Chapter 10: Arrays and Collections

Objectives

After you have read and studied this chapter, you should be able to

- Manipulate a collection of data values, using an array.
- Declare and use an array of primitive data types in writing a program.
- Declare and use an array of objects in writing a program.
- Define a method that accepts an array as its parameter and a method that returns an array.
- Describe how a two-dimensional array is implemented as an array of arrays.
- Manipulate a collection of objects, using lists and maps.

10.1 Array Basics

Suppose we want to compute the annual average rainfall from 12 monthly averages. We can use three variables and compute the annual average as follows:

```
import java.util.Scanner;
 public class Eg1
 {
    public static void main(String[] args)
    {       double sum, rainfall, annualAverage;
            sum = 0.0;
            Scanner scan = new Scanner(System.in);
            for (int i = 0; i < 12; i++)
            {
                   System.out.print("Rainfall for month " + (i+1) + ": ");
                   rainfall = scan.nextDouble();
                   sum += rainfall;
            }
            annualAverage = sum / 12.0;
  }
}
```

Now suppose we want to compute the difference between the annual and monthly averages for every month and display a table with three columns, similar to the one shown in Figure 10.1.

To compute the difference between the annual and monthly averages, we need to remember the 12 monthly rainfall averages. Without remembering the 12 monthly averages, we won't be able to derive the monthly variations after the annual average is computed. Instead of using 12 variables januaryRainfall, februaryRainfall, and so forth to solve this problem, we use an array.

Annual Average Rainfall: 15.03 mm

Month	Average	Variation
1	13.3	1.73
2	14.9	0.13
3	14.7	0.33
4	23.0	7.97
5	25.8	10.77
6	27.7	12.67
7	12.3	2.73
8	10.0	5.03
9	9.8	5.23
10	8.7	6.33
11	8.0	7.03
12	12.2	2.83

Figure 10.1 Monthly rainfall figures and their variation from the annual average.

An array is a collection of data values of the same type. For example, we may declare an array consisting of double, but not an array consisting of both int and double. The following declares an array of double:

```
double[] rainfall;
```

The square brackets indicate the array declaration. The brackets may be attached to a variable instead of the data type. For example, the declaration

```
double rainfall[];
```

is equivalent to the previous declaration. In Java, an array is a reference data type. Unlike the primitive data type, the amount of memory allocated to store an array varies, depending on the number and type of values in the array. We use the new operator to allocate the memory to store the values in an array.

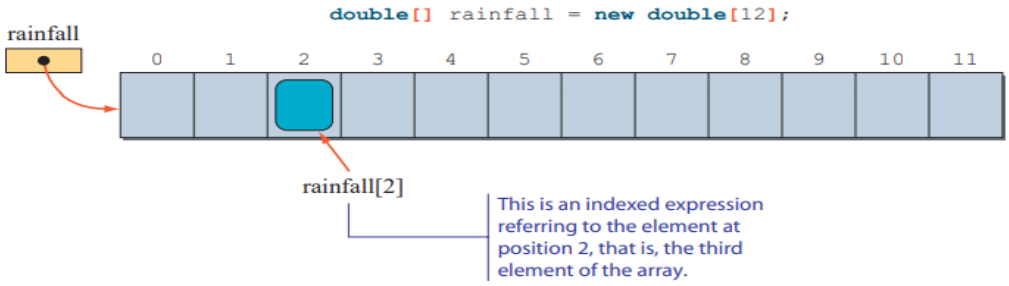


Figure 10.2 An array of 12 double values.

The following statement allocates the memory to store 12 double values and associates the identifier rainfall to it.

```
rainfall = new double[12]; //create an array of size 12
```

Figure 10.1 shows this array.

We can also declare and allocate memory for an array in one statement, as in

```
double[] rainfall = new double[12];
```

The number 12 designates the size of the array—the number of values the array contains. We use a single identifier to refer to the whole collection and use an ***indexed expression*** to refer to the individual values of the collection. An individual value in an array is called an ***array element***. Zero-based indexing is used to indicate the positions of an element in the array. They are numbered 0, 1, 2, . . . , and size – 1, where size is the size of an array. For example, to refer to the third element of the rainfall array, we use the indexed expression

```
rainfall[2]
```

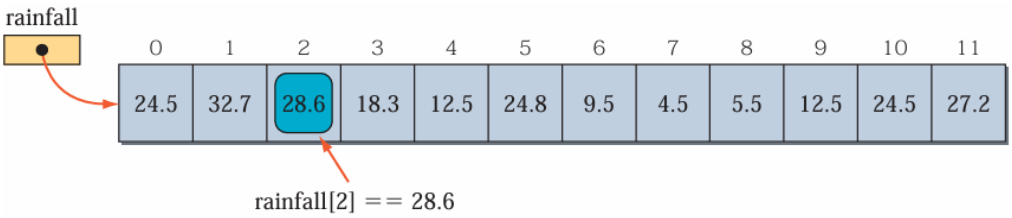


Figure 10.3 An array of 12 double values after all 12 are assigned values

Using the rainfall array, we can input 12 monthly averages and compute the annual average as

```
import java.util.Scanner;
public class Eg2
{       public static void main(String[] args)
        {       Scanner scan = new Scanner(System.in);
                double[] rainfall = new double[12];
                double annualAverage,
                sum = 0.0;
                for (int i = 0; i < 12; i++) {
                System.out.print("Rainfall for month " + (i+1) + ": ");
                rainfall[i] = scan.nextDouble();
                sum += rainfall[i];
                }
                annualAverage = sum / 12.0;
        }
}
```

Can also be declared as

```
double rainfall[] = new double[12];
```

After the 12 monthly averages are stored in the array, we can print out the table. Here's the complete program:

```
import java.util.Scanner;
public class Ch10Rainfall
{       public static void main(String[] args)
        {       Scanner scan = new Scanner (System.in);
                double[] rainfall = new double[12];
                double annualAverage, sum, difference;
                sum = 0.0;
                for (int i = 0; i < 12; i++)
                {
                        System.out.print("Rainfall for month " + (i+1) + ": ");
                        rainfall[i] = scan.nextDouble();
                        sum += rainfall[i];
                }
                annualAverage = sum / 12.0;
                System.out.format("Annual Average Rainfall:%5.2f\n\n",
                                annualAverage);
```

```
            for (int i = 0; i < 12; i++)
            {       System.out.format("%3d", i+1); //month #
                    //average rainfall for the month
                    System.out.format("%15.2f", rainfall[i]);
                    //difference between the monthly and annual averages
                    difference = Math.abs( rainfall[i] - annualAverage );
                    System.out.format("%15.2f\n", difference);
            }
        }
}
```

Here is a sample output:

```
Rainfall for month 1: 12
Rainfall for month 2: 11.3
Rainfall for month 3: 13.5
Rainfall for month 4: 16.1
Rainfall for month 5: 13
Rainfall for month 6: 17
Rainfall for month 7: 12
Rainfall for month 8: 21.3
Rainfall for month 9: 10.2
Rainfall for month 10: 11
Rainfall for month 11: 12.4
Rainfall for month 12: 12.6
Annual Average Rainfall:13.53

  1          12.00           1.53
  2          11.30           2.23
  3          13.50           0.03
  4          16.10           2.57
  5          13.00           0.53
  6          17.00           3.47
  7          12.00           1.53
  8          21.30           7.77
  9          10.20           3.33
 10          11.00           2.53
 11          12.40           1.13
 12          12.60           0.93
```

Notice the prompts for getting the values in the previous example are Rainfall for month 1, Rainfall for month 2, and so forth. A better prompt will spell out the month name, for example, Rainfall for January, Rainfall for February, and so forth. We can easily achieve a better prompt by using an array of strings. Here's how:

```
double[] rainfall = new double[12]; //an array of double
String[] monthName = new String[12]; //an array of String
double annualAverage, sum = 0.0;
monthName[0] = "January";
monthName[1] = "February";
```

```
monthName[2] = "March";
monthName[3] = "April";
monthName[4] = "May";
monthName[5] = "June";
monthName[6] = "July";
monthName[7] = "August";
monthName[8] = "September";
monthName[9] = "October";
monthName[10] = "November";
monthName[11] = "December";
```

```
String[] monthName = {"January", "February", "March",
                      "April", "May", "June", "July",
                      "August", "September", "October",
                      "November", "December" };
```

```
for (int i = 0; i < rainfall.length; i++) {
       System.out.print("Rainfall for month" +  monthName[i] + ": ");
       rainfall[i] = scan.nextDouble();
       sum += rainfall[i];
}
 annualAverage = sum / 12.0;
```

Let's try some more examples. We assume the rainfall array is declared, and all 12 values are read in. The following code computes the average rainfall for the odd months (January, March, . . .) and the even months (February, April, . . .).

```
double oddMonthSum, oddMonthAverage, evenMonthSum, evenMonthAverage;
oddMonthSum = 0.0; evenMonthSum = 0.0;
//compute the average for the odd months
 for (int i = 0; i < rainfall.length; i += 2){
       oddMonthSum += rainfall[i];
  }
oddMonthAverage = oddMonthSum / 6.0;
//compute the average for the even months
for (int i = 1; i < rainfall.length; i += 2){
       evenMonthSum += rainfall[i];
 }
  evenMonthAverage = evenMonthSum / 6.0;
  oddMonthAverage  = oddMonthSum / 6.0;
  evenMonthAverage = evenMonthSum / 6.0;
```

We can compute the same result by using one for loop.

```
for(int i=0; i<rainfall.length; i+=2){
        oddMonthSum += rainfall[i];
        evenMonthSum += rainfall[i+1];
}
```

To compute the average for each quarter (quarter 1 has January, February, and March; quarter 2 has April, May, and June; and so forth), we can write

```
for (int i = 0; i < 3; i++ ) {
       quarter1Sum += rainfall[i];
       quarter2Sum += rainfall[i+3];
       quarter3Sum += rainfall[i+6];
       quarter4Sum += rainfall[i+9];
```

```
}
quarter1Average = quarter1Sum / 3.0;
quarter2Average = quarter2Sum / 3.0;
quarter3Average = quarter3Sum / 3.0;
quarter4Average = quarter4Sum / 3.0;
```

We can use another array to store the quarter averages instead of using four variables:

```
double[] quarterAverage = newdouble[4];
 for(int i = 0; i < 4; i++) {
       sum = 0;
       for(int j = 0; j < 3; j++) {            //compute the sum of
              sum += rainfall[3*i + j];        //one quarter
}
quarterAverage[i]= sum / 3.0;     //average for quarter i+1
}
```

Notice how the inner for loop is used to compute the sum of one quarter. The following table illustrates how the values for the variables i and j and the expression 3*i + j change.

i	j	3*i + j
0	0	0
	1	1
	2	2
1	0	3
	1	4
	2	5
2	0	6
	1	7
	2	8
3	0	9
	1	10
	2	11

Here's the complete program:

```
import java.util.*;
public class Ch10RainfallStat {
       public static void main(String[] args) {
              Scanner scan = new Scanner(System.in);
String[]    monthName={"January","February","March","April","May","June",    "July",
                    "August","September", "October", "November", "December" };

double [ ] rainfall = new double[12];
double[] quarterAverage = new double[4];
double annualAverage, sum, difference;
double oddMonthSum, oddMonthAverage, evenMonthSum, evenMonthAverage;

 sum = 0.0;
 for (int i = 0; i < rainfall.length; i++){
```

```
   System.out.print("Rainfall for month " + monthName[i] + ": ");
   rainfall[i] = scan.nextDouble();
   sum += rainfall[i];
     }
 annualAverage = sum / 12.0;
 System.out.format( "Annual Average Rainfall:%6.2f\n\n", annualAverage );

 oddMonthSum = 0.0;
 evenMonthSum = 0.0;
 ////////////// Odd and Even Month Averages ////////////////////////////
for (int i = 0; i < rainfall.length; i += 2) {
      oddMonthSum += rainfall[i];       //compute the average for the odd months
     }
 oddMonthAverage = oddMonthSum / 6.0;
for (int i = 1; i < rainfall.length; i += 2) {
      evenMonthSum += rainfall[i];     //compute the average for the even months
     }
 evenMonthAverage = evenMonthSum / 6.0;
System.out.format("Odd Month Rainfall Average: %6.2f\n", oddMonthAverage );
System.out.format("Even Month Rainfall Average:%6.2f\n\n", evenMonthAverage );

//////////////////// Quarter Averages ////////////////////////////////////
for (int i = 0; i < 4; i++) {
       sum = 0;
       for (int j = 0; j < 3; j++){     //compute the sum of
            sum += rainfall[3*i + j]; //one quarter
          } //end for j loop
           quarterAverage[i] = sum / 3.0; //average for quarter i+1
            System.out.format("Rainfall Average Qtr.%3d:%6.2f\n", i+1,
                              quarterAverage[i]);
         } // end for i loop
     }//end main
 }//end class
```

10.2 Array of Objects

Array elements are not limited to primitive data types. To illustrate the processing of an array of objects, we will use the Person class in the following examples. Here's the portion of the Person class definition we will use in this section:

Public Methods of the `Person` Class

`public int getAge ( )`
Returns the age of a person. Default age of a person is set to 0.

`public char getGender( )`
Returns the gender of a person. The character `F` stands for female and `M` for male. Default gender of a person is set to the character `U` for unknown.

`public String getName ( )`
Returns the name of a person. Default name of a person is set to `Not Given`.

`public void setAge ( int age )`
Sets the age of a person.

`public void setGender( char gender )`
Sets the gender of a person to the argument `gender`. The character `F` stands for female and `M` for male. The character `U` designates unknown gender.

`public void setName ( String name )`
Sets the name of a person to the argument `name`.

Now let's study how we can create and manipulate an array of Person objects. An array of objects is declared and created just as an array of primitive data types is. The following are a declaration and a creation of an array of Person objects.

```
Person[] person; //declare the person array
person = new Person[20]; //and then create it
```

Execution of the above code will result in a state shown in Figure 10.2. Notice that the elements, that is, Person objects, are not yet created; only the array is created. Array elements are initially null. Since each individual element is an object, it also must be created. To create a Person object and set it as the array's first element, we write

```
person[0] = new Person( );
```

Figure 10.3 shows the state after the first Person object is added to the array. Notice that no data values are assigned to the object yet. The object has default values at this point. To assign data values to this object, we can execute

```
person[0].setName ( "Ms. Latte" );
person[0].setAge ( 20 );
person[0].setGender( 'F' );
```

The indexed expression

```
person[0]
```

```
        person[0]
```

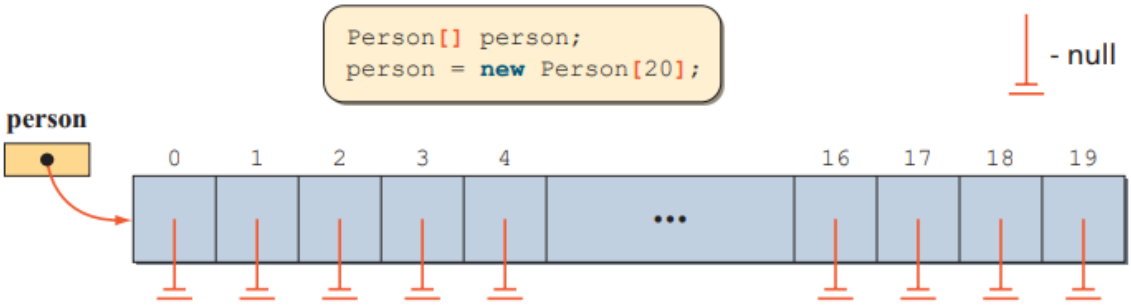


Figure 10.2: An array of Person objects after the array is created.

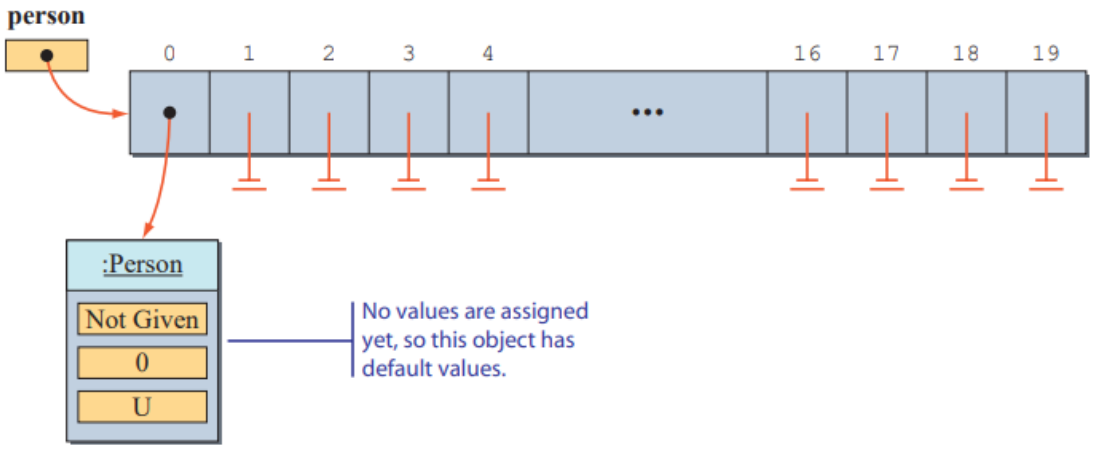


Figure 10.3: The person array with one Person object added to it.

Here's the complete program that summarizes the topics covered so far in this section:

```
//Person.java
	public class Person
	{	String name, inpStr;
		int age;
		char gender;
		public String getName() {
			return name;
		}
		public void setName(String name) {
			this.name = name;
		}
		public int getAge() {
			return age;
		}
		public void setAge(int age) {
			this.age = age;
		}
		public char getGender() {
			return gender;
		}
		public void setGender(char gender) {
			this.gender = gender;
		}
	}
// Ch10ProcessPersonArray.java
import java.util.Scanner;
public class Ch10ProcessPersonArray
{	public static void main(String[] args)
```

```
{
        Person[] person; //declare the person array
        person = new Person[5]; //and then create it
        //----------- Create person Array -------------------//
        String name, inpStr;
        int age;
        char gender;
        Scanner scan = new Scanner(System.in);
        scan.useDelimiter(System.getProperty("line.separator"));
        for (int i = 0; i < person.length; i++)
        {       //read in data values
                System.out.print("Enter name: ");
                name = scan.next();
                System.out.print("Enter age: ");
                age = scan.nextInt();
                System.out.print("Enter gender: ");
                inpStr = scan.next();
                gender = inpStr.charAt(0);
                //create a new Person and assign values
                person[i] = new Person( );
                person[i].setName ( name );
                person[i].setAge ( age );
                person[i].setGender( gender );
        }
//---------------- Compute Average Age -----------------//
        float sum = 0, averageAge;
        for (int i = 0; i < person.length; i++)
        {
                sum += person[i].getAge();
        }
        averageAge = sum / (float) person.length;
        System.out.println("Average age: " + averageAge);
        System.out.println("\n");
//------ Find the youngest and oldest persons ----------//
//------ Approach No. 3: Using person reference --------//
        Person youngest, //points to the youngest person
        oldest; //points to the oldest person
        youngest = oldest = person[0];
        for (int i = 1; i < person.length; i++)
        {       if (person[i].getAge() < youngest.getAge())
```

```
                    {       youngest = person[i];  //found a younger person
                    }
                    else if (person[i].getAge() > oldest.getAge())
                    {       oldest = person[i];   //found an older person
                    }
            }
            System.out.println("Oldest : " + oldest.getName() + " is " +
                                oldest.getAge() + " years old.");
            System.out.println("Youngest: " + youngest.getName() + " is " +
                                youngest.getAge() + " years old.");
        //------------- Search for a particular person ----------------//
            System.out.print("Name to search: ");
            String searchName = scanner.next();
            int i = 0;
        //still more persons to search
        while (i < person.length && !person[i].getName().equals(searchName))
         {   i++;
         } //end while

        if (i == person.length)
        {       //not found - unsuccessful search
            System.out.println( searchName + " was not in the array" );
        }
        else
         {//found - successful search
            System.out.println("Found " + searchName + " at position " + i);
         }
    }
}
```

Here is a sample output:

```
Enter name: Su Myat
Enter age: 20
Enter gender: F
Enter name: Min Thu
Enter age: 21
Enter gender: M
Enter name: May Mon
Enter age: 19
Enter gender: F
Enter name: Aung Phyo
Enter age: 18
Enter gender: M
Enter name: Aye Su Mon
Enter age: 16
Enter gender: F
Average age: 18.8


Oldest : Min Thu is 21 years old.
Youngest: Aye Su Mon is 16 years old.
Name to search: May Mon
Found May Mon at position 2
```

Figure 10.4 shows how the Person variables oldest and youngest point to objects in the person array.

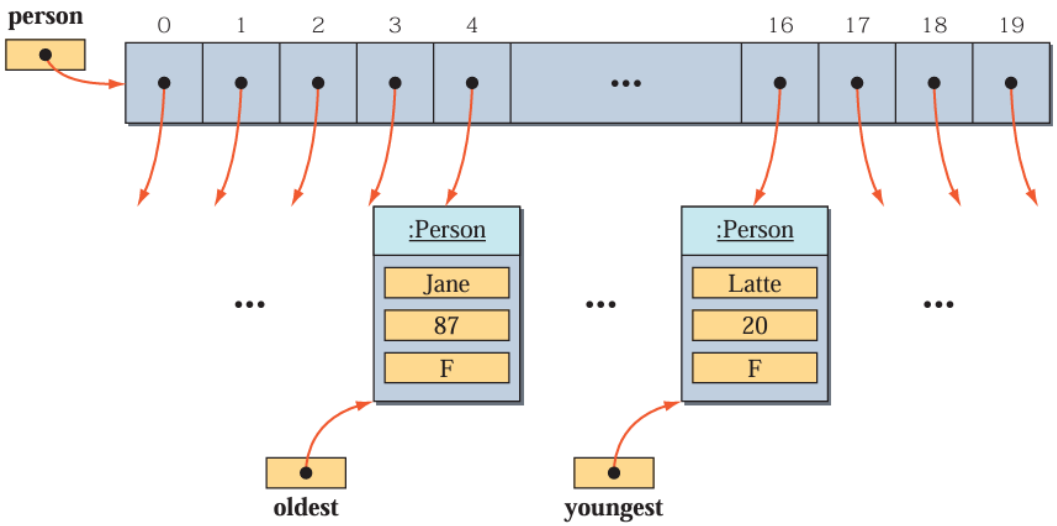


Figure 10.4 An array of Person objects with two Person variables.

Now let's consider the deletion operation. The deletion operation requires some kind of a search routine to locate the Person object to be removed.

With approach 1, any index position can be set to null, so there can be "holes," that is, null references, anywhere in the array. Figure 10.4 illustrates how the object at position 1 is deleted by using approach 1. Instead of intermixing real and null references, the second approach will pack the elements so that the real references occur at the beginning and the null references at the end.

The second solution is a better one if the Person objects are not arranged in any order. Since we are not arranging them in any order, we will use the second solution. Figure 10.5 illustrates how the object at position 1 is replaced by the last element.

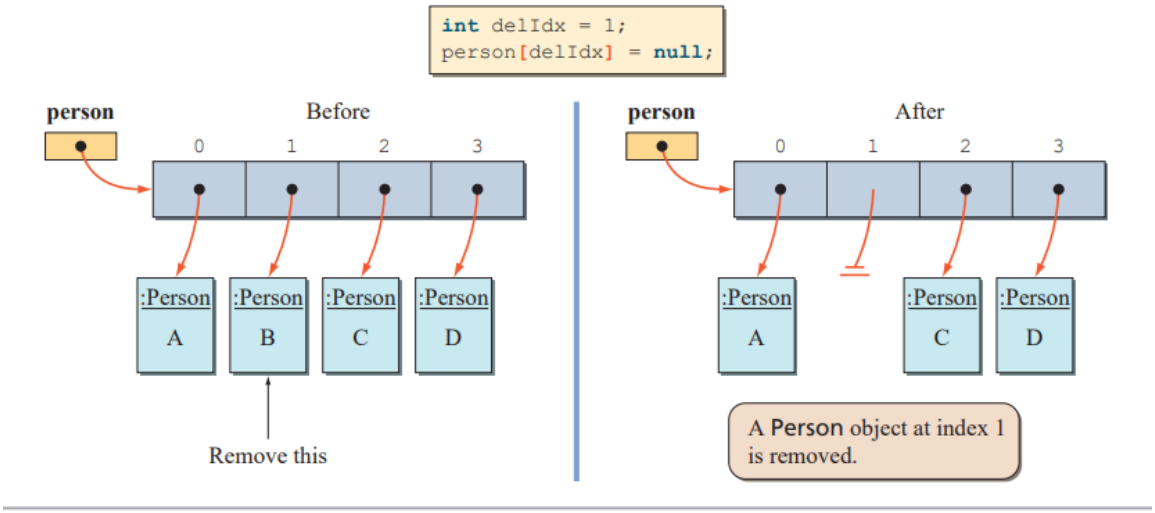


Figure 10.5: Approach 1 deletion: setting a reference to null. The array length is 4.

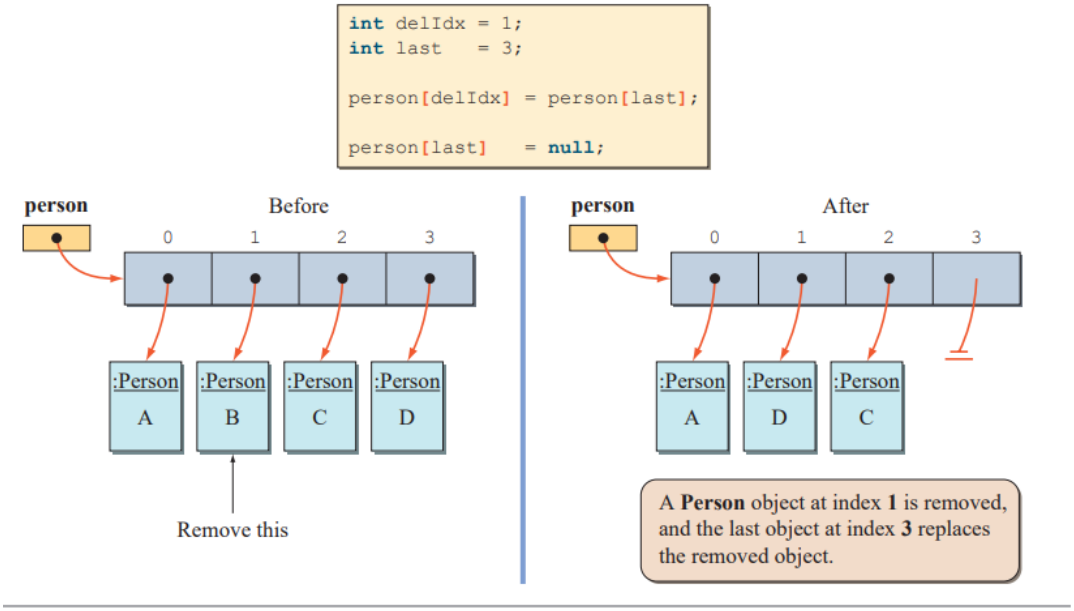
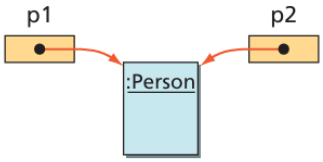


Figure 10.6: Approach 2 deletion: replace the removed element with the last element in the array. The array length is 4.

The following code will result in two references pointing to a single Person object:

```
Person p1, p2;
p1 = new Person();
p2 = p1;
```



10.3 The For-Each Loop

This is a new form of the for loop in Java 5.0. There is no official name to this for loop, but the term for-each is used most often. The term enhanced for loop is also used by many to refer to this for loop. Let's assume number is an int array of 100 integers. Using the standard for loop, we compute the sum of all elements in the number array as follows:

```
int sum = 0;
for (int i = 0; i < number.length; i++)
{
        sum = sum + number[i];
}
```

By using a for-each loop, we can compute the sum as follows:

```
int sum = 0;
for (int value: number)
{
        sum = sum + value;
}
```

The general syntax for the for-each loop is

for (<type> < variable > : < array >)

< loop body >

Assuming that 100 Person objects are created and assigned to person [0] to person [99], we can list the name of every person in the array by using the following for-each loop:

```
for (Person p: person)
{
        0System.out.println(p.getName());
}
```

Contrast this to the standard for loop:

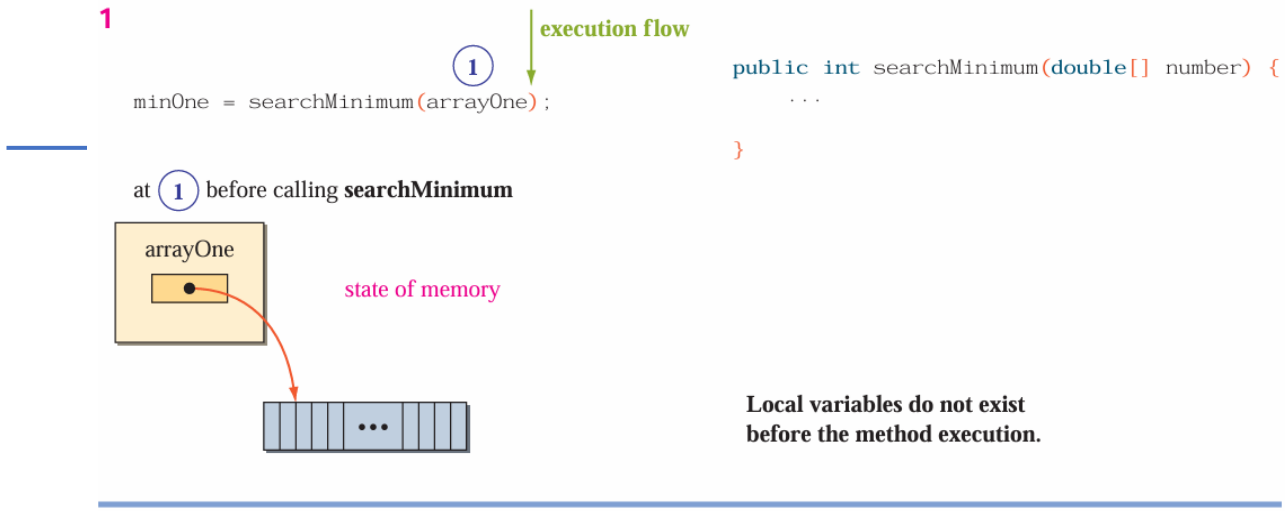
The for-each loop is, in general, cleaner and easier to read. The for-each loop allows access to only a single array.

```
for (int i = 0; i < person.length; i++)
{
        System.out.println(person[i]. getName());
}
```

10.4 Passing Arrays to Methods

Let's define a method that returns the index of the smallest element in an array of real numbers. The array to search for the smallest element is passed to the method. When an array is passed to a method, only its reference is passed. A copy of the array is not created in the method. Here's the method:

```
public int searchMinimum(double[] number)
{
        int indexOfMinimum = 0;
        for (int i = 1; i < number.length; i++)
        {
                if (number[i] < number[indexOfMinimum])
                {
                        indexOfMinimum = i; // found a smaller element
                }
        }
        return indexOfMinimum;
}
```



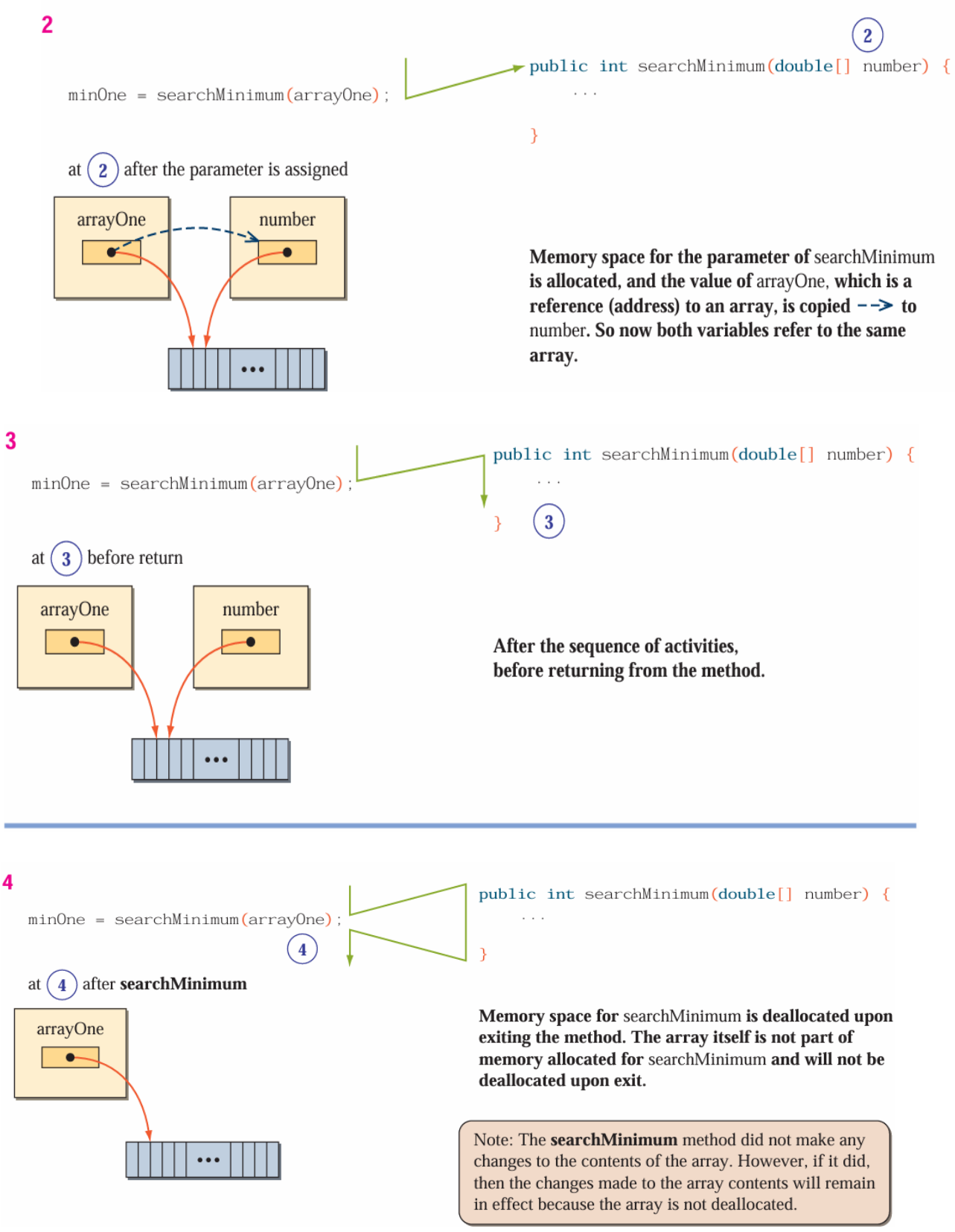


Figure 10.7 Passing an array to a method means we are passing a reference to an array. We are not passing the whole array.

10.5 Two-Dimensional Arrays

Data represented in tabular form (organized in rows and columns) is a very effective means for communicating many different types of information. In Java, this tabular representation of data is implemented using a ***two-dimensional array***. A two-dimensional array (or 2D array in Java) is a linear data structure that is used to store data in tabular format. Let's understand the creation of Java's two-dimensional array with an example:

```
int[][] marks;              //Declaring 2D array
marks  =  new int[5][4];    //Creating a 2D array
marks[0][0] = 89            //initialize data in marks array
```

	Column 1	Column 2	Column 3	Column 4
Row 1	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]
Row 2	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]
Row 3	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]
Row 4	marks[3][0]	marks[3][1]	marks[3][2]	marks[3][3]
Row 5	marks[4][0]	marks[4][1]	marks[4][2]	marks[4][3]

```
int marks =  { { 89, 56, 64, 70 },
               { 75, 65, 60, 57 },
               { 80, 52, 53, 67 },
               { 55, 45, 64, 71 },
               { 40, 52, 60, 37 } };
```

	[0]	[1]	[2]	[3]
[0]	89	56	64	70
[1]	75	65	60	57
[2]	80	52	53	67
[3]	55	45	64	71
[4]	40	52	60	37

Multiplication Table

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	26	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Here is a sample program for 2-dimensional array:

```
public class TwoDArray
{       public static void main(String[] args)
        {
                int[][] arr = { { 1, 2 }, { 3, 4 } };
                for (int i = 0; i < 2; i++)
                        for (int j = 0; j < 2; j++)
                                System.out.println("arr[" + i + "][" + j + "] = "
                                                                + arr[i][j]);
        }
}
```

Output is

```
arr[0][0] = 1
arr[0][1] = 2
arr[1][0] = 3
arr[1][1] = 4
```

10.6 Lists and Maps

Once an array is created, its capacity cannot be changed. For example, if we create an array of 20 elements, then we are limited to store at most 20 elements in using this array. If we need to add elements, then we have to create a new array.

The first is the List interface. Like a class, an ***interface*** is a reference data type; but unlike a class, an interface includes only constants and abstract methods. An ***abstract*** method has only the method header (or, more formally, the method prototype); that is, it has no method body. It contains the index-based methods to insert, update, delete, and search the elements. It can have duplicate elements also.

Let's study how we can use the methods of the List interface. First we need to declare and create an instance of a class that implements the List interface.If we declare a list object as

```
List myList = new ArrayList();
```

there are no restrictions on the type of objects we can add to the list. For example, we can add String objects, Person objects, Vehicle objects, and so forth to this myList.

We call such list a ***heterogeneous list***. In most applications, there is no need to maintain such heterogeneous lists. What we need is a ***homogeneous list***, where the elements are restricted to a specific type such as a list of Person objects, a list of String objects, a list of Book objects, and so forth. Specifying the element type improves the program reliability because an error such as trying to add a wrong type of object to a list can be caught during the compile time.

It is strongly recommended to use homogeneous lists. To specify a homogeneous list, we must include the type of elements in the declaration and creation statements. Here's an example that declares and creates a list of Person objects:

```
List<Person> friends = new ArrayList<Person>( );
```

In the following example, we create a list named friends and add four Person objects to the list:

```
List<Person> friends = new ArrayList<Person>();
Person person;
person = new Person("Jane", 10, 'F');
friends.add(person);
person = new Person("Jack", 16, 'M');
```

```
friends.add(person);
person = new Person("Jill", 8, 'F');
friends.add(person);
person = new Person("John", 12, 'M');
friends.add(person);
```

We use the **get** method to access an object at index position i. For example, to access the Person object at position 3 (note: the first element is at position 0) in the friends list, we write

```
Person p = friends.get(3);
```

Here's how we can print out the names of all those in the friends list by using the for-each loop:

```
for (Person p: friends)
{      System.out.println(p.getName());
}
```

This **for-each** loop is actually a shortcut for using an iterator pattern. When we call the iterator method of a list, it returns an Iterator object (an instance of a class that implements the Iterator interface) that supports the two methods **hasNext** and **next**. Here's the code to print out the names of all those in the friends list by using an iterator:

```
Person p;
Iterator<Person> itr = friends.iterator();
while (itr.hasNext())
{      p = itr.next();
       System.out.println(p.getName());
}
```

To remove an element from a list, we use the **remove** method. here's how we remove the Person object at index position 2 in the friends list:

```
friends.remove(2);
```

Lists and Primitive Data Types

With an array, we can store either primitive data values (int, double, etc.) or objects. With a list, we can store only objects. If we need to store primitive data values in a list, then we must use wrapper classes such as Integer, Float, and Double. To specify a homogeneous list of integers, we have to do something like this:

```
List <Integer> intList = new ArrayList <Integer>( );
intList.add(new Integer(15));
intList.add(new Integer(30));
...
```

The following code computes the sum of integer values stored in intList:

```
int sum = 0;
for (int value : intList)
{     sum = sum + value;
}
```

When we write, for example,

```
intList.add(30);
```

the compiler translates it to

```
intList.add(new Integer(30));
```

This is called ***auto boxing***.

And when we write

```
int num = intList.get(1);
```

the compiler translates it to

```
int num = intList.get(1).intValue( );
```

This is called ***auto unboxing.***

Map

Let's move on to another useful interface called **Map.** A map consists of entries, with each entry divided into two parts: **key** and **value**. No duplicate keys are allowed in the map. Both key and value can be an instance of any class. The main advantage of a map is its performance in locating an entry, given the key.

When declaring and creating a map, we must specify the type for the key and the value. For example, to declare and create a map with String as both its **key and value**, we write

```
Map<String,String> table;
table = new TreeMap<String,String>( );
```

We use its **put** method to add the **key-value pairs** to the map as

```
table.put("CS0101", "Great course. Take it");
```

where the first argument is the **key** and the second argument is the **value**.

To remove an entry, we use the **remove** method with the key of an entry to remove from the map, for example,

```
table.remove("CS2300");
```

Instead of removing individual elements, we can remove all of them at once by calling the clear method. The statement

```
table.clear( );
```

removes everything from the map, making it an empty map.

To retrieve the value associated to a key, we call the map's get method.

```
String courseEval = table.get("CS102");
```

We can ask the map if it contains a given key. To check, for example, whether the map contains an evaluation for course number CS0455, we write

```
boolean result = table.containsKey("CS0455");
```

If there's no matching entry, then the value null is returned.

To traverse a map, we must first call its **entrySet** method to get a set of elements.

Two useful methods defined in the Map.Entry interface are the **getKey** and **getValue**, whose purpose is to retrieve the key and the value of an entry, respectively. To put it all together, here's an example that outputs the course numbers and their evaluations stored in the table map:

```
for (Map.Entry<String, String> entry: table.entrySet())
{       System.out.println(entry.getKey() + ":\n" +
                           entry.getValue() + "\n");
}
```

****

Exercises

1. Rewrite the following for loop by using a for-each loop.

```
for (int i = 0; i < number.length; i++) {
      System.out.println(number[i]);
}
```

2. Rewrite the following for loop by using the standard for loop.

```
for (Person p : person){
     System.out.println(p.getName());
}
```

3. Write a int countChar(char [ ] [ ] charArr, char ch) method that takes in 2 arguments; a 2 dimensional array of char, charArr and a character named ch. The method will return the number of occurrences of ch in the 2 dimensional array charArr.

```
public class Ex3
{  int countChar(char[][] charArr, char ch)
    {   int count = 0;
        for (int i = 0; i < charArr.length; i++)
              for ( int j = 0; j < charArr[i].length; j++)
                    if ( ch == charArr[i][j])
                          count ++;
        return count;
    }
    public static void main(String[] args) {
        char chArr[][] = {{'a','b'},{'a','e'},{'e','d'}};
        Ex3 obj = new Ex3 ();
        int count = obj.countChar(chArr , 'a');
        System.out.println("Count = "+ count);
     }
}
```

Output

Count = 2

4. Write a countMultiple (int [ ] arr, int x) method that takes in 2 arguments; an array of integer, arr and an integer named x. The method returns the number of integers in the array which is a multiple of x.

```
public class Ex4
{   int countMultiple(int[] arr, int x)
    {
        int count = 0;
```

```
        for ( int i = 0; i < arr.length; i++)
        {       if(arr[i] % x == 0)
                   count ++;
        }
        return count;
    }
    public static void main(String[] args)
    {   int Arr[ ] = {4, 5, 10, 20, 3};
        Ex4 obj = new Ex4( );
        int co = obj.countMultiple( Arr, 5);
        System.out.println("Count Multiple = " + co);
    }
}
```

Output

Count Multiple = 3

5. Write a getHighest(int[] arr) method which returns the highest integer value in the array.

```
public class Ex5
{    int getHighest(int[] arr, int x)
    {    int max = 0;
         for ( int i = 0; i < arr.length; i++)
         {       if(arr[i] > max)
                    max = arr[i];
         }
         return max;
    }
    public static void main(String[] args)
    {    int Arr[ ] = {4, 5, 10, 20, 3};
         Ex5 obj = new Ex5( );
         int hi = obj.getHighest( Arr, 5);
         System.out.println("Highest Number = " + hi);
    }
}
```

Output

Highest Number = 20

6. Write a Java program to calculate the average value of array elements.

```
public class Ex6
{
   public static void main(String[] args)
   {    int[] numbers = new int[]{20, 30, 25, 35, -16, 60, -100};
        //calculate sum of all array elements
        int sum = 0;
        for(int i=0; i < numbers.length ; i++)
```

```
        {
            sum = sum + numbers[i];
        }
            //calculate average value
            double average = sum / numbers.length;
           System.out.println("Average value of the array elements is : " +
                              average);
    }
}
```

Output

```
Average value for the array elements is : 7.0
```

7. What is the output of the following program?

```
public class Ex7
{    public static void main(String[] args)
     {      int arr[][] = {{1,2,3,4},{3,2},{4,6,1}};
            System.out.println(arr.length);
            System.out.println(arr[0].length);
            System.out.println(arr[1].length);
            System.out.println(arr[0][1][2]);
            System.out.println(arr[2][1]++);
            System.out.println(arr[2][1]);
     }
}  // Sample String Examples
```

```
public class Ch2Pg56SubString {
   public static void main(String[] args) {
       String s;
       s = "Espresso";
    System.out.println(s.substring(2, 7) ); //press
    System.out.println(s.substring( 6, 8 ) ); //so
    System.out.println(s.substring( 0, 8 ) );//  Espresso
    System.out.println(s.substring( 1, 5) );  //spre
    System.out.println(s.length() );    // 8
  }
}
```

```
public class Ch2Pg58IndexOf {
   public static void main(String[] args) {
     String text = "I Love Java and Java loves me.";
      System.out.println( text.indexOf("J"));   //7
      System.out.println(  text.indexOf("love"));  //21
      System.out.println(  text.indexOf("ove"));    //3
      System.out.println(  text.indexOf("ME"));   //-1
    }
 }
```

```
    String text = "I Love Java and Java loves me.";
     System.out.println( text.indexOf("J"));   //7
     System.out.println(  text.indexOf("love"));  //21
     System.out.println(  text.indexOf("ove"));    //3
     System.out.println(  text.indexOf("ME"));   //-1
    }
```

```
    String text1 = "Jon";
    String text2 = "Java";
    System.out.println( text1 + text2); //JonJava
    System.out.println(  text1 + " " + text2); //Jon Java
    System.out.println(  "How are you, " +   text1    +
                         "?");  // How are you,  Jon?
```