Arrays

12

12.1 Arrays

12.1.1 The Structure of an Array

An **array** is a finite sequence of elements of some common data type, where an element of an array can be accessed by combing the name given to the array and the position of the element in the sequence, called **index**. Arrays are reminiscent of number sequence. In mathematics, an element of a number sequence is specified by the name of the sequence and the subscript representing the position of the element in the sequence, e.g., n_0 and n_9. In Java, numbers surrounded by square brackets specify positions and the positions always start from 0.

To declare an array data type, a pair of matching square brackets is attached as a suffix, as shown next. The first is an array of `int` values, the second is an array of `String` values, and the last declaration is an array of `double` values.

```
1 int[] myIntegers;
2 String[] myWords;
3 double[] someNumbers;
```

Arrays are objects, so before accessing their individual elements, they must be instantiated. An array is instantiated with its number of elements. We call the number of elements in an array the **length** or **size** of the array. The length specification appears inside a pair of square brackets `[]` after the type specification of the individual elements. Examples of the length specification are shown next. Line 4 instantiates a 10-element array of `int` values, Line 5 instantiates a 17-element array of `String` values, and Line 6 instantiates a 40-element array of `double` values.

```
1 int[] myIntegers;
2 String[] myWords;
3 double[] someNumbers;
4 myIntegers = new int[ 10 ];
5 myWords = new String[ 17 ];
6 someNumbers = new double[ 40 ];
```


M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_12

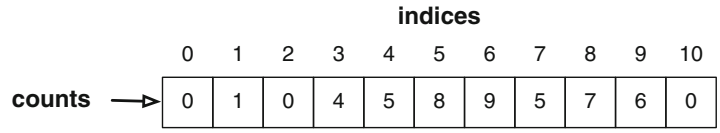

Fig. 12.1 A view of an array. The array has length 11 and the indexes are 0 through 10

Given an array that has been instantiated, we can obtain the number of elements in the array by attaching `.length` to the name of the array. Unlike the method `length` of `String`, the `.length` attached to an array is something called an instance variable (we will learn in detail what instance variables are in Chap. 16) that is publicly accessible but cannot be modified.

An individual element (or a slot) in an array can be specified with its name followed by a pair of brackets that contains inside the index to the element, e.g., `a[ 7 ]` and `b[ 9 ]`. Since the indexes in arrays always start from 0, given an array of some `N` elements, the index of the last element is `N - 1`. This means that if `x` is the name of an array, its elements are thus `x[ 0 ], ..., x[ x.length - 1 ]`.

Using the same examples as before, the code:

```
1 int[] myIntegers;
2 String[] myWords;
3 double[] someNumbers;
4 myIntegers = new int[ 10 ];
5 myWords = new String[ 17 ];
6 someNumbers = new double[ 40 ];
7 System.out.println( myIntegers.length );
8 System.out.println( myWords.length );
9 System.out.println( someNumbers.length );
```

produces the output:

```
1 10
2 17
3 40
```

We often use a drawing like the one in Fig. 12.1 to visualize an array. Each square is an element of the array and `counts` is the name of the array. The numbers appearing above the elements are indices to the elements.

12.1.2 Computing the Number of Occurrences

12.1.2.1 Computing Distributions

Consider receiving an indefinite number of exam scores and computing the score distribution. The scores are integers between 0 and 100, so there are 101 possible scores. We group the scores into eleven bins: 0..9, 10..19, ..., 90..99, and 100, where the last bin is exclusively for 100, the highest possible score. The task is to count, for each bin, the scores that belong to it, and then report the counts and print a histogram.

We use an `int` array named `counts` for recording the counts.

```
    int[] counts = new int[ 11 ];
```

The elements of `counts` are assigned indexes from 0 to 10. Given a score, `score`, the index of the bin that `score` belongs to can be obtained by the quotient by 10. We store this quotient in an `int` variable, `position`, and then increase `counts[ position ]` by 1. The statement `counts` is treated as an `int` variable.

```
1 int position = score / 10;
2 counts[ position ] ++;
```

At the time of instantiation, each element of the array is initialized with the default value of its data type. The array `counts` is an array of `int` data, so the initial value of its elements is 0. No scores have been entered yet at the time of instantiation, so the counts must be set to 0 for all the bins before starting to receive the scores. Thus, there is no need for explicit initialization.

Here is the first part of the source code, where the program receives the scores. The program declares an `int` array `counts` and instantiates it as an 11-element array (Line 8). The program asks the user to start entering numbers and enter CTRL-D to report the end of input (Lines 10 and 11). To receive input, the program uses a while-loop with `keyboard.hasNext()` as the continuation condition. As we saw in Chap. 11, `keyboard.hasNext()` returns `false` only when the user enters CTRL-D at the start of the input sequence. In the loop-body, the program receives a new score using `keyboard.nextInt()`, and then stores it in the variable `score` (Line 14). The program determines the index of the bin in the manner we have discussed, and then stores it in the variable `position` (Line 15). The program then increases `counts[ position ]` by 1 (Line 16).

```
1  import java.util.*;
2  public class ScoreDist
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7
8      int[] counts = new int[ 11 ];
9
10     System.out.println( "Enter scores in the range 0..100." );
11     System.out.println( "When you are done, enter CTRL-D." );
12     while ( keyboard.hasNext() )
13     {
14       int score = keyboard.nextInt();
15       int position = score / 10;
16       counts[ position ] ++;
17     }
18
```

Listing 12.1 A program for computing the distribution of scores in range 0..100 (part 1).

The program must produce two types of output from the counts thus obtained, a numerical output and a bar-histogram. In both outputs, the program uses the numbers 0, 10, ..., 90, 100 as the representatives of the bins. These representative numbers are equal to ten times the indexes to the bins. In the bar-histogram output, the program prints a horizontal line of hash marks for each bin. The lengths of the lines are equal to the counts. The histogram can be generated using a double for-loop.

```
19      System.out.println( "The histogram:" );
20      for ( int index = 0; index <= 10; index ++ )
21      {
22        System.out.printf( "%3d:%d%n", index * 10, counts[ index ] );
23      }
24
25      System.out.println( "The bar histogram:" );
26      for ( int index = 0; index <= 10; index ++ )
27      {
28        System.out.printf( "%3d:", index * 10 );
29        for ( int i = 1; i <= counts[ index ]; i ++ )
30        {
31          System.out.print( "#" );
32        }
33        System.out.println();
34      }
35    }
36  }
```

Listing 12.2 A program for computing the distribution of scores in range 0..100 (part 2).

An execution example of the code appears next:

```
1  Enter scores in the range 0..100.
2  When done, enter either CTRL-D.
3  0 11 22 33 44 55 66 77 88 99 89 97 96 95 100 79 77 75 56 51 51 68
4  The histogram:
5    0:1
6   10:1
7   20:1
8   30:1
9   40:1
10  50:4
11  60:2
12  70:4
13  80:2
14  90:4
15 100:1
16 The bar histogram:
17   0:#
18  10:#
19  20:#
20  30:#
21  40:#
22  50:####
23  60:##
24  70:####
25  80:##
26  90:####
27 100:#
```

We now change the histogram to a cumulative histogram, where the cumulative count at index `i` is the number of all scores satisfying `score / 10 <= i`. In other words, each score value, `score`, is counted in all the bins whose indexes are greater than or equal to `score / 10`.

We obtain the cumulative counts in an array named `cumulative` in two step. We obtain the noncumulative counts, `counts`, as before, and then, for each value of `index` between 0 and 10, calculate `cumulative[ index ]` as the total of `counts[ 0 ], ..., counts[ index ]`. In this two-step calculation, we use the fact that for each value of `index` between 1 and 10, `cumulative[ index ]` is equal to `cumulative[ index - 1 ] + counts[ index ]`, and compute the cumulative counts using a series of additions as shown next. Since 0 is the smallest index value, `cumulative[ 0 ]` is equal to `counts[ 0 ]`.

```
20      cumulative[ 0 ] = counts[ 0 ];
21      for ( int index = 1; index <= 10; index ++ )
22      {
23        cumulative[ index ] = cumulative[ index - 1] + counts[ index ];
24      }
```

The next is the source code for producing a cumulative histogram. The program uses two arrays (Lines 8 and 9). The program prints the same instruction as before (Lines 11 and 12), and generates the counts in the same manner as before (Lines 13–18).

```
1  import java.util.*;
2  public class ScoreDistCumulative
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7
8      int[] counts = new int[ 11 ];
9      int[] cumulative = new int[ 11 ];
10
11     System.out.println( "Enter scores in the range 0..100." );
12     System.out.println( "When you are done, enter CTRL-D." );
13     while ( keyboard.hasNext() )
14     {
15       int score = keyboard.nextInt();
16       int position = score / 10;
17       counts[ position ] ++;
18     }
19
```

Listing 12.3 A program for computing a cumulative distribution (part 1).

After computing the cumulative counts, the program produces a cumulative bar-graph using the same algorithm as before (Lines 27–35).

```
20      cumulative[ 0 ] = counts[ 0 ];
21      for ( int index = 1; index <= 10; index ++ )
22      {
23        cumulative[ index ] = cumulative[ index - 1] + counts[ index ];
24      }
25
26      System.out.println( "The bar histogram:" );
27      for ( int index = 0; index <= 10; index ++ )
28      {
29        System.out.printf( "%3d:", index * 10 );
30        for ( int i = 1; i <= cumulative[ index ]; i ++ )
31        {
32          System.out.print( "#" );
33        }
34        System.out.println();
35      }
36    }
37  }
```

Listing 12.4 A program for computing a cumulative distribution (part 2).

With the same input sequence as before, the program produces the following output:

```
1  % java ScoreDistCumulative
2  Enter scores in the range 0..100.
3  When done, enter either CTRL-D.
4  34 5 6 7 88 99 100 98 80 78 67 88 98 99 87 77 61 76 38 48 95 85
5  The bar histogram:
6    0:###
7   10:###
8   20:###
9   30:#####
10  40:######
11  50:######
12  60:########
13  70:###########
14  80:################
15  90:####################
16 100:####################
```

12.1.2.2 Simulating the Action of Throwing Two Dice

Our next example is a program for computing, by simulation, the probability distribution of the number that is "rolled" by "throwing" two fair dice. The number that is "rolled" ranges from 2 to 12. The program simulates the action of throwing two dice repeatedly, and computes the frequencies that these numbers are "rolled". At the end, the frequencies are converted to percentages. The program receives how many "rolls" must be made from the user.

The program uses a 13-element array of `double` data named `rates` to record the frequencies. For each number `i` between 2 and 12, the program uses `rates[ i ]` to record the number of times `i` is "rolled". The counts are converted to percentages using the same array, so the data type of the elements of the array are not `int` but `double`.

Here is the declaration of the array:

```
    double[] rates = new double[ 13 ];
```

Previously, in Sect. 7.5, we used the formula `(int)( Math.random() * 6 ) + 1` to simulate the throw of one die. Since we use a pair of dice, we use the sum of two copies of this formula:

```
1       int pos = (int)( Math.random() * 6 ) + 1
2               + (int)( Math.random() * 6 ) + 1;
```

By combining the two occurrences of `+1`, we obtain:

```
1       int pos = (int)( Math.random() * 6 )
2               + (int)( Math.random() * 6 ) + 2;
```

We cannot simplify the formula further. In other words, neither

```
    int pos = (int)( Math.random() * 6 + Math.random() * 6 ) + 2;
```

nor

```
    int pos = (int)( Math.random() * 12 ) + 1;
```

are equivalent to the formula used in the program.

Here is a source code for the program. As mentioned earlier, the program uses a 13-element array of `double`, `rates` (Line 6). The program receives the number of times that the dice are "rolled" from the user, and stores it in the variable `rounds` (Lines 10 and 11). The program uses a for-loop to produce the required number of rolls (Line 13). At each round of the for-loop, the number generated is stored in an `int` variable, `pos` (Lines 15 and 16). The value of `pos` is between 2 and 12. The program increases `rates[ pos ]` by 1.

```
1  import java.util.*;
2  public class TwoDice
3  {
4    public static void main( String[] args )
5    {
6      double[] rates = new double[ 13 ];
7
8      Scanner keyboard = new Scanner( System.in );
9
10     System.out.print( "Enter the no. of trials: " );
11     int rounds = keyboard.nextInt();
12
13     for ( int i = 1; i <= rounds; i ++ )
14     {
15       int pos = (int)( Math.random() * 6 )
16               + (int)( Math.random() * 6 ) + 2;
17       rates[ pos ] ++;
18     }
19
```

Listing 12.5 A program that simulated the action of throwing two dice (part 1).

After finishing the simulation, the program converts the counts to percentages. The program accomplishes this by multiplying each count by 100 and then dividing it by `rounds` (Lines 20–23). Since the data type of the elements of the array is `double`, the value of `rates[ pos ] * 100` is already `double`, so although `rounds` is an `int` value, the division is correctly calculated. To print the percentage values (Lines 25–28), the program uses the `printf` format of `%2d:%6.2f%%%n`, where the `%%` instructs to print the percent sign. Since the numbers that are "rolled" are between 2 and 12, the iterations in the for-loops (Lines 20 and 25) start from 2, ignoring the indexes 0 and 1. In fact, the first two elements `rates[ 0 ]` and `rates[ 1 ]` exist, but are not accessed at all.

```
20      for ( int pos = 2; pos <= 12; pos ++ )
21      {
22        rates[ pos ] = rates[ pos ] * 100 / rounds;
23      }
24
25      for ( int pos = 2; pos <= 12; pos ++ )
26      {
27        System.out.printf( "%2d:%6.2f%%%n", pos, rates[ pos ] );
28      }
29    }
30  }
```

Listing 12.6 A program that simulated the action of throwing two dice (part 2).

Here is the result of running the program with 10,000 throws:

```
1  Enter the no. of trials: 10000
2   2:  2.83%
3   3:  5.43%
4   4:  8.13%
5   5: 11.25%
6   6: 14.11%
7   7: 16.90%
8   8: 13.67%
9   9: 10.82%
10 10:  8.56%
11 11:  5.41%
12 12:  2.89%
```

Here is the result of running the program with one million throws:

```
1  Enter the no. of trials: 1000000
2   2:  2.77%
3   3:  5.54%
4   4:  8.36%
5   5: 11.07%
6   6: 13.88%
7   7: 16.63%
8   8: 13.90%
9   9: 11.17%
10 10:  8.33%
11 11:  5.56%
12 12:  2.78%
```

12.1.2.3 Initialization of an Array

As mentioned earlier, when an array of some type `T` is instantiated, the initial value of each element of the array is the default value for the type `T`. Specifically, the initial value is 0 for each number type, the character with index 0 for `char` (often denoted as `\0`), `false` for `boolean`, and `null` for all other types. The value `null` is printed as the `String` literal `"null"` when processed with `System.out.print`, `printf`, or `println`. `null` is not an object, so no instance method can be applied to it. For example, the code

```
1  String[] data = new String[ 10 ];
2  System.out.println( data[ 0 ].length() );
```

produces a run-time error `NullPointerException` at the second line.

12.1.3 `ArrayIndexOutOfBoundsException`

An attempt to access an element of an array using an invalid index leads to a run-time error `ArrayIndexOutOfBoundsException`.

Suppose we have a program that receives a series of `String` values from the user, stores the elements of the series in an array, and then interactively recalls an element of the data at the index the user specifies. The interaction with the user is repeated using a do-while loop that is terminated when the user enters a negative index for a query. The elements of the series may contain the white space character, so the program uses `nextLine` to read the elements of the series. While any negative value is interpreted as the signal for termination, the program treats any nonnegative value as a valid index. Given such a program, the user may enter a large index value to make `ArrayIndexOutOfBoundsException` occur.

A source code for this program is shown next. The programs asks for the number of elements in the series at the start (Line 8). To receive an integer input from the user, we have exclusively used `nextInt` in the past. However, now that the elements of the array are to be read using `nextLine`, it is a good idea to read every input with `nextLine`. Unfortunately, there is no `Scanner` method that reads an input line, coverts it to an integer, and returns the integer. Fortunately, we can use the method `Integer.parseInt` for converting a `String` data to an integer. `Integer.parseInt` is a static method of class `Integer`. If the `String` does not represent an integer, for example, `"#$%"`, the method produces a run-time error and the execution halts at that point. The program uses `Integer.parseInt` to convert the input line to an integer, and stores the value in an `int` variable, `dim` (Line 9).

The program then runs as follows: The program instantiates `data` as an array of `String` data having length `dim` (Line 11). The program then uses a for-loop that iterates the sequence `0, ..., dim - 1` using a variable named `index`, and receives the elements of the array, where each element is read with `nextLine` (Lines 14–18). In the do-while loop (Lines 19–27), the program receives an index to an element to recall (Lines 21–22), and if the index is nonnegative (Line 23), prints the data stored at the index (Line 25). The continuation condition of the do-while loop is `index >= 0`, so the loop terminates when `index < 0`.

```
1  import java.util.*;
2
3  public class StringArrayPlain
4  {
5    public static void main( String[] args )
6    {
7      Scanner keyboard = new Scanner( System.in );
8      System.out.print( "Enter the no. of data: " );
9      int dim = Integer.parseInt( keyboard.nextLine() );
10
11     String[] data = new String[ dim ];
12
13     int index;
14     for ( index = 0; index < dim; index ++ )
15     {
16       System.out.printf( "Enter line at position %d: ", index );
17       data[ index ] = keyboard.nextLine();
18     }
19     do
20     {
21       System.out.print( "Enter position (negative to quit): " );
22       index = Integer.parseInt( keyboard.nextLine() );
23       if ( index >= 0 )
24       {
25         System.out.printf( "At %d we have %s%n", index, data[ index ] );
26       }
27     } while ( index >= 0 );
28   }
29 }
```

Listing 12.7 A program that builds an array of `String` data and then presents its contents.

In the following execution example, the user enters a non-existing index 9 for a 9-element array, thereby makes `ArrayIndexOutOfBoundsException` occur.

```
1  Enter the no. of data: 9
2  Enter line at position 0: Tomato
3  Enter line at position 1: Green Pepper
4  Enter line at position 2: Lettuce
5  Enter line at position 3: Pickle
6  Enter line at position 4: Black Olive
7  Enter line at position 5: Jalapeno Pepper
8  Enter line at position 6: Honey Mustard
9  Enter line at position 7: Mayonnaise
10 Enter line at position 8: Hot Sauce
11 Enter position (negative to quit): 0
12 The value at 0 is Tomato
13 Enter position (negative to quit): 2
14 The value at 2 is Lettuce
15 Enter position (negative to quit): 3
16 The value at 3 is Pickle
17 Enter position (negative to quit): 9
18 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 9
19   at StringArrayPlain.main(ArrayDataPlain.java:23)
```

The run-time error `ArrayIndexOutOfBounds` can be prevented from happening by changing the condition in Line 23 to `if ( index >= 0 && index < dim )`.

12.2 Relative Indexing

12.2.1 The Concept of Relative Indexing

The program that simulates the "rolls" of two dice, uses an array consisting of 13 slots, ignoring the first two slots. The code that appears next is an alternate one that uses an array of exactly 11 slots. In the program, we think of a slot index `i` as representing `i + 2` as the number that is "rolled". The changes from the previous program are highlighted. They are:

- The length of the array has been changed to 11 (Line 6).
- The `+ 2` in the formula for determining the number that is "rolled" has been removed, since 2 was to be subtracted from the number generated by the formula (Lines 15 and 16).
- The range of `index` has been changed to `0, ..., 10` in the for-loops (Lines 20 and 25).
- The number represented by the index value `pos` has been changed to `pos + 2` (Line 27).

```
1  import java.util.*;
2  public class TwoDiceExact
3  {
4    public static void main( String[] args )
5    {
6      double[] rates = new double[ 11 ];
7
8      Scanner keyboard = new Scanner( System.in );
9
10     System.out.print( "Enter the no. of trials: " );
11     int rounds = keyboard.nextInt();
12
13     for ( int i = 1; i <= rounds; i ++ )
14     {
15       int pos = (int)( Math.random() * 6 )
16               + (int)( Math.random() * 6 );
17       rates[ pos ] ++;
18     }
19
20     for ( int pos = 0; pos <= 10; pos ++ )
21     {
22       rates[ pos ] = rates[ pos ] * 100 / rounds;
23     }
24
25     for ( int pos = 0; pos <= 10; pos ++ )
26     {
27       System.out.printf( "%2d:%6.2f%%%n", pos + 2, rates[ pos ] );
28     }
29   }
30 }
```

Listing 12.8 A program for simulating the action of throwing two dice that uses an array of eleven elements.

The idea we used can be summarized as: *use the relative distance from a fixed number as an array index.* We call this idea **relative indexing**. To obtain a relative index from a given index, we subtract a fixed number, called the **base index**. In the above simulation, the base index is 2. Relative indexing allows programmers to avoid using slots that are not part of the computation.

12.2.2 Calculating the BMI for a Range of Weight Values

Consider computing the BMI value for a fixed height (in inches) and a range of integer weight values (in pounds), where the range is specified with its minimum and maximum. Suppose the minimum and the maximum are stored in `int` variables, `minWeight` and `maxWeight`. Since the weight values are integers, we can use an array of `double` data with `maxWeight + 1` elements, and use only the elements of array at indexes between `minWeight` and `maxWeight`. With the use of relative indexing, the array length can be reduced to `maxWeight - minWeight + 1`, where the element at an index `i` represents the BMI for the weight `minWeight + i`.

The following program uses this relative indexing. The program receives the minimum and maximum weights from the user, and stores them in variables, `minWeight` and `maxWeight` (Line 8 and 9). The program receives the height from the user and stores it in a variable, `height` (Line 10). The program uses `minWeight` as the base index. The size of the range of weights, `maxWeight - minWeight + 1`, is stored in an `int` variable, `size` (Line 11). The program instantiates an array of `double` data, `bmi` (Line 12). Then, for each index value `i` between 0 and `size - 1` (Line 13), the program stores the BMI value for the height equal to `height` and the weight equal to `minWeight + i` in `bmi[ i ]` (Line 15). At the end, the program prints the BMI values thus calculated along with their weights (Lines 17–20).

```
1  import java.util.*;
2  public class BMIFixedHeight
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7      System.out.print( "Minimum weight, maximum weight, height: " );
8      int minWeight = keyboard.nextInt();
9      int maxWeight = keyboard.nextInt();
10     double height = keyboard.nextDouble();
11     int size = maxWeight - minWeight + 1;
12     double[] bmi = new double[ size ];
13     for ( int i = 0; i < size; i ++ )
14     {
15       bmi[ i ] = 703.0 * ( minWeight + i ) / height / height;
16     }
17     for ( int i = 0; i < size; i ++ )
18     {
19       System.out.printf( "%3d:%5.2f%n", ( minWeight + i ), bmi[ i ] );
20     }
21   }
22 }
```

Listing 12.9 A program that computes the BMI value for a range of weights.

Here is an execution example of the code.

```
1  Minimum weight, maximum weight, height: 155 167 67
2  155:24.27
3  156:24.43
4  157:24.59
5  158:24.74
6  159:24.90
7  160:25.06
```

```
161:25.21
162:25.37
163:25.53
164:25.68
165:25.84
166:26.00
167:26.15
```

12.2.3 Counting the Occurrences of Characters

Next, consider computing the number of times each letter of the alphabet appears in an input line, after converting each uppercase letter to lowercase. The program receives an input line from the user, and then converts it to all lowercase with the method `toLowerCase()`. Since there are 26 letters in the lowercase alphabet, the program uses a 26-element `int` array, `count`, for counting the occurrences of the letters, where the elements 0, . . . , 25 correspond to the letters `'a', ..., 'z'`. In the ASCII table, the 26 letters occupy consecutive positions starting from 97, so subtracting 97 from a lowercase letter produces its position in the array `count`, and adding 97 to an index value produces the position of the corresponding letter in the ASCII table. The number 97 can be substituted with the letter `'a'` wherever needed.

The program receives an input from the user, stores it in a variable named `line` (Line 8), and then generates its lowercase version, `lineLower` (Line 9).

```
1  import java.util.*;
2  public class LetterCount
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7      System.out.print( "Enter: " );
8      String line = keyboard.nextLine();
9      String lineLower = line.toLowerCase();
10
```

Listing 12.10 A program that counts the occurrences of letters (part 1). The part responsible for receiving input.

The program then instantiates an array of 26 elements, `count` (Line 11). The program uses a for-loop to examine the characters of `lineLower` (Line 12). The program stores the character being examined in a `char` variable named `c` (Line 14). The program then obtains the relative index of `c` in a variable named `index` by subtracting `'a'` from `c` (Line 15). If `c` is a lowercase letter, then the value of `index` must be between 0 and 25. The program checks if the value of `index` is between 0 and 25 (Line 16) and if so, increases `count[ index ]` by 1 (Line 18).

```
11      int[] count = new int[ 26 ];
12      for ( int pos = 0; pos <= lineLower.length() - 1; pos ++ )
13      {
14        char c = lineLower.charAt( pos );
15        int index = c - 'a';
16        if ( index >= 0 && index <= 25 )
17        {
18          count[ index ] ++;
19        }
20      }
21
```

Listing 12.11 A program that counts the occurrences of letters (part 2). The part responsible for generating counts.

To present the result, the program prints the input line (Line 22), the lowercase version (Line 23), and a header line (Line 24). The program then uses a for-loop that iterates over the sequence `0, ..., 25` with a variable named `index` (Line 25). The character represented by `index` is obtained by converting the relative index to its absolute index with the addition of `'a'`, and then casting of `(char)` on the absolute index. In other words, `(char)( 'a' + index )` produces the letter (Line 28). The count corresponding to the letter is `count[ index ]` (Line 28). The program prints the letter and the count using `printf` with the format of `"%6c: %2d%n"` (Line 27). The format instructs to allocate six character spaces to the letter and two character spaces to the count.

```
22      System.out.printf( "Input: %s%n", line );
23      System.out.printf( "Lower: %s%n", lineLower );
24      System.out.println( "Letter:count" );
25      for ( int index = 0; index <= 25; index ++ )
26      {
27        System.out.printf( "%6c: %2d%n",
28            (char)( 'a' + index ), count[ index ] );
29      }
30    }
31  }
```

Listing 12.12 A program that counts the occurrences of letters (part 3). The part responsible for printing the result.

Next is an execution example of the code:

```
1  Enter: I can write programs in Java.  I want to learn more.
2  Input: I can write programs in Java.  I want to learn more.
3  Lower: i can write programs in java.  i want to learn more.
```

```
Letter:count
     a:  6
     b:  0
     c:  1
     d:  0
     e:  3
     f:  0
     g:  1
     h:  0
     i:  4
     j:  1
     k:  0
     l:  1
     m:  2
     n:  4
     o:  3
     p:  1
     q:  0
     r:  5
     s:  1
     t:  3
     u:  0
     v:  1
     w:  2
     x:  0
     y:  0
     z:  0
```

In this example, since the input line is short, several letters have 0 as their counts. Since each letter-count pair does not require more than 12 character spaces, we can expect to be able to reduce the number of output lines by skipping letters whose counts are 0 and by printing multiple letter-count pairs in each line of output.

The program `LetterCountNeat` prints the result using this idea. Both `LetterCount` and `LetterCountNeat` have the same code for receiving the input and generating counts (except for the class declaration), so only the remaining part of the program for `LetterCountNeat` is presented here.

`LetterCountNeat` prints up to four letter-count pairs in one line. The header for the letter-count output has been revised to:

```
"Letter:count Letter:count Letter:count Letter:count"
```

(Lines 24 and 25). To emphasize that a count is associated with the letter it is printed together, the format for the count has been changed to `%-6d`. This means that the number will be printed immediate after the colon.

To print multiple letter-count pairs while ignoring letters that did not appear, the program uses a new `int` variable named `p`. The variable `p` represents the number of letter-count pairs that have been printed so far. We scan the elements, as before, using the variable `index`. Each time a letter with a positive count is encountered, the program takes the following action: (a) it prints the letter-count pair, (b) it increases `p` by 1, and (c) if the value of `p` is a multiple of 4, it prints the newline. Furthermore, to ensure that the output of the program ends with a newline, if the value of `p` is not a multiple of 4 after exiting the loop, the program prints the newline.

The next code implements this idea. Line 29 is for checking if the count is positive, Lines 31 and 32 are for printing the letter-count pair, Line 33 is for increasing the value of `p`, and Lines 34–37 are for adding the newline after printing four pairs. Lines 40–43 handle the last newline.

```
22      System.out.printf( "Input: %s%n", line );
23      System.out.printf( "Lower: %s%n", lineLower );
24      System.out.println(
25          "Letter:count Letter:count Letter:count Letter:count" );
26      int p = 0;
27      for ( int index = 0; index <= 25; index ++ )
28      {
29        if ( count[ index ] > 0 )
30        {
31          System.out.printf( "%6c:%-6d",
32              (char)( 'a' + index ), count[ index ] );
33          p ++;
34          if ( p % 4 == 0 )
35          {
36            System.out.println();
37          }
38        }
39      }
40      if ( p % 4 != 0 )
41      {
42        System.out.println();
43      }
44    }
45  }
```

Listing 12.13 A new version of the program that counts the occurrences of letters.

The output of the new program with the same input as before is shown next:

```
Enter: I can write programs in Java.  I want to learn more.
Input: I can write programs in Java.  I want to learn more.
Lower: i can write programs in java.  i want to learn more.
Letter:count Letter:count Letter:count Letter:count
     a:6          c:1          e:3          g:1
     i:4          j:1          l:1          m:2
     n:4          o:3          p:1          r:5
     s:1          t:3          v:1          w:2
```

Here is the output with another input:

```
Enter: Old MacDonald Had a Farm
Input: Old MacDonald Had a Farm
Lower: old macdonald had a farm
Letter:count Letter:count Letter:count Letter:count
     a:5          c:1          d:4          f:1
     h:1          l:2          m:2          n:1
     o:2          r:1
```

12.3 Arrays of `boolean` Data

We often use a `boolean` variable as a check mark and an array of `boolean` data as a series of check marks. Here, we study the use of an array of `boolean` data using a source code for a program that finds prime numbers.

A non-zero integer m is a **divisor** of another integer n if the remainder of n divided by m is 0. If m is a divisor of n, we say that n is **divisible** by m. A positive integer $n \geq 2$ is a **prime number** (or

simply a **prime**) if 1 and n are the only positive divisors of n. In other words, n is prime if and only if for all m, $2 \leq m \leq n - 1$, the remainder of n divided by m is 0. An integer $n \geq 2$ that is not a prime number is called a **composite number**. The 16 smallest prime numbers are as follows:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 33, 37, 41, 43, 47$$

It is known that there are infinitely many prime numbers.

Every positive integer $n \geq 2$ can be expressed as the product of prime numbers, where the same prime number may appear multiple times. This decomposition is unique, if the order of appearance of the prime numbers is ignored. For example, 100 is the product of prime numbers 2, 2, 5, and 5. The primality of a number $n \geq 3$ thus can be tested by examining if any prime number less than n divides n. In other words,

($\star$) a positive number $n \geq 3$ is a prime number if and only if no prime number p between 2 and $n - 1$ divides n.

Consider discovering, given some maximum number, `limit`, all the prime numbers between 2 and `limit`. The property ($\star$) suggests the following algorithm: Repeat (a) and (b) for all values of `n` starting with 2 and ending with `limit`:

(a) Using a loop, test if `n` is divisible by any prime number less than `n`.
(b) If `n` is divisible by a prime number less than `n`, record `n` as composite; otherwise, record `n` as prime.

We implement this algorithm using an array of `boolean` values, `flags`. `flags` has `limit + 1` elements. At the end of the calculation, we establish a condition: for each index `i >= 2`, `flags[ i ]` is `true` if and only if `i` is prime.

There are multiple ways to implement the determination of a primality of `n` by testing if any smaller prime number divides `n` (such a strategy is called **trial division**). Here is a simple one. We tentatively store `true` to `flags[ n ]`. We then execute trial divisions by the primes between 2 and `n - 1`. If one of the primes in that range is found to be a divisor of `n`, we change the value of `flags[ n ]` to `false`. The idea can be encoded as follows:

```
1       flags[ n ] = true;
2       for ( int m = 2; m <= n - 1; m ++ )
3       {
4         if ( flags[ m ] && n % m == 0 )
5         {
6           flags[ n ] = false;
7         }
8       }
```

Since the value of `flags[ n ]` does not change from `false` to `true`, the loop can be terminated as soon as the value changes to `false`:

```
1       flags[ n ] = true;
2       for ( int m = 2; m <= n - 1; m ++ )
3       {
4         if ( flags[ m ] && n % m == 0 )
5         {
6           flags[ n ] = false;
7           break;
8         }
9       }
```

If `break` is executed, the final value of `m` is strictly less than `n`; otherwise, the final value of `m` is equal to `n`. Thus, the value of `flags[ n ]` is equivalent to the condition `m < n`.

```
1       for ( int m = 2; m <= n - 1; m ++ )
2       {
3         if ( flags[ m ] && n % m == 0 )
4         {
5           break;
6         }
7       }
8       flags[ n ] = m == n;
```

Using a while-loop, this can be written as:

```
1       int m = 2;
2       while ( m <= n - 1 && !( flags[ m ] && n % m == 0 ) )
3       {
4         m ++;
5       }
6       flags[ n ] = m == n;
```

Because of the truncation rule in conditional evaluation (see Sect. 6.2.4), the second term `!(flags[ m ] && n % m == 0 )` is evaluated only if `m < n`. By using DeMorgan's laws, we obtain:

```
1       int m = 2;
2       while ( m <= n - 1 && ( !flags[ m ] || n % m != 0 ) )
3       {
4         m ++;
5       }
6       flags[ n ] = m == n;
```

This is the version that is used in the next program `PrimeFinding`.

The first part of the code executes the trial division algorithm as we have developed. The program receives the value of `limit` from the user (Lines 7 and 8), instantiates the array `flags` as `boolean[ limit + 1 ]` (Line 10). Lines 11–19 are for determining the elements of `flags`. When the value of `n` is equal to 2, no trivial division is made. Since the initial value of the array elements is `true` and 2 is a prime number, the programs works when `n` is equal to 2.

To print the prime numbers that have been discovered, the program uses the idea from `LetterCountNeat`. This time, the program prints each prime number with its position (represented by `p`) (Line 30). The primes appear in increasing order.

```
import java.util.*;
public class PrimeFinding
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter limit: " );
    int limit = keyboard.nextInt();
      //---- prime discovery
    boolean[] flags = new boolean[ limit + 1 ];
    for ( int n = 2; n <= limit; n ++ )
    {
      int m = 2;
      while ( m < n && ( !flags[ m ] || n % m != 0 ) )
      {
        m ++;
      }
      flags[ n ] = m == n;
    }
      //---- inventory
    int p = 0;
    for ( int n = 2; n <= limit; n ++ )
    {
      if ( flags[ n ] )
      {
        p ++;
        System.out.printf( "%5d:%-8d", p, n );
        if ( p % 4 == 0 )
        {
          System.out.println();
        }
      }
    }
    if ( p % 4 != 0 )
    {
      System.out.println();
    }
  }
}
```

Listing 12.14 A program that finds all prime numbers less than or equal to a given bound.

Here is an execution example of the code:

```
Enter limit: 1000
    1:2          2:3          3:5          4:7
    5:11         6:13         7:17         8:19
    9:23        10:29        11:31        12:37
   13:41        14:43        15:47        16:53
   17:59        18:61        19:67        20:71
   21:73        22:79        23:83        24:89
   25:97        26:101       27:103       28:107
   29:109       30:113       31:127       32:131
   33:137       34:139       35:149       36:151
   37:157       38:163       39:167       40:173
   41:179       42:181       43:191       44:193
   45:197       46:199       47:211       48:223
   49:227       50:229       51:233       52:239
```

```
15    53:241      54:251      55:257      56:263
16    57:269      58:271      59:277      60:281
17    61:283      62:293      63:307      64:311
18    65:313      66:317      67:331      68:337
19    69:347      70:349      71:353      72:359
20    73:367      74:373      75:379      76:383
21    77:389      78:397      79:401      80:409
22    81:419      82:421      83:431      84:433
23    85:439      86:443      87:449      88:457
24    89:461      90:463      91:467      92:479
25    93:487      94:491      95:499      96:503
26    97:509      98:521      99:523     100:541
27   101:547     102:557     103:563     104:569
28   105:571     106:577     107:587     108:593
29   109:599     110:601     111:607     112:613
30   113:617     114:619     115:631     116:641
31   117:643     118:647     119:653     120:659
32   121:661     122:673     123:677     124:683
33   125:691     126:701     127:709     128:719
34   129:727     130:733     131:739     132:743
35   133:751     134:757     135:761     136:769
36   137:773     138:787     139:797     140:809
37   141:811     142:821     143:823     144:827
38   145:829     146:839     147:853     148:857
39   149:859     150:863     151:877     152:881
40   153:883     154:887     155:907     156:911
41   157:919     158:929     159:937     160:941
42   161:947     162:953     163:967     164:971
43   165:977     166:983     167:991     168:997
```

12.4 Using Multiple Arrays

We often use multiple arrays having the same lengths to manipulate a sequence of elements, where each element consists of multiple data.

Consider writing a program that receives names (as `String` data) and scores (as `double` data) for some number of people. The user specifies the number of people, and then, for each person, enters the name and the score. The program records the names and scores, and retrieves them upon user's request. For retrieval, the user specifies a range of scores with its lower and upper bounds. The program then finds all the scores that are in the range, and then prints the scores with their associated names.

The program `NameScore` accomplishes this task. The initial part declares some variables that are used throughout the program. `num` is the number of people to appear on the list (Line 10), `names` and `scores` are the arrays to record the names and the scores (Lines 7 and 8). `flags` represents which persons' scores are in the range specified by the user (Line 9). Each time the user specifies a range of scores, the elements of `flags` are recalculated as follows: for each index `i`, `true` is stored in `flags[ i ]` if the value of `scores[ i ]` is in the specified range, and `false` is stored otherwise.

```
1  import java.util.*;
2  public class NameScore
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7      String[] names;
8      double[] scores;
9      boolean[] flags;
10     int num;
11
```

Listing 12.15 A program that receives names and scores and performs search (part 1).

Next, the program receives the number of people to appear on the list, and then stores the number in `num` (Lines 12 and 13). `num` is then used to instantiate the length for each of the three array instantiations (Lines 14–16). Then, using a for-loop, the program receives the names and the scores from the user (Lines 18–23).

```
12     System.out.print( "Enter the number of people: " );
13     num = keyboard.nextInt();
14     names = new String[ num ];
15     scores = new double[ num ];
16     flags = new boolean[ num ];
17
18     for ( int i = 0; i < num; i ++ )
19     {
20       System.out.printf( "Enter name and score for person %d: ", i );
21       names[ i ] = keyboard.next();
22       scores[ i ] = keyboard.nextDouble();
23     }
24
```

Listing 12.16 A program that receives names and scores and performs search (part 2).

The interaction for retrieval uses a `String` variable named `answer` (Line 25). The interaction is repeated indefinitely using a do-while loop (Line 26). At the end of the loop-body, the user is asked to respond to the question of whether or not the program should continue (Line 50). The response is stored in a `String` variable named `answer` (Line 51). If the response does not start with `'y'`, the loop is terminated (Line 52).

To retrieve scores, the program receives two `double` values from the user, and stores them in `double` variables, `low` and `high` (Lines 27–29). The program uses a variable named `count` to record the number of scores that have passed the test (Line 31). The program uses a for-loop that iterates over the sequence of valid index values with a variable named `i` (Line 32). For each index value of `i`, the program stores the value of `( scores[ i ] >= low && scores[ i ] <= high )` in `flags[ i ]` (Line 34), and then, if the value obtained is `true` (Line 35), increases the value of `count` by 1 (Line 37). The program announces the value of `count` after completing the recalculation of the elements in `flags` (Line 40). To report which scores are in the range, the program uses a for-loop again with the iteration variable `i` (Line 42). If `flags[ i ]` has the value of `true` (Line 44), the program prints `i`, `names[ i ]`, and `scores[ i ]` in one line using the format `"%3d %10s %6.2f%n"`. This means that three character spaces are allocated for `i`, ten character

spaces are allocated for `names[ i ]`, and six character spaces with exactly two digits after the decimal point are allocated for `scores[ i ]`.

```
25      String answer;
26      do {
27        System.out.print( "Enter the lower and upper limits: " );
28        double low = keyboard.nextDouble();
29        double high = keyboard.nextDouble();
30
31        int count = 0;
32        for ( int i = 0; i < num; i ++ )
33        {
34          flags[ i ] = ( scores[ i ] >= low && scores[ i ] <= high );
35          if ( flags[ i ] )
36          {
37            count ++;
38          }
39        }
40        System.out.println( "The count is " + count + "." );
41
42        for ( int i = 0; i < num; i ++ )
43        {
44          if ( flags[ i ] )
45          {
46            System.out.printf( "%3d %10s %6.2f%n",
47                i, names[ i ], scores[ i ] );
48          }
49        }
50        System.out.print( "Continue (y/n)? " );
51        answer = keyboard.next();
52      } while ( answer.startsWith( "y" ) );
53    }
```

Listing 12.17 A program that receives names and scores and performs search (part 3).

Here is an example of how the program works:

```
1  Enter the number of people: 15
2  Enter name and score for person 0: Anna 80.0
3  Enter name and score for person 1: Bill 90.5
4  Enter name and score for person 2: Christy 97.5
5  Enter name and score for person 3: Dave 79.5
6  Enter name and score for person 4: Emily 81.0
7  Enter name and score for person 5: Fred 86.0
8  Enter name and score for person 6: Gail 95.0
9  Enter name and score for person 7: Harry 92.5
10 Enter name and score for person 8: Irene 98.0
11 Enter name and score for person 9: Jack 99.0
12 Enter name and score for person 10: Kim 93.5
13 Enter name and score for person 11: Luis 74.5
14 Enter name and score for person 12: Mimi 76.0
15 Enter name and score for person 13: Nick 77.0
16 Enter name and score for person 14: Olga 84.5
17 Enter the lower and upper limits: 0 100
18 The count is 15.
19   0       Anna  80.00
20   1       Bill  90.50
21   2    Christy  97.50
22   3       Dave  79.50
```

```
  4       Emily  81.00
  5        Fred  86.00
  6        Gail  95.00
  7       Harry  92.50
  8       Irene  98.00
  9        Jack  99.00
 10         Kim  93.50
 11        Luis  74.50
 12        Mimi  76.00
 13        Nick  77.00
 14        Olga  84.50
Continue (y/n)? y
Enter the lower and upper limits: 0 50
The count is 0.
Continue (y/n)? y
Enter the lower and upper limits: 70 80
The count is 5.
  0       Anna  80.00
  3       Dave  79.50
 11        Luis  74.50
 12        Mimi  76.00
 13        Nick  77.00
Continue (y/n)? n
```

12.5 `String` Methods That Return an Array

In Chap. 9, we mentioned two `String` methods that each return an array, `toCharArray` and `split`. For a `String` data `w`, the method call `w.toCharArray()` returns an array of `char` that is equal to the character-by-character representation of `w`. The method `split` receives a `String` parameter `p`. For a `String` data `w` and a `String` parameter `p`, the method call `w.split( p )` returns an array of `String` generated by splitting `w` by viewing `p` as the delimiter. For example, suppose `w` is equal to the `String` literal `"-ab:cd::ef-"`. The return value of `w.toCharArray()` is an 11-element array:

```
{ '-', 'a', 'b', ':', 'c', 'd', ':', ':', 'e', 'f', '-' }
```

The return value of `w.split( "::" )` is a 2-element array:

```
{ "-ab:cd", "ef-" }
```

The return value of `w.split( ":" )` is a 4-element array:

```
{ "-ab", "cd", "", "ef-" }
```

and the return value of `w.split( "-" )` is a 3-element array:

```
{ "", "-ab:cd::ef-", "" }
```

We can use the method `split` when multiple data values appear in one line and the values are separated with some common delimiter. Suppose we need to convert a `String` data `line` into an `int` array, assuming that `line` consists of some integer values separated by tab-stops in between (for example, `"10\t20\t30"`). One approach to solving this problem is to read `line` twice using `Scanner` as follows:

```
Scanner sc = new Scanner( line );
int count = 0;
while ( sc.hasNext() )
{
  count ++;
  sc.next();
}
int[] data = new int[ count ];
Scanner sc = new Scanner( line );
for ( int i = 0; i < count; i ++ )
{
  data[ i ] = sc.nextInt();
}
```

Since the tokens are separated by tab-stops, `split` can be used as follows:

```
String[] tokens = line.split( "\t" );
int[] data = new int[ tokens.length ];
for ( int i = 0; i < tokens.length; i ++ )
{
  data[ i ] = Integer.parseInt( tokens[ i ] );
}
```

Summary

- An array is a series of data of some common data type.
- The declaration of an array type requires an empty pair of matching brackets at the end, such as `int[]` and `String[]`.
- An array data type is an object data type, so requires an instantiation.
- The instantiation of an array requires the specification of the number of elements in the series that it represents. The number is provided in a pair of brackets after he type of the individual elements, such as `new int[ 10 ]`.
- The value of the length specified in an instantiation must be nonnegative.
- For an array `x`, `x.length` returns the number of elements in `x`.
- An individual element in an array can be accessed by combining the name of the array and its index in the form: `name[ index ]`.
- When a program tries to access an element of an array at an invalid index, an run-time error `ArrayIndexOutOfBoundsException` occurs.
- Relative indexing is a concept that represents the absolute index with the combination of the base index and the distance from the base index.

Exercises

1. **Simple questions about an array** Suppose an array `dataValues` has been constructed using the following code fragment:

```
int[] dataValues = new int[20];
for ( int index = 0; index < 20; index ++ )
{
   dataValues[ index ] = index + 1;
}
```

State the value of each of the following after the execution of the code:
(a) `dataValues.length`
(b) `dataValues[10]`
(c) `dataValues[15] - dataValues[5]`

2. **An array of random numbers with no duplication** Write a program, `RandomNumberArray`, that generates a random array of `int` data whose length, `size`, is specified by the user. The elements in the array must satisfy the following conditions:
   - The range of the array elements is 0 through `10 * size - 1`.
   - No two elements in the array are equal to each other.

   The task can be accomplished by a series of, possibly repetitive, random number generation. To generate a random number at index `i`, the program generates a candidate according the first rule, and then checks whether the same number appears in any positions between 0 and `i - 1`. If the number already appears, the program generates a new random number. It repeats this process until a random number not appearing in positions 0 through `i - 1` is generated.

   Here is an execution example of the intended program:

```
Enter the size: 0:109
1:100
2:23
3:57
4:123
5:22
6:137
7:120
8:1
9:60
10:146
11:68
12:8
13:52
14:96
```

3. **The second maximum in an array** Write a program, `SecondMaximum`, that receives a positive integer from the user, generates an array of random real numbers between 0 and 10,000, whose length is equal to the integer that the user has entered, and then computes the second largest number appearing in the array. The program may fail to work properly if the number that the user enters is less than or equal to 0. If the number that the user enters is 1, the second largest number is undefined, but the program must report the only element of the array as the second largest number.
4. **Random string** Write a program, `RandomString`, that receives a positive integer from the user, and then generates a `String` of random characters whose length is equal to the integer that the user has specified. The program must use table look-up in the following manner. The program instantiates a 26-element array of `char` values whose elements are `'a'` through `'z'`. It then creates a random `String` data of the specified length by selecting, for each character position of the `String`, a random position between 0 and 25, and then appending the element in the table at that random position to the `String` being generated.

   Here is an execution example of the program:

```
Enter the length: 20
ylufdstaeldwbyvotdph
```

5. **The standard deviation of elements in an array** The standard deviation of a series of numbers, $a_1, \cdots, a_n$, is given by the following formula:

$$\sqrt{\frac{\sum_{i=1}^{n}(a_i - \text{the average})^2}{n-1}}$$

Write a program, `StdDev` that generates an array of random real numbers greater than or equal to 0 and less than 10,000, and then computes the standard deviation of the elements in the array. The user specifies the length of the array. Since the formula has $n - 1$ in the denominator, the length of the array must be greater than or equal to 2. To obtain an integer greater than or equal to 2, the program keeps asking the user to enter a value for the length until the value entered is greater than or equal to 2. After the calculation, the program must print the elements in the array, the average, and the standard deviation. Each real number, when printed, must be formatted by allocating four character spaces before the decimal point and ten digits after the decimal point.

The following is an execution example of the program:

```
Enter the length: 1
Enter the length: 0
Enter the length: 10
  0: 8695.0374293399
  1: 4425.0711045508
  2:  187.4407268688
  3: 6606.5239258968
  4: 8713.3827987525
  5: 1437.1711093935
  6: 2549.7630539290
  7: 7171.4009860844
  8: 6077.1461415774
  9: 7729.9647709803
average = 5359.2902047373
stddev = 3059.5779033118
```

6. **Counting decimals** Write a program, `CountDecimals`, that receives an indefinite number of tokens from the user, and then, in the series of tokens, computes the number of occurrences of each decimal. The program must report the count of each decimal on the screen at the end.

   The program may work as follows:

```
> CSC120
> CSC220
> BIL169H
> ECO101
> 0: 3
1: 4
2: 3
3: 0
4: 0
5: 0
6: 1
7: 0
8: 0
9: 1
```

   The `'>'` appearing at the start of the first four lines is the prompt.

7. **Modification to the prime number finding problem** The prime number discovery program we discussed in this chapter prints all the primes it has discovered. Write a new version, `PrimeFindingNew`, where the program produces an array of `int` data consisting of the prime numbers that have been discovered, and then prints the elements in the array.

One way to generate such an array is to collect all the indexes at which the elements of `flags` are `true`. To accomplish this, the program may scan the `flags` to obtain the number of times `true` appears as an element. This is the length of the array to be returned. After that, the program can instantiate an array of `int` data whose length is equal to the count that just has been obtained. Then, the program stores in the array the indexes at which the elements of `flags` are `true`.

8. **`toCharArray`** Write a program, `MyToCharArray`, that receives an input line from the user, converts it to an array of `char` values, and prints the elements in the array. The program must generate the output with six elements per line with three character spaces allocated to the indexes and one character space allocated for the element, as shown in the example below.

   The program may work as follows:

```
1  Enter your input: This is a test of toCharArray.
2    0:T,   1:h,   2:i,   3:s,   4: ,   5:i
3    6:s,   7: ,   8:a,   9: ,  10:t,  11:e
4   12:s,  13:t,  14: ,  15:o,  16:f,  17:
5   18:t,  19:o,  20:C,  21:h,  22:a,  23:r
6   24:A,  25:r,  26:r,  27:a,  28:y,  29:.
```

Programming Projects

9. **The distribution of token lengths** Write a program, `TokenLengthDist`, that receives an indefinite number of tokens from the user, and then computes the distribution of the lengths of the tokens. Treat any token having length greater than 15 as having length exactly 16. The user specifies the end of input by entering CTRL-D. To count, the program uses an array of length 17 and ignores slot 0.

   The program may work as follows:

```
1  Below enter any number of tokens. At the end enter CTRL-D.
2  I am writing a program by the name of TokenLengthDist. This is an
       interesting problem to solve.
3   1:2
4   2:6
5   3:1
6   4:2
7   5:0
8   6:1
9   7:3
10  8:0
11  9:0
12 10:0
13 11:1
14 12:0
15 13:0
16 14:0
17 15:0
18 16:1
```

10. **The Birthday Paradox** The "Birthday Paradox" states: the chances are 50–50 for a group of random 23 people to contain a pair of people having the same birthdays. Write a program, `BirthdayParadox`, that tests this paradox as follows: The program creates a 365-element array of `boolean` values (initially the elements are all `false`). The program also initializes an `int` variable that plays the role of a counter to 0. For an indefinite number of times, the program generates a random integer between 0 and 364. After generating each random integer, if the element of the array at the randomly generated index is `false`, then the program changes the

element of the array to `true`, and increases the value of the counter by 1; otherwise, the program exits the loop. After exiting the loop, the program reports the final value of the counter.

The program may work as follows (two runs):

```
% java BirthdayParadox
Found a collision at round 27
% java BirthdayParadox
Found a collision at round 48
```

11. **More on the Birthday Paradox** As a continuation of the previous problem, write a program `BirthdayParadoxStats` that receives a positive integer `r` from the user, executes the experiment as described in the previous question `r` times, stores the `r` counter values thus generated in an array of `int` data having `r` elements, and then reports the contents of this `r`-element array. The output of the result must produce 15 elements per line, as follows:

```
Enter the number of rounds: 200
 33  21  33  26  36  51  47  44  34  29   4  21  23  19  10
 21  33  13  48  59  37  12  11  31  30  54   8  28  19  29
 40   4   9  26  33  19  10  30  16  60   5  34  16  31  25
 38  30  19  44  31  48  43  30  20  25   4  16  50  15  32
 13  19  25  30  48  35  26  38  20  33   6  20  26  33  54
 25   6  37  27  35  11  10  28  29   6  21   7   7  30  43
 20  22  14  30  14  16  23  37  25  47  38  15  25  27   9
 19  15  27  31  29  25  10  33  15  13  15  30  44  34  39
 15  50  10  22   9  14  28  38  31  37   8  23  21  31  34
 30  30  28  18  17  36  12  14  32  22  40  18  44  35  10
 31  81  11  20  21  21  33  12  48  21  12  23  26  30  15
  7  20  41  14  12  27  27  10  36   6  32  38  35  34  22
 47  18   3  14   6  33  20  34  24  20  51  11  10  20  30
 24  21  21  21  31
```

12. **Random permutation of N things** Write a program, `RandomNThings`, that receives a positive integer `num` from the user, and then generates a random permutation of indexes `0, ..., N - 1`. The following algorithm can be used to accomplish this task:
    - Instantiate an array, `perm`, having length `num`.
    - For each index `i` between 0 and `N - 1`, store `i` in `perm[ i ]`.
    - Use a for-loop that iterates over the sequence `0, ..., num - 2` with a variable, `j`. At each round, generate a random integer between `j` and `num - 1`, store the integer in a variable, `k`, and then exchange the values between `perm[ j ]` and `perm[ k ]`.
13. **Generating a random rectangle in a range** Write a program, `RandomRectangle`, that receives a positive integer from the user, and then prints a rectangle. Let `size` be the variable in which the integer that the user enters is recorded. The coordinates of the rectangle should be integers between 0 and `size` (that is, the value `size` is included in the range). Each side of the rectangle should be either parallel to the horizontal axis or parallel to the vertical axis.

    To represent the rectangle generated, use the coordinates of the lower left corner and the upper right corner. Since there are four values total in the representation, use a four-element array of `int` values for the representation, where the four elements in the array are the x-coordinate of the lower left corner, the y-coordinate of the lower left corner, the x-coordinate of the upper right corner, and the y-coordinate of the upper right corner in this order.

The following algorithm can be used for the task.

- Select four random integers between 0 and `size` and store them in a four-element array of `int` values, `coord`.
- If either `coord[ 0 ] == coord[ 2 ]` or `coord[ 1 ] == coord[ 3 ]`, go back to the first step.
- If either `coord[ 0 ] > coord[ 2 ]`, exchange the values between them.
- If either `coord[ 1 ] > coord[ 3 ]`, exchange the values between them.