing an array argument to a method, where only the array's name should be used. Empty brackets are only used when declaring an array variable.

Binary Search

This popular search technique searches the given *ITEM* in minimum possible comparisons. The *binary search* requires the array, to be scanned, must be sorted in any order (for instance, say ascending order). In binary search, the *ITEM* is searched for in smaller *segment* (nearly half the previous segment) after every stage. For the first stage, the segment contains the entire array.

To search for *ITEM* in a sorted array (in *ascending order*), the *ITEM* is compared with *middle element* of the segment (*i.e.*, in the entire array for the first time). If the *ITEM* is more than the middle element, latter part of the segment becomes new segment to be scanned ; if the *ITEM* is less than the *middle element*, former part of the segment becomes new segment to be scanned.

The same process is repeated for the new segment(s) until either the *ITEM* is found (search successful) or the segment is reduced to the single element and still the ITEM is not found (search unsuccessful).

Program 8.10

To perform binary search in a given array of integers.

```
public class BinarySearch {
   public static void main(String[ ] args)   {
      int[ ] array = { 11,15,  25, 27, 45, 47, 63, 70, 77, 90} ;
      int key = 77 ;          // item to be searched for
      binary_search (array, 0, 9, key) ;
   }

   //Binary Search Method

   // Method accepts a pre-sorted array, the index of the starting element for the search,
   // the range (number) of elements to be searched,
   // and the number for which we are searching.
   public static void binary_search(int[ ] array, int lowerbound, int upperbound, int key)
   {
      int position ;
      int compare_count = 1 ;
```

```
        position = ( lowerbound + upperbound) / 2;     // calculate initial search position.
        while((array[position] != key) && (lowerbound < upperbound))
        {
            compare_count++;
            // if the value in the search position is greater than the number for which we
            //are searching, then change upperbound to search position minus one.
            if (array[position] > key)   {
                upperbound = position - 1;
            }
            else  {
                lowerbound = position + 1 ; // Else, change lowerbound to search position plus one.
            }
            position = (lowerbound + upperbound) / 2;
        }
        if (lowerbound < upperbound)  {
            System.out.println("A binary search found the number in "
                                + compare_count + "comparisons.");
            System.out.println("The number was found in array subscript" + position);
        }
        else
            System.out.println("Number not found by binary search after "
                                + compare_count + "comparisons.");
    }
}
```

```
A binary search found the number in 3 comparisons.
The number was found in array subscript 8
```

Binary search can work for only sorted arrays whereas linear search can work for both sorted as well as unsorted arrays.

8.3.4B Sorting

Sorting of an array means arranging the array elements in a specified order *i.e.*, either ascending or descending order.

Some very common and popular sorting techniques are : selection sort and bubble sort. Let us quickly revise these techniques.

Selection Sort

The basic idea of a selection sort is to repeatedly select the smallest key in the remaining unsorted array and bring it to its right position. In other words, in an unsorted array, the first smallest will be brought (generally via swapping) to first position, second smallest to second position, third smallest to third position, and so on.

Program 8.11

To perform selection sort in a given array of integers.

```
public class SelectionSort  {
    public static void main(String[] args) throws Exception {
        int[ ] array = { 41, 15,  25, 77, 45, 37, 23, 70, 27, 9} ;
```