Chapter 15

Arrays of Arrays

15.1 Initializing Two-Dimensional Arrays

You already know that you can initialize a one-dimensional array by putting the values in braces:

```
int[] ages = {35, 27, 45, 58};
```

Because two-dimensional arrays are stored in Java as an array of arrays, as shown in Figure 15.1, you can initialize them in a simliar manner.

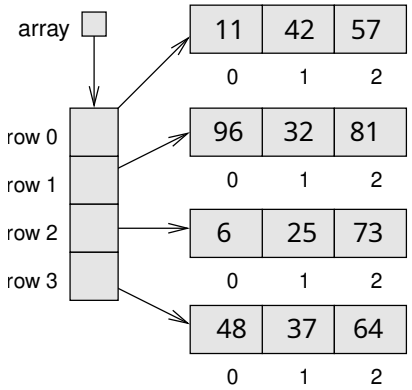

Figure 15.1: Storing rows and columns with a 2D array.

```
int[ ][ ] array = {
    {11, 42, 57},
    {96, 32, 81},
    { 6, 25, 73},
    {48, 37, 64}
};
```

However, there are times that we need to be able to read two-dimensional arrays from a user. It would be useful to have a small library of code that would allow us to read and print two-dimensional arrays so that we would not have to copy and paste the code into every program that needs it. That is the subject of the next section and the second exercise.

15.2 Programs in Multiple Files

Up to this point, all of our Java programs have been written in one file. We have used `import` to bring in classes and methods from other files. Now it's time to learn how to do this ourselves. There are many ways to structure a program so that it can be divided into separate files that are brought together at run time; the method we're using here is the simplest.

As an example, let's say we would like to have some methods that are generally useful for doing statistics with arrays of `double` values. We'll put them into a `public` class of their own. The following code gives the outline; you can download the entire file from the code repository at https://github.com/jdeisenberg/ThinkJava2ExCode

```
public class ArrayStats {

    public static double mean(double[ ] data) {
        // calculation code goes here
        return result;
    }

    public static double stdv(double[ ] data) {
        // code goes here
        return result;
    }
}
```

You can now use these methods from another program, say, `TestStats.java` by putting the `ArrayStats.java` file in the same directory as the `TestStats.java` file. If you do this, you don't need to do an `import`:

```
public class TestStats {

    public static void main(String[ ] args) {
        double [ ] values = {10.0, 47.0, 6.6, 505.0217, 11.0};
        double avg = ArrayStats.mean(values);
        double stdDev = ArrayStats.stdv(values);
        System.out.printf("Average: %.3f\n", avg);
        System.out.printf("Standard Deviation: %.3f\n", stdDev);
    }
}
```

More complex ways of dividing a Java program into separate files involves the concepts of `package` and `module`. You can see documentation about this at https://docs.oracle.com/javase/specs/jls/se9/html/jls-7.html.

15.3 Exercises

Exercise 15.1 Create a two-dimensional array of `int` that represents the following table of sales (in thousands of units), measured quarterly for three years. The first row and the leftmost column won't be part of your table; they are here to label the items properly so you can understand how the table is structured.

Year	Q1	Q2	Q3	Q4
1	27	24	29	28
2	36	38	34	33
3	34	39	37	36

Then, have your program calculate and print:

- The total sales for each of the three years
- The total sales for each quarter (across the three years)

- The grand total of sales

Exercise 15.2 In this exercise, you'll write methods for doing input and output of a two-dimensional array. Put these methods into a file named `IO2D.java` (standing for Input/Output 2D). You will use these methods in the subsequent exercises.

Input: Write a `public static double[ ][ ]` method named `getDoubleArray2D` with four parameters:

- A `Scanner` object
- The prompt for the description of the array to enter (`String`)
- The number of rows (`int`)
- The number of columns (`int`)

The method returns a two-dimensional array of `double`. For example, you might call it as:

```
double [ ][ ] temps = getDoubleArray2D(input,
  "Enter temperature data one row at a time", 7, 3);
```

Output: Write a `public static void` method named `printDoubleArray2D`, with two parameters:

- A formatting `String` like you would use in `System.out.printf` or `String.format`.
- The array to be printed

This method will print out the arrays, using the given format for each element in the array. You might call it like this:

```
printDoubleArray2D("%.1f", temps);
```

Then, in a class named `TestInputOutput`, write a `main` method that creates a 3 row, 4 column array of `double` and calls `getDoubleArray2D` and `printDoubleArray2D` to read an array and print it out.

Here is sample output from this code:

```
double[ ][ ] values = IO2D.getDoubleArray2D(input,
    "Enter three rows of data, one row at a time.", 3, 4);
System.out.println();
System.out.println("Here is your data:");
IO2D.printDoubleArray2D("%7.1f", values);
```

```
Enter three rows of data, one row at a time.
Enter 4 items for row 1: 3.4 7.8 2.93 -12.1
Enter 4 items for row 2: 14.4 8.75 3.45 18.2225
Enter 4 items for row 3: 31.2 -56.4 19.21 47.66

Here is your data:
    3.4     7.8     2.9   -12.1
   14.4     8.8     3.5    18.2
   31.2   -56.4    19.2    47.7
```

Exercise 15.3 In this exercise and the following exercises, you will develop methods that do mathematics with two-dimensional arrays, also called *matrices*. Create a file named `MatrixMath.java`, which will have a `public class MatrixMath`.

First, write a method that adds two matrices. You do this by adding the corresponding entries of the two matrices:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{pmatrix} \rightarrow \begin{pmatrix} 11 & 13 & 15 \\ 17 & 19 & 21 \\ 23 & 25 & 27 \end{pmatrix}$$

Here is the signature of the method:

```
public static double[][] add(double[][] arr1, double[][] arr2)
```

The two arrays must have the same number of rows and columns. Write a method named `isAddCompatible` that will do that test for the users of your code. The method returns `true` if the arrays can be added, `false` otherwise. Its signature is:

```
public static boolean isAddCompatible(double[][] arr1, double[][] arr2)
```

To test these methods, write a separate file with a `main` method that:

- Asks the user for the dimensions of one array and reads it in.
- Asks the user for the dimensions of another array and reads it in.
- If the arrays are compatible, add them and print the resulting array. If not, print an appropriate error message.

Exercise 15.4 Add a method to the `MatrixMath` class:

```
public static double[][] scale(double[][] arr, double factor)
```

This method returns a new array the same size as the input array, with each element in the array multiplied by the given factor. Write a test program in a separate file that asks the user for the dimensions of an array, reads it in, asks for a scaling factor, reads that in, and scales the elements by the given factor and prints the result.

Exercise 15.5 Add a method to the `MatrixMath` class that returns the *transpose* of an array. When you transpose an array, the first row becomes the first column, the second row becomes the second column, etc. For example:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix}$$

The header of the method is:

```
public static double[][] transpose(double[][] arr)
```

Write a test program in a separate file that asks the user for the dimensions of an array, reads it in, transposes it, and prints the result.

Exercise 15.6 Adding matrices is fairly straightforward. Multiplying them is anything but! In this exercise, you will write a method to multiply two matrices. The header of the method is:

```
public static double[][] multiply(double[][] a, double [][] b)
```

To multiply a matrix a by matrix b, the number of columns in a must be the same as the number of rows in b, and the two matrices must have elements of the same or compatible types. Let c be the result of the multiplication. Assume that a has n columns. Each element c_{ij} is $a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + ... + a_{in} \times b_{nj}$. For example,

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{pmatrix} =$$

$$\begin{pmatrix} 1 \cdot 7 + 2 \cdot 8 + 3 \cdot 9 & 4 \cdot 7 + 5 \cdot 8 + 6 \cdot 9 \\ 1 \cdot 10 + 2 \cdot 11 + 3 \cdot 12 & 4 \cdot 10 + 5 \cdot 11 + 6 \cdot 12 \end{pmatrix} = \begin{pmatrix} 50 & 68 \\ 122 & 167 \end{pmatrix}$$

Implement a method with this header:

```
public static boolean isMultiplyCompatible(double [][] arr1,
    double [][] arr2)
```

This will return `true` if the number of columns in `arr1` is equal to the number of rows in `arr2`, `false` otherwise.

Your `main` method will prompt the user for the number of rows and columns for the first matrix, then its contents. It will then prompt the user for the number of rows and columns for the second matrix, then its contents. It will then determine if the matrices are compatible by calling `isMultiplyCompatible`. If the matrices are compatible, the program will call the `multiply` method and print the returned matrix. If not, it will print an appropriate error message.

Here is an example of a run of the program:

```
Matrix A
Enter number of rows: 2
Enter number of columns: 3
Enter contents by rows: 1 2 3 4 5 6
Matrix B
Enter number of rows: 3
Enter number of columns: 2
Enter contents by rows: 7 10 8 11 9 12

Matrix C is
50.0 68.0
122.0 167.0
```

Exercise 15.7 Write a program named *Dice.java* that will do the following:

1. Ask the user how many times they wish to roll a pair of dice, from 1-999. Validate the input—keep asking until the number is in range. Write a method named `getNRolls` to do this step. The method takes a `Scanner` object as its parameter and returns the number of rolls the user desires.

2. Create a 6×6 array named `rolls` to hold the result of the rolls.

3. Simulate randomly rolling a pair of dice as many times as the user requested. As you roll the dice, count how many times each combination of dice was rolled. For example, if the first die is a 3 and the second die is 5, you will add one to the entry at `rolls[2][4]`. (If you generate numbers in the range 1-6, you will have to subtract one at appropriate places. If you generate numbers in the range 0-5, you don't need to do any subtraction). Write a method named `rollDice` to do this step. It takes the number of rolls as its parameter and returns the two-dimensional array of `rolls`.

4. Print out the array, properly labeled, so that the numbers all line up. You can do this in `main` or write a method to do it.

5. Go through the array you created in step 3 and figure out how many times the dice totaled 2, 3, 4, ... 12. Write a method called `calcFrequencies` to do this. It will take the `rolls` array as its parameter and return a single-dimensional array of length 11, which your `main` method will store in an array named `totals`.

6. Print the frequency array from step 5, properly labeled. You may do this in `main` or write a method to do it.

Yes; I know this is "inefficient." It is possible to construct the `totals` array at the same time that you create the `rolls` array. You could do everything in `main` without any other methods. However, I want you to practice passing and returning arrays to and from methods, so I decided to write the exercise this way.

Sample output:

```
How many times do you want to roll the dice (1-999)? 200
Frequencies for individual dice
      1   2   3   4   5   6
    ------------------------
1|    3   7   2   3   8   4
2|    4   7   2   5  10   5
3|    9  10   7   4   6   5
4|    6   4   5   7   4   8
5|    3   3   3   8   6   6
6|    5   5   8   7   3   8

Frequencies of totals
 2:   3
 3:  11
 4:  18
 5:  21
 6:  27
 7:  31
 8:  26
 9:  25
10:  21
11:   9
12:   8
```