STUDENT LESSON

Two-Dimensional Arrays

**INTRODUCTION:** Two-dimensional arrays allow the programmer to solve problems involving rows and columns. Many data processing problems involve rows and columns, such as an airplane seat reservation system or the mathematical modeling of bacterial growth. A classic problem involving two-dimensional arrays is the bacteria simulation program presented in Lab Assignment AB23.1, *Life.* After surveying the syntax and unique aspects of larger data structures, we will apply them to more challenging lab assignments.

The key topics for this lesson are:

A. Two-Dimensional Arrays

B. Passing Two-Dimensional Arrays to Methods

C. Two-Dimensional Array Algorithms

**VOCABULARY:** MATRIX ROW

COLUMN LENGTH

**DISCUSSION:** A. Two-Dimensional Arrays

1. Often the data a program uses comes from a two-dimensional situation. For example, maps are two-dimensional (or more), the layout of a printed page is two-dimensional, a computer-generated image (such as on your computer's screen) is two-dimensional, and so on.

For these situations, a Java programmer can use *a two-dimensional array*. This allows for the creation of table-like data structures with a row and column format. The first subscript is a row index in a table while the second subscript is a column index in a table. Here is example code, in Sample Code 23-1, including a diagram of the array table.

Sample Code 23-1

**int**[][] table = **new** **int**[3][4];

**int** row, col;

**for** (row = 0; row < 3; row++){

**for** (col = 0; col < 4; col++){

table[row][col] = row + col;

}

}

**0**

**1**

**2**

**3**

**1**

**2**

**3**

**4**

**2**

**3**

**4**

**5**

**0**

**1**

**2**

**3**

**0**

**1**

**2**

table

2. Two-dimensional arrays are objects. A variable such as table is a reference to a 2D array object. The declaration

**int**[][] table;

defines a table that holds a reference to a 2D array of integers. Without any further initialization, table holds the value **null**.

3. The declaration

**int**[][]table = **new** **int**[3][4];

defines a table that holds a reference to a 2D array of integers, creates an array object of 3 rows and 4 columns, and puts the reference in table. All the elements of the array are initialized to zero.

4. The declaration

**int**[][] table = { {0,0,0,0},

{0,0,0,0},

{0,0,0,0} };

does exactly the same thing as the previous declaration (and would not ordinarily be used.)

5. The declaration

**int**[][]table = { {0,1,2,3},

{1,2,3,4},

{2,3,4,5} };

creates an array of the same dimensions (same number of rows and columns) as the previous array and initializes the elements to the given values in each cell.

6. If no initializer is provided for an array, then when the array is created, it is automatically filled with the appropriate values: zero for numbers, **false** for **boolean**, and **null** for objects.

7. Just as with one-dimensional arrays, the row and column numbering of a 2-D array begins at subscript zero (0). The 3 rows of the table are numbered from 0...2. Likewise, the 4 columns of the table are numbered from 0...3.

8. The array table is filled with the sums of row and col, which is accomplished by Sample Code 23-2 (see below)*.* To access each location of the matrix, specify the row coordinate first, then the column:

table[row][col]

Each subscript must have its own square brackets.

9. The length of a 2D array is the number of *rows* it has. The row index will run from 0 to length -1. The number of rows in table is given by the value table.length.

Each row of a 2D array has its own length. To get the number of columns in table, use any of the following:

table[0].length

table[1].length

table[2].length.

There is actually no rule that says that all the rows of an array must have the same length, and some advanced applications of arrays use varying-sized rows. But if you use the **new** operator to create an array in the manner described above, you'll always get an array with equal-sized rows.

10. The routine that assigned values to the array used the specific numbers of rows and columns. That is fine for this particular program, but a better definition would work for an array of any two dimensions.

Sample Code 23-2

**int**[][] table = **new** **int**[3][4];

**int** row, col;

**for** (row = 0; row < table.length; row++){

**for** (col = 0; col < table[row].length; col++){

table[row][col] = row + col;

}

}

In Sample Code 23-2, the limits of the **for** loops have been redefined using table.length and table[row].length so that they work with *any* two-dimensional array of **int**s with any number of rows and columns .

B. Passing Two-Dimensional Arrays to Methods

1. The following program in Sample Code 23-3 illustrates parameter passing of an array. The purpose of this method is to print out the array.

Sample Code 23-3

// A program to illustrate 2D array parameter passing

**public** **void** printTable (**int**[][] pTable){

**for** (**int** row = 0; row < pTable.length; row++){

**for** (**int** col = 0; col < pTable[row].length; col++){

System.out.printf(“%4d”, pTable[row][col]);

}

System.out.println();

}

}

2. The printTable method uses a reference parameter, **int**[][] pTable. The local identifier pTable serves as an alias for the actual parameter grid passed to the method.

3. When a program is running and it tries to access an element of an array, the Java virtual machine checks that the array element actually exists. This is called *bounds checking*. If the program tries to access an array element that does not exist, the Java virtual machine will generate an ArrayIndexOutOfBoundsException exception. Ordinarily, this will halt the program.

C. Two-Dimensional Array Algorithms

1. The most common 2-D array algorithms will involve processing the entire grid, usually row-by-row or column-by-column.

2. Problem-solving on a matrix could involve processing:

a. one row

b. one column

c. one cell

d. adjacent cells in different directions

|  |  |
| --- | --- |
| **SUMMARY/ REVIEW:** | Two-dimensional arrays will be applied to two interesting problems. The simulation of life in a petri dish of bacteria will require a two-dimensional array representation. The second and third lab assignments are different versions of the "Knight's Tour" problem - an interesting and demanding chess movement problem. |

**ASSIGNMENT:** Lab Assignment AB23.1, *Life*

Lab Assignment AB23.1, Data File, *life100.txt*

Lab Assignment AB23.1, Supplementary Code, *MakeLife.java*

Lab Assignment AB23.2, *Knight's Tour 1*

Lab Assignment AB23.3, *Knight's Tour 2*

Lab Assignment AB23.3, Data File, *access.txt*

Worksheet AB23.1*, Magic Square*

Worksheet AB23.2, *Two-Dimensional Arrays*