16 Multidimensional Arrays

1 Multidimensional arrays

The arrays we have seen so far are “1-dimensional”, by which is meant that a single index suffices to identify any component of the array. A 2-dimensional array can be visualised as a grid; for example, a 2-dimensional integer array b with 4 columns and 3 rows can be pictured as

	0	1	2	3
0	2	7	1	9
1	3	6	4	**5**
2	6	3	2	7

b[1][3]

Each row and column is indexed from 0, and each component cell is identified by supplying two indices as indicated. We say that an array with m rows and n columns is an “m×n” array; for example, the array above is 3×4. A 2-dimensional array is also called a matrix. Arrays may have more than two dimensions.

In Java, a 3×4 integer array b is created by the following declaration:

```
int[][] b = new int[3][4];
```

Initialisation of the array can be included in the definition, as typified by

```
int[][] b = {{3,4,6,2}, {6,1,2,1}, {9,2,4,3}};
```

2 Example: magic squares

We write a program to generate a magic square, i.e. a square grid of distinct numbers whose rows, columns, and diagonals all have the same sum. The following is a magic square of order 3, i.e. a 3×3 magic square using the numbers 1 to 9, incl. (verify that each row, column, and diagonal adds up to 15):

8	1	6
3	5	7
4	9	2

When the numbers in the magic square are 1, 2, 3, ... we call it a *normal* magic square. We confine our attention to normal magic squares. The following is an algorithm to generate a (normal) magic square of order n, provided n is odd. Deposit 1, 2, 3, ..., n^2 in the cells of an n×n grid proceeding as follows. The first cell to receive a value is the middle of the top row. The successor cell in each case will be the cell immediately to the north-east (see how 6 follows 5 above). However, if the north-east cell does not exist, we wrap around the borders (see how 2 follows 1, and how 3 follows 2). After each group of n numbers have been deposited, the successor cell is immediately to the south rather than the north-east (see how 4 is to the south of 3 above). Test your understanding of this on the magic square above. For a further check, generate a magic square of order 5. You should get the following:

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

The program should print the number in each cell “right-justified”, i.e. each column of numbers is aligned on the right-hand side. The magic squares program is given below. The command

```
java MagicSquare 5
```

will generate a magic square of order 5 (note that the size of the square is supplied as a command line argument).

```
class MagicSquare {
   public static void main(String[] args) {
    // Generate a magic square of order args[0], which
    // must be an odd positive integer
        int n = Integer.parseInt(args[0]);
        int[][] ms = new int[n][n]; // for the magic square
        int num = 1; // next number to be deposited
        int j = 0; int k = n/2; // next cell to be filled is ms[j][k]
        while (num<=n*n) {
            ms[j][k] = num; // fill in cell
            if (num%n==0) { // after each n steps go south
                j++;
            }
            else { // otherwise go north-east, with wrap around if necc.
                j--; if (j<0) j = n-1;
                k++; if (k==n) k = 0;
            }
            num++;
        }
        // print magic square
        for (j=0; j<n; j++) {
            for (k=0; k<n; k++) {
                // print  ms[j,k] right-justified in field of width 5
                System.out.printf("%5d", ms[j][k]);
            }
            System.out.println();
        }
    }
}
```

3 Arrays of arrays

We usually envisage a 2-dimensional array as a grid, as we have done above, and that picture will is sufficiently accurate to program with. The true picture is slightly more complex, however. A 2-dimensional array is actually implemented as an array of arrays: each row is a 1-dimsensional array. For example, given

```
int[][] b = new int[3][4]
```

b[1] is a 1-dimensional array of integers with four elements b[1][0], b[1][1], b[1][2], and b[1][3]. It is not a requirement that each row array have the same number of elements, e.g.

```
    int[][] b = {{3,4,6,2}, {6,1}, {9,4,3}};
```

It is possible to supply b[1], say, whenever a 1-dimensional array is expected, as in:

```
    int[][] b = {{3,4,6,2}, {6,1}, {9,4,3}};
    int[] c = b[1];
```

We don’t often use these additional possibilities.