Chapter 11

Two-Dimensional Arrays

This chapter introduces Java arrays with two subscripts for managing data logically stored in a table-like format—in rows and columns. This structure proves useful for storing and managing data in many applications, such as electronic spreadsheets, games, topographical maps, and student record books.

11.1 2-D Arrays

Data that conveniently presents itself in tabular format can be represented using an array with two subscripts, known as a two-dimensional array. Two-dimensional arrays are constructed with two pairs of square brackets to indicate two subscripts representing the row and column of the element.

***General Form:* A two-dimensional array consruction (all elements set to default values)**

type **[ ] [ ]** *array-name* **= new** *type* **[***row-capacity***] [***column-capacity***] ;**
type **[ ] [ ]** *array-name* **= { {** element[0][0], element[0][1], element[0][2], … **} ,**
{ element[1][0], element[1][1], element[1][2], … **} ,**
{ element[2][0], element[2][1], element[2][2], … **} };**

- *type* may be one of the primitive types or the name of any Java class or interface
- *identifier* is the name of the two-dimensional array
- *rows* specifies the total number of rows
- *columns* specifies the total number of columns

Examples:

```
double[][] matrix = new double[4][8];

// Construct with integer expressions
int rows = 5;
int columns = 10;
String[][] name = new String[rows][columns];

// You can use athis shortcut that initializes all elements
int[][] t = { { 1, 2, 3 },   // First row of 3 integers
              { 4, 5, 6 },   // Row index 1 with 3 columns
              { 7, 8, 9 } }; // Row index 2 with 3 columns
```

Referencing Individual Items with Two Subscripts

A reference to an individual element of a two-dimensional array requires two subscripts. By convention, programmers use the first subscript for the rows, and the second for the columns. Each subscript must be bracketed individually.

General Form: **Accessing individual two-dimensional array elements**

two-dimensional-array-name **[***rows***] [***columns***]**

- *rows* is an integer value in the range of 0 through the number of rows - 1
- *columns* is an integer value in the range of 0 through the number of columns - 1

Examples:

```
String[][] name = new String[5][10];
name[0][0] = "Upper Left";
name[4][9] = "Lower Right";
assertEquals("Upper Left", name[0][0]);

// name.length is the number of rows,
// name[0].length is the number of columns
assertEquals("Lower Right", name[name.length-1][name[0].length-1]);
```

Nested Looping with Two-Dimensional Arrays

Nested looping is commonly used to process the elements of two-dimensional arrays. This initialization allocates enough memory to store 40 floating-point numbers—a two-dimensional array with five rows and eight columns. Java initializes all values to 0.0 when constructed.

```
int ROWS = 5;
int COLUMNS = 8;
double[][] table = new double[ROWS][COLUMNS];  // 40 elements set to 0.0
```

These nested `for` loops initialize all 40 elements to `-1.0`.

```
// Initialize all elements to -1.0
for (int row = 0; row < ROWS; row++) {
  for (int col = 0; col < COLUMNS; col++) {
    table[row][col] = -1.0;
  }
}
```

Self-Check

Use this construction of a 2-D array object to answer questions 1 through 8:

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

11-1 What is the value of `a[1][2]`?

11-2 Does Java check the range of the subscripts when referencing the elements of `a`?

11-3 How many `int`s are properly stored by `a`?

11-4 What is the row (first) subscript range for `a`?

11-5 What is the column (second) subscript range for `a`?

11-6 Write code to initialize all of the elements of `a` to 999.

11-7 Declare a two-dimensional array `sales` such that stores 120 doubles in 10 rows.

11-8 Declare a two-dimensional array named `sales2` such that 120 floating-point numbers can be stored in 10 columns.

A two-dimensional array manages tabular data that is typically processed by row, by column, or in totality. These forms of processing are examined in an example class that manages a grade book. The data could look like this with six quizzes for each of the nine students.

	Quiz #0	1	2	3	4	5
0	67.8	56.4	88.4	79.1	90.0	66.0
1	76.4	81.1	72.2	76.0	85.6	85.0
2	87.8	76.4	88.7	83.0	76.3	87.0
3	86.4	54.0	40.0	3.0	2.0	1.0
4	72.8	89.0	55.0	62.0	68.0	77.7
5	94.4	63.0	92.9	45.0	75.6	99.5
6	85.8	95.0	88.1	100.0	60.0	85.8
7	76.4	84.4	100.0	94.3	75.6	74.0
8	57.9	49.5	58.8	67.4	80.0	56.0

This data will be stored in a tabular form as a 2D array. The 2D array will be processed in three ways:

1. Find the average quiz score for any of the 9 students
2. Find the range of quiz scores for any of the 5 quizzes
3. Find the overall average of all quiz scores

Here are the methods that will be tested and implemented on the next few pages:

```
// Return the number of students in the data (#rows)
public int getNumberOfStudents()

// Return the number of quizzes in the data (#columns)
public int getNumberOfQuizzes()
```

```
// Return the average quiz score for any student
public double studentAverage(int row)

// Return the range of any quiz
public double quizRange(int column)

// Return the average of all quizzes
public double overallAverage()
```

Reading Input from a Text File

In programs that require little data, interactive input suffices. However, initialization of arrays quite often involves large amounts of data. The input would have to be typed in from the keyboard many times during implementation and testing. That much interactive input would be tedious and error-prone. So here we will be read the data from an external file instead.

The first line in a valid input file specifies the number of rows and columns of the input file. Each remaining line represents the quiz scores of one student.

```
 9   6
 67.8   56.4   88.4   79.1   90.0   66.0
 76.4   81.1   72.2   76.0   85.6   85.0
 87.8   76.4   88.7   83.0   76.3   87.0
 86.4   54.0   40.0    3.0    2.0    1.0
 72.8   89.0   55.0   62.0   68.0   77.7
 94.4   63.0   92.9   45.0   75.6   99.5
 85.8   95.0   88.1  100.0   60.0   85.8
 76.4   84.4  100.0   94.3   75.6   74.0
 57.9   49.5   58.8   67.4   80.0   56.0
```

The first two methods to test will be the two getters that determine the dimensions of the data. The actual file used in the test has 3 students and 4 quizzes. The name of the file will be passed to the `QuizData` constructor.

```
@Test
public void testGetters() {
  /* Process this small file that has 3 students and 4 quizzes.
   3 4
   0.0 10.0 20.0 30.0
   40.0 50.0 60.0 70.0
   80.0 90.0 95.5 50.5
  */
  QuizData quizzes = new QuizData("quiz3by4");
  assertEquals(3, quizzes.getNumberOfStudents());
  assertEquals(4, quizzes.getNumberOfQuizzes());
}
```

The name of the file will be passed to the `QuizData` constructor that then reads this text data using the familiar Scanner class. However, this time a new File object will be needed. And this requires some understanding of exception handling.

Exception Handling when a File is Not Found

When programs run, errors occur. Perhaps an arithmetic expression results in division by zero, or an array subscript is out of bounds, or there is an attempt to read a file from a disk using a specific file name that does not exist. Or perhaps, the expression in an array subscript is negative or 1 greater than the capacity of that array. Programmers have at least two options for dealing with these types of exception:

- Ignore the exception and let the program terminate
- Handle the exception

However, in order to read from an input file, you cannot ignore the exception. Java forces you to try to handle the exceptional event. Here is the code the tries to have a Scanner object read from an input file named quiz.data. Notice the argument is now a new File object.

```
Scanner inFile = new Scanner(new File("quiz.data));
```

This will not compile. Since the file "quiz.data" may not be found at runtime, the code may throw a `FileNotFoundException`. In this type of exception (called a checked exception), Java requires that you put the construction in a `try` block—the keyword `try` followed by the code wrapped in a block, `{ }`.

```
try {
    code that may throw an exception when an exception is thrown
}
catch (Exception anException) {
    code that executes only if an exception is thrown from code in the above try block.
}
```

Every `try` block must be followed by a at least one `catch` block—the keyword `catch` followed by the anticipated exception as a parameter and code wrapped in a block. The `catch` block contains the code that executes when the code in the `try` block causes an exception to be thrown (or called a method that throws an exception). So to get a Scanner object to try to read from an input file, you need this code.

```
Scanner inFile = null;
try {
  inFile = new Scanner(new File(fileName));
}
catch (FileNotFoundException fnfe) {
  System.out.println("The file '" + fileName + " was not found");
}
```

This will go into the `QuizData` constructor that reads the first two integers as the number of rows followed by the number of columns as integers. The file it reads from is passed as a string to the constructor. This allows the programmer to process data stored in a file (assuming the data is properly formatted and has the correct amount of input.

```
// A QuizData object will read data from an input file an allow access to
// any students quiz average, the range of any quiz, and the average quiz
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class QuizData {
  // Instance variables
  private double[][] quiz;
  private int numberOfStudents;
  private int numberOfQuizzes;

  public QuizData(String fileName) {
    Scanner inFile = null;
    try {
      inFile = new Scanner(new File(fileName));
    }
    catch (FileNotFoundException e) {
      System.out.println("The file '" + fileName + " was not found");
    }

    // More to come ...
}
```

Because the private instance variables members are known throughout the `QuizData` class, the two-dimensional array named `quiz` can, from this point forward, communicate its subscript ranges for both rows and columns at any time and in any method. These values are stored

```
// Get the dimensions of the array from the input file
numberOfStudents = inFile.nextInt();
numberOfQuizzes = inFile.nextInt();
```

The next step is to allocate memory for the two-dimensional array:

```
quiz = new double[numberOfStudents][numberOfQuizzes];
```

Now with a two-dimensional array precisely large enough to store `numberOfStudents` rows of data with `numberOfQuizzes` quiz scores in each row, the two-dimensional array gets initialized with the file data using nested `for` loops.

```
// Initialize a numberOfStudents-by-numberOfQuizzes array
for (int row = 0; row < getNumberOfStudents(); row++) {
  for (int col = 0; col < getNumberOfQuizzes(); col++) {
    quiz[row][col] = inFile.nextDouble();
  }
}
} // End QuizData(String) constructor
```

`QuizData` also has these getters now s the first test method has both assertions passing

```
public int getNumberOfStudents() {
  return numberOfStudents;
}

public int getNumberOfQuizzes() {
  return numberOfQuizzes;
}
```

However, more tests are required to verify the 2D array is being initialized properly. One way to do this is to have a `toString` method so the array can be printed.

Self-Check

11-9 Write method `toString` that will print the elements in any QuizData object to look like this:

```
0.0 10.0 20.0 30.0
40.0 50.0 60.0 70.0
80.0 90.0 95.5 50.5
```

Student Statistics: Row by Row Processing

To further verify the array was initialized, we can write a test to make sure all three students have the correct quiz average.

```
@Test
public void testStudentAverage() {
  /* Assume the text file "quiz3by4" has these four lines of input data:
     3 4
     0.0 10.0 20.0 30.0
     40.0 50.0 60.0 70.0
     80.0 90.0 95.5 50.5
  */
  QuizData quizzes = new QuizData("quiz3by4");
  assertEquals(15.0, quizzes.studentAverage(0), 0.1);
  assertEquals(220.0 / 4, quizzes.studentAverage(1), 0.1);
  assertEquals((80.0+90.0+95.5+50.5) / 4, quizzes.studentAverage(2), 0.1);
}
```

The average for one student is found by adding all of the elements of one row and dividing by the number of quizzes. The solution uses the same row as `col` changes from 0 through 3.

```
// Return the average quiz score for any student
public double studentAverage(int row) {
  double sum = 0.0;
  for (int col = 0; col < getNumberOfQuizzes(); col++) {
    sum = sum + quiz[row][col];
  }
  return sum / getNumberOfQuizzes();
}
```

Quiz Statistics: Column by Column Processing

To even further verify the array was initialized, we can write a test to ensure correct quiz ranges.

```
@Test
public void testQuizAverage() { // Assume the text file "quiz3by4" has these 4 lines
  // 3 4
  // 0.0 10.0 20.0 30.0
  // 40.0 50.0 60.0 70.0
  // 80.0 90.0 95.5 50.5
  QuizData quizzes = new QuizData("quiz3by4");
  assertEquals(80.0, quizzes.quizRange(0), 0.1);
  assertEquals(80.0, quizzes.quizRange(1), 0.1);
  assertEquals(75.5, quizzes.quizRange(2), 0.1);
  assertEquals(40.0, quizzes.quizRange(3), 0.1);
}
```

The range for each quiz is found by first initializing the min and the max by the quiz score in the given column. The loop uses the same column as `row` changes from 1 through 3 (already checked row 0). Inside the loop, the current value is compared to both the min and the max to ensure the max – min is the correct range.

```
// Find the range for any given quiz
public double quizRange(int column) {
  // Initialize min and max to the first quiz in the first row
  double min = quiz[0][column];
  double max = quiz[0][column];
  for (int row = 1; row < getNumberOfStudents(); row++) {
    double current = quiz[row][column];
    if (current < min)
      min = current;
    if (current > max)
      max = current;
  }
  return max - min;
}
```

Overall Quiz Average: Processing All Rows and Columns

The test for overall average shows that an expected value of 49.67.

```
@Test
public void testOverallAverage() {
  QuizData quizzes = new QuizData("quiz3by4");
  assertEquals(49.7, quizzes.overallAverage(), 0.1);
}
```

Finding the overall average is a simple matter of summing every single element in the two-dimensional array and dividing by the total number of quizzes.

```
public double overallAverage() {
  double sum = 0.0;
  for (int studentNum = 0; studentNum < getNumberOfStudents(); studentNum++) {
    for (int quizNum = 0; quizNum < getNumberOfQuizzes(); quizNum++) {
      sum += quiz[studentNum][quizNum];
    }
  }
  return sum / (getNumberOfQuizzes() * getNumberOfStudents());
}
```

Answers to Self-Checks

11-1 0.0

11-2 Yes

11-3 12

11-4 0 through 2 inclusive

11-5 0 through 3 inclusive

11-6
```
for (int row = 0; row < 3; row++) {
  for (int col = 0; col < 4; col++) {
    a [row][col] = 999;
  }
}
```

11-7
```
double[][]sales = new double[10][12];
```

11-8
```
double[][]sales2 = new double[12][10];
```

11-9
```
public String toString() {
  String result = "";
  for (int studentNum = 0; studentNum < getNumberOfStudents(); studentNum++){
    for (int quizNum = 0; quizNum < getNumberOfQuizzes(); quizNum++) {
      result += " " + quiz[studentNum][quizNum];
    }
    result += "\n";
  }
  return result;
}
```