Multidimensional Arrays 14

14.1 Rectangular Arrays

14.1.1 Defining Multi-Dimensional Arrays

Arrays may have more than one dimension. We call arrays having more than one dimension **multi-dimensional arrays**. For an integer $N \geq 1$, an N-dimensional array as a type is declared with N pairs of brackets `[]`. In the following code, `mDouble` is declared as a two-dimensional array of `double` and `myFlags` is declared as a three-dimensional array of `boolean`.

```
double[][] myDouble;
boolean[][][] myFlags;
```

In an instantiation of a multi-dimensional array, the length must be specified for at least one dimension, but not necessarily for all of them. In the following code, the first line instantiates a two-dimensional array whose first dimension has length 11 and whose second dimension has length 35, and the second line instantiates a three-dimensional array whose first dimension has length 3. In the second array, the three elements `myFlags[ 0 ]`, `myFlags[ 1 ]`, and `myFlags[ 2 ]` are expected to be two-dimensional arrays, but they are presently `null` and so their shapes are unknown yet.

```
myDouble = new double[ 11 ][ 35 ];
myFlags = new boolean[ 3 ][][];
```

In an instantiation of a multi-dimensional array, if one dimension is without length specification, so must be its subsequent dimensions. Therefore,

```
myDouble = new double[][ 7 ];
myFlags = new boolean[ 5 ][][ 4 ];
```

are both syntactically incorrect.

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_14

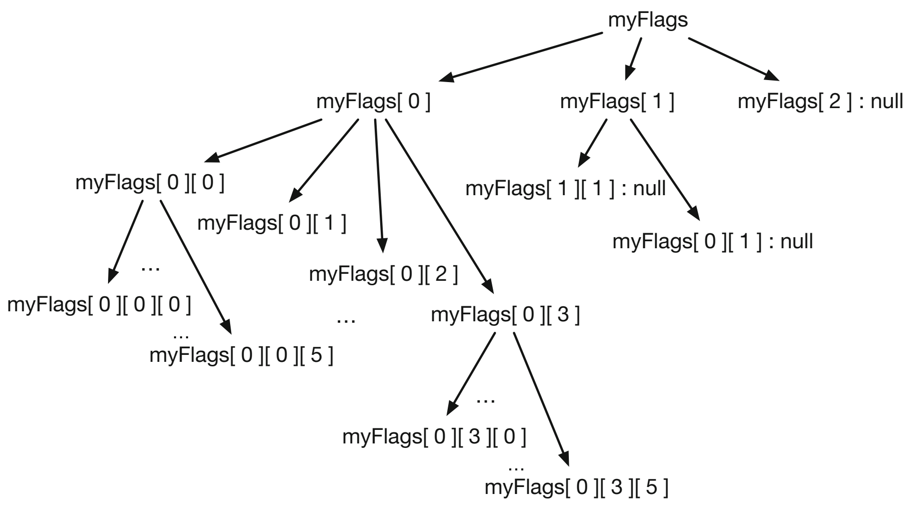

Fig. 14.1 The structure of a multi-dimensional array. The word "nulll" indicates subarrays that are `null`

After Line 4 of the above code example, possibly different two-dimensional arrays can be assigned to the three elements of `myFlags`. For example,

```
myFlags[ 0 ] = new boolean[ 4 ][ 6 ];
myFlags[ 1 ] = new boolean[ 2 ][];
```

assigns a 4-by-6 array to `myFlag[ 0 ]` and a two-dimensional array with first dimension having length 2 to `myFlag[ 1 ]`, but keeps `myFlags[ 2 ]` as `null`. A sub-array element of a multi-dimensional array can be accessed by specifying indexes to consecutive dimensions, where the values of the indexes start from 0 in every dimension that is already defined. Each sub-array element of a multi-dimensional array that is not equal to `null` can be inquired for the length of its first dimension using `.length`. In other words, after Line 6, `myFlags.length`, `myFlags[ 0 ].length`, and `myFlags[ 1 ].length` produce the values 3, 4, and 7 respectively, and for all `i` between 0 and 3, `myFlags[ 0 ][ i ].length` produces the value 6. For other index specification, such as `myFlags[ 1 ][ 0 ]`, whose subarray is `null` leads to `NullPointerException`. Figure 14.1 shows the structure of `myFlags`.

For a multi-dimensional array with length specifications for all of its possible dimensions, if its length specifications are uniform (for example, 3-by-5-by-6), we call it a **regularly shaped array**; otherwise, we call it an **irregularly shaped array** or a **jagged array**. We call a two-dimensional regularly shaped array a **rectangular array**, and a rectangular array whose dimensions have the same lengths a **square array**.

14.1.2 Summing the Elements in Subsequences

Consider computing the summation of elements for all subsequences of a number sequence $a_0, \ldots, a_{N-1}$. The task is to obtain

$$s(i, j) = \sum_{k=i}^{j} a_k = a_i + \ldots + a_j$$

for all i and j such that $0 \leq i \leq j \leq N - 1$. For all $0 \leq j < i \leq N - 1$, $s(i, j)$ is defined to be 0. Including the cases where the value is trivially 0, there are N possibilities for both i and j, so we can view the task as calculating the elements of a regularly-shaped two-dimensional array, both of whose dimensions have length N. Suppose the input sequence is represented by an array of `int`, `a`, whose length is `n`. The goal is to produce a two-dimensional array of `int`, `sums[ n ][ n ]`. The declaration and the instantiation of the two-dimensional array can be as follows:

```java
int[][] sums;
sums = new int[ a.length ][ a.length ];
```

We note the following three properties:

- For all `i` and `j` such that `i > j`, `sum[ i ][ j ]` must become 0. Since the default value of `int` is 0, this property is already satisfied.
- For all `i`, `sums[ i ][ i ]` must become `a[ i ]`.
- For all `i` and `j` such that `i < j`, `sum` must satisfy the condition `sum[ i ][ j ] == a[ j ] + sums[ i ][ j - 1 ]`.

These properties suggest the following strategy for computing the table. For each row `i`, store `a[ i ]` in `sums[ i ][ i ]`, and then using a for-loop that iterates over the sequence `i + 1, ..., a.length - 1` with another `int` variable named `j`, store `a[ j ] + sums[ i ][ j - 1 ]` in `sums[ i ][ j ]`.

The following source code is based upon these observations. The method `readData` is for obtaining an input number sequence from the user. The method has the return type of `int[]` (Line 9). The user first specifies the length of the sequence. This value is stored in an `int` variable named `len` (Line 11). After instantiating an array `numbers` with `new int[ len ]` (Line 12), the program receives the elements of the array from the user (Lines 13–17), and then returns the array (Line 18).

```java
 1 import java.io.*;
 2 import java.util.*;
 3 // computer all partials sums of an array
 4 public class PartialSumsAll
 5 {
 6   // read an array from a given scanner
 7   public static int[] readData()
 8   {
 9     Scanner keyboard = new Scanner( System.in );
10     System.out.print( "Enter the length: " );
11     int len = keyboard.nextInt();
12     int[] numbers = new int[ len ];
13     for ( int pos = 0; pos < len; pos ++ )
14     {
15       System.out.printf( "Enter element at %d: ", pos );
16       numbers[ pos ] = keyboard.nextInt();
17     }
18     return numbers;
19   }
20
```

Listing 14.1 A program that computing partial sums using a two-dimensional array (part 1). The method `readData`

The method `partialsAll` receives a one-dimensional `int` array, `oneD`, as its formal parameter, and returns a two-dimensional array whose elements are the values of the summations. The program instantiates a two-dimensional array as `oneD.length` by `oneD.length` (Line 24). The method then executes a double for-loop to fill this array. The external for-loop generates the row index with a variable named `rowPos` (Line 25). For each value of `rowPos`, the method stores `oneD[ rowPos ]` in `twoD[ rowPos ][ rowPos ]` (Line 27), and then using an internal for-loop, generates the column index values `rowPos + 1, ..., oneD.length - 1` with a variable named `colPos` (Line 28), and stores `twoD[ rowPos ][ colPos - 1 ] +oneD [ colPos ]` in `twoD[ rowPos ][ colPos ]` (Lines 30 and 31). After the double for-loop, the method returns `twoD` (Line 34).

```
21    // compute the partial sums of an input array
22    public static int[][] partialsAll( int[] oneD )
23    {
24      int[][] twoD = new int[ oneD.length ][ oneD.length ];
25      for ( int rowPos = 0; rowPos < oneD.length; rowPos ++ )
26      {
27        twoD[ rowPos ][ rowPos ] = oneD[ rowPos ];
28        for ( int colPos = rowPos + 1; colPos < oneD.length; colPos ++ )
29        {
30          twoD[ rowPos ][ colPos ]
31              = twoD[ rowPos ][ colPos - 1 ] + oneD[ colPos ];
32        }
33      }
34      return twoD;
35    }
36
```

Listing 14.2 A program that computing partial sums using a two-dimensional array (part 2). The method `partialsAll`

The method `print` is for printing the data of a two-dimensional array (Line 38). The program uses a double for-loop to generate row-column index pairs with variables `rowPos` and `colPos` (Lines 40 and 42), and then prints `twoD[ rowPos ][ colPos ]` using the format `%5d` (Line 44). At the end of each row, the method prints the newline (Line 46).

```
37    //-- print
38    public static void print( int[][] twoD )
39    {
40      for ( int rowPos = 0; rowPos < twoD.length; rowPos ++ )
41      {
42        for ( int colPos = 0; colPos < twoD.length; colPos ++ )
43        {
44          System.out.printf( "%5d", twoD[ rowPos ][ colPos ] );
45        }
46        System.out.println();
47      }
48    }
49
```

Listing 14.3 A program that computing partial sums using a two-dimensional array (part 3). The method `print`

The method `main` obtains a one-dimensional array from the user by calling `readData` (Line 53), computes the partial sums by calling `partialsAll` (Line 54), and then prints the contents of the partial sums by calling `print` (Line 55).

```
50    //-- main
51    public static void main( String[] args )
52    {
53      int[] a = readData();
54      int[][] sums = partialsAll( a );
55      print( sums );
56      // print( partialsAll( readData() ) );
57    }
58  }
```

Listing 14.4 A program that computing partial sums using a two-dimensional array (part 4). The main part

Alternatively, the three method calls appearing in the method `main` can be combined into one:

```
    print( partialsAll( readData() ) );
```

Here is an execution example of the program:

```
Enter the length: 5
Enter element at 0: 10
Enter element at 1: 13
Enter element at 2: 17
Enter element at 3: -30
Enter element at 4: -5
   10   23   40   10    5
    0   13   30    0   -5
    0    0   17  -13  -18
    0    0    0  -30  -35
    0    0    0    0   -5
```

14.2 Jagged Arrays

Consider receiving the names and scores for a number of people, and then computing the average score for each person, where the numbers of scores available can be different from person to person. We can use a two-dimensional jagged array for recording the scores, an array of `String` data to record names, and an array of `double` data to record the average scores.

The first part deals with the instantiation of the arrays. The user enters the number of people whose records are available. The program stores this number in a variable named `nPeople` (Lines 8 and 9), and then instantiates the three arrays (Lines 11–13).

```
 1  import java.util.*;
 2  public class Jagged
 3  {
 4    public static void main( String[] args )
 5    {
 6      Scanner keyboard = new Scanner( System.in );
 7
 8      System.out.print( "Enter # of people: " );
 9      int nPeople = keyboard.nextInt();
10
11      double[][] data = new double[ nPeople ][];
12      String[] names = new String[ nPeople ];
13      double[] average = new double[ nPeople ];
14
```

Listing 14.5 A program that receives the names and scores for a number of students and reports the averages of the scores (part 1). The part responsible for receiving the number of people and creating array variables

The next is the part for receiving the names and scores. The program uses a for-loop that iterates over the sequence `0, ..., nPeople - 1` with a variable named `index`. At each round, the program receives the name of the person (Lines 17 and 18), receives the number of scores for that person, stores the number in a variable named `size` (Lines 19 and 20), instantiates `data[ index ]` with `new double[ size ]` (Line 21), and then uses an interior for-loop to generate column indexes `0, ..., size - 1` with a variable named `col` (Line 23) to receive the elements `data[ index ][ 0 ], ..., data[ index ][ size - 1 ]` (Line 25).

```
15      for ( int index = 0; index < nPeople; index ++ )
16      {
17        System.out.printf( "Enter name for %d: ", index );
18        names[ index ] = keyboard.next();
19        System.out.printf( "Enter #entries %d: ", index );
20        int size = keyboard.nextInt();
21        data[ index ] = new double[ size ];
22        System.out.printf( "Enter %d data: ", size, index );
23        for ( int col = 0; col < size; col ++ )
24        {
25          data[ index ][ col ] = keyboard.nextDouble();
26        }
27      }
28
```

Listing 14.6 A program that receives the names and scores for a number of students and reports the averages of the scores (part 2). The part responsible for receiving the names and the scores

The calculation of the averages requires a double for-loop. The external for-loop iterates over the sequence `0, ..., nPeople - 1` with a variable named `index` (Line 29). The internal for-loop iterates over the sequence `0, ..., data[ index ].length - 1` with a variable named `col` (Line 31). The program stores the total of the scores appearing in that row in `average[ index ]` (Line 33). Then, with the division by `data[ index ].length`, the program scales the total stored in `average[ index ]` to the average.

```
29      for ( int index = 0; index < nPeople; index ++ )
30      {
31        for ( int col = 0; col < data[ index ].length; col ++ )
32        {
33          average[ index ] += data[ index ][ col ];
34        }
35        average[ index ] /= data[ index ].length;
36      }
37
```

Listing 14.7 A program that receives the names and scores for a number of students and reports the averages of the scores (part 3). The part responsible for computing the averages

The last part of the code handles the output. Again, the program uses a double for-loop (Lines 38 and 42) for accessing the individual scores. The program also prints the name (Line 40) and the average (Line 47) for each person.

```
38      for ( int index = 0; index < nPeople; index ++ )
39      {
40        System.out.printf( "Name at %d is %s%n", index, names[ index ] );
41        System.out.print( "The data are: " );
42        for ( int col = 0; col < data[ index ].length; col ++ )
43        {
44          System.out.printf( " %.3f", data[ index ][ col ] );
45        }
46        System.out.println();
47        System.out.printf( "Average is %.3f%n", average[ index ] );
48      }
49    }
50  }
```

Listing 14.8 A program that receives the names and scores for a number of students and reports the averages of the scores (part 4). The part responsible for reporting the results

Here is an execution example of the code.

```
Enter # of people: 4
Enter name for 0: Amelia
Enter #entries 0: 5
Enter 5 data: 90 91 92 93 84
Enter name for 1: Brittany
Enter #entries 1: 4
Enter 4 data: 89 91 89 94
Enter name for 2: Carolyn
Enter #entries 2: 3
Enter 3 data: 75 76 77
Enter name for 3: Diane
Enter #entries 3: 4
Enter 4 data: 100 89 99 97
Name at 0 is Amelia
The data are:  90.000 91.000 92.000 93.000 84.000
Average is 90.000
Name at 1 is Brittany
The data are:  89.000 91.000 89.000 94.000
Average is 90.750
Name at 2 is Carolyn
The data are:  75.000 76.000 77.000
Average is 76.000
Name at 3 is Diane
The data are:  100.000 89.000 99.000 97.000
Average is 96.250
```

Summary

- By attaching multiple bracket pairs, `[]`, to a data type, multi-dimensional arrays can be declared.
- An instantiation of a multi-dimensional array requires the length specification for the first dimension.
- In an instantiation of a multi-dimensional array, if the length specification of one dimension is skipped, then the length specification must be skipped for all the dimensions that follow.

Exercises

1. **Creating and printing a jagged array** Write a program, `CreatePrintJagged`, that instantiates a two-dimensional jagged array of `double` values that has ten rows. The lengths of the ten rows should be `0, ..., 9` in this order. For each individual element of the array, the program generates a random real number between 100 and 200. After generating the array, the program prints the elements of the array, one row per line. For printing the numbers, the program must use `%9.4f`. In this manner, without additional spacing, one whitespace appears before each number.
2. **Converting a square array to a one-dimensional array, part1** Write a method, `rectangularArrayToLinear`, that converts a two-dimensional rectangular array of `int` values to a one-dimensional array of `int` values by concatenating the elements of the rows of the array.
3. **Converting a square array to a one-dimensional array, part2** Write a method, `rectangularArrayToLearnAlt`, that converts a two-dimensional rectangular array of

int values to a one-dimensional array of int values by concatenating the elements of the columns of the array.

4. **Converting a one-dimensional array to a square array, part 1** Write a method, breakIntoSquareArray, that converts a one-dimensional array of int values, to a two-dimensional square array of int, where the length of the one-dimensional array may not be a perfect square. The return type of the method is int[][]. Find the largest perfect square that is less than or equal to the length of the one-dimensional array. Use the square root of the largest square as the length of each dimension of the square array. Place the elements of the one-dimensional array should be placed in the two-dimensional array in the row major order. If the length of the one-dimensional array is not a perfect square, there will be elements that are not included in the square array.
5. **Converting a one-dimensional array to a square array, part 2** Write a method, breakIntoSquareArrayAlt, that converts a one-dimensional array of int to a two-dimensional square array of int, where the length of the one-dimensional array may not be a perfect square. The return type of the method is int[][]. Find the largest perfect square that is less than or equal to the length of the one-dimensional array. Use the square root of the largest square as the length of each dimension of the square array. Place the elements of the one-dimensional array should be placed in the two-dimensional array in the column major order.
6. **Square array** Write a method, isArraySquare, that receives a two-dimensional array of double values, and then returns a boolean value representing whether or not the array is a square array. The method must return true if the array has 0 rows.
7. **Rectangular array** Write a method, isArrayRectangle, that receives a two-dimensional array of double values, and then returns a boolean value representing whether or not the array is a rectangular array. The method must return true if the array has 0 rows.
8. **Diagonal array** A square matrix is said to be a diagonal matrix if its all 0 except where the row index is equal to the column index. Write a method, isDiagonal, that checks whether or not a two-dimensional array is a diagonal array. The formal parameter of a method is a two-dimensional array of double values. The method must return a boolean value representing whether or not the array is a diagonal array.
9. **Upper-triangular array** A two-dimensional array is an upper-triangular array if it is a square array and is all 0 except where the column index is greater than or equal to the row index. Write a method, isUpperTriangular, that checks if a two-dimensional array is a upper-triangular array. The formal parameter of a method is a two-dimensional array of double values. The method must return a boolean value representing whether or not the array is an upper-triangular array.

Programming Projects

10. **Sudoku checking** Sudoku is a number puzzle in which the goal is to complete a 9-by-9 table that is partially filled. When completed, the table must satisfy the following conditions:
    - In each row, 1 through 9 must appear exactly once each.
    - In each column, 1 through 9 must appear exactly once each.
    - The 9-by-9 table can be naturally divided into nine 3-by-3 subtables. The third requirement is that in each 3-by-3 subtable, 1 through 9 must appear exactly once each.

    Write a method, sudokuSolutionCheck, that receives a 9-by-9 two-dimensional array of int values as its formal parameter, and then returns whether or not the array represents a Sudoku solution. The method may halt with ArrayIndexOutOfBoundsExceptions if the row or column dimension of the array is smaller than 9 or the array contains an element smaller than 1 or greater than 9.

11. **Sudoku checking with holes** Continuing on the previous problem, suppose that a partial solution to a Sudoku puzzle is represented by a 9-by-9 array of `int` values, where the elements are from 0 through 9 with 0 means that the value of the square is yet to be determined. Write a method, `sudokuPartialSolutionCheck`, that receives a 9-by-9 array of `int` values as its formal parameter, and then returns whether or not the partial solution represented by the array contains no violations of the three rules given in the statement of the previous question.
12. **Bingo card generation** Bingo is a game played by any number of players and one master. Each player has a card on which a 5-by-5 table of numbers is printed, where the numbers are chosen from 1, ..., 99 with no duplication and the center of the table has no number written. No two players have the same tables. To play the game, the master generates a random permutation of the number sequence 1, ..., 99, and then announces the numbers as they appear in the permuted sequence. Each time the master announces a number, each player checks if the number that just has been announced appears on her card. If so, she crosses out the number on her card. The center is thought of as being crossed out. If one row, one column, or one diagonal has been completely crossed out on her card, a player receives a prize and leaves the game.

    Suppose we use a 5×5 array of `int` values to encode a Bingo card, with -1 representing any number that has already been crossed out. Write a program, `BingoCardGenerate`, that generates a random Bingo card where the center is the only place that the number has been crossed out. A simple solution will be to conduct a random exchange on an array { `1, ..., 99` } 25 times, fill the array with the first 25 numbers on the permuted sequence, and then change the center to -1.

    Here is a possible output of the program:

```
23  9 15 77 67
64 19 86  7 45
24 68 -1 74 22
78 34 63 59 40
39 25  8  1  2
```

13. **Bingo checking** Continuing on the previous question, suppose we use a 5×5 array of `boolean` values to encode the places of a card that have been crossed out. In the encoding, `true` means "crossed out" and `false` means "has not been crossed out". Write a method, `bingoCheck`, that receives a 5-by-5 `boolean` table, and then determines whether or not the player can claim the completion to receive a prize.