Two-dimensional Arrays

TWO-DIMENSIONAL ARRAYS were introduced in Subsection 3.8.5, but we haven't done much with them since then. A 2D array has a type such as `int[][]` or `String[][]`, with two pairs of square brackets. The elements of a 2D array are arranged in rows and columns, and the `new` operator for 2D arrays specifies both the number of rows and the number of columns. For example,

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

This creates a 2D array of int that has 12 elements arranged in 3 rows and 4 columns. Although I haven't mentioned it, there are initializers for 2D arrays. For example, this statement creates the 3-by-4 array that is shown in the picture below:

```
int[][]  A  =  {  {  1,  0, 12, -1 },
                  {  7, -3,  2,  5 },
                  { -5, -2,  2, -9 }
               };
```

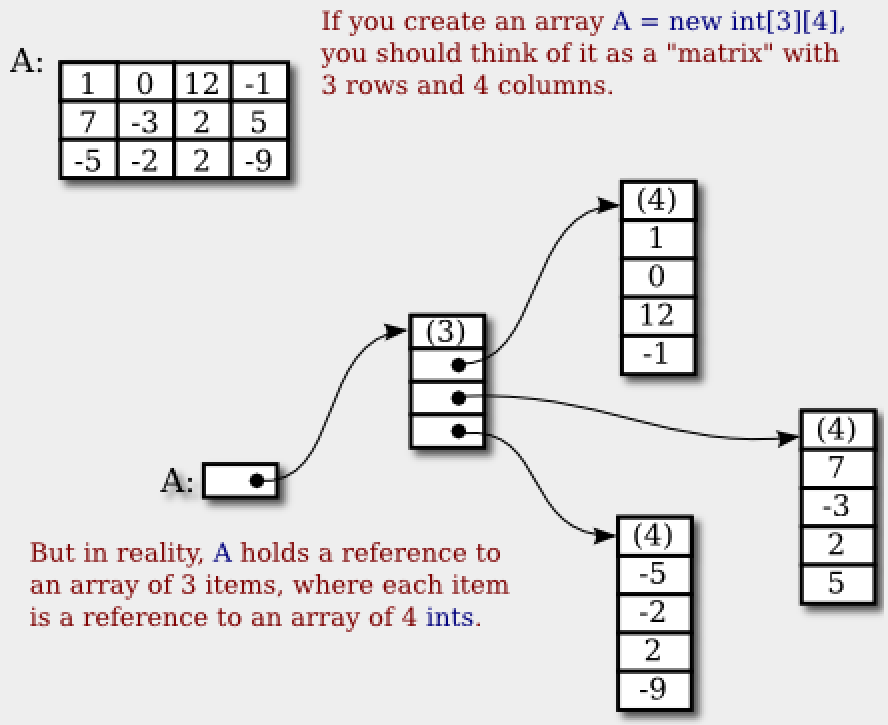
An array initializer for a 2D array contains the rows of A, separated by commas and enclosed between braces. Each row, in turn, is a list of values separated by commas and enclosed between braces. There are also 2D array literals with a similar syntax that can be used anywhere, not just in declarations. For example,

```
A  =  new int[][] {  {  1,  0, 12, -1 },
                     {  7, -3,  2,  5 },
                     { -5, -2,  2, -9 }
                  };
```

All of this extends naturally to three-dimensional, four-dimensional, and even higher-dimensional arrays, but they are not used very often in practice.

7.5.1 The Truth About 2D Arrays

But before we go any farther, there is a little surprise. Java does not actually have two-dimensional arrays. In a true 2D array, all the elements of the array occupy a continuous block of memory, but that's not true in Java. The syntax for array types is a clue: For any type *BaseType*, we should be able to form the type `BaseType[]`, meaning "array of *BaseType*." If we use `int[]` as the base type, the type that we get is "int[][] meaning "array of int[]" or "array of array of int." And in fact, that's what happens. The elements in a 2D array of type int[][] are variables of type int[]. And remember that a variable of type int[] can only hold a pointer to an array of int. So, a 2D array is really an array of pointers, where each pointer can refer to a one-dimensional array. Those one-dimensional arrays are the rows of the 2D array. A picture will help to explain this. Consider the 3-by-4 array A defined above.



If you create an array A = new int[3][4], you should think of it as a "matrix" with 3 rows and 4 columns.
A:
1 0 12 -1
7 -3 2 5
-5 -2 2 -9
A:
(3)
(4)
1
0
12
-1
(4)
7
-3
2
5
(4)
-5
-2
2
-9
But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

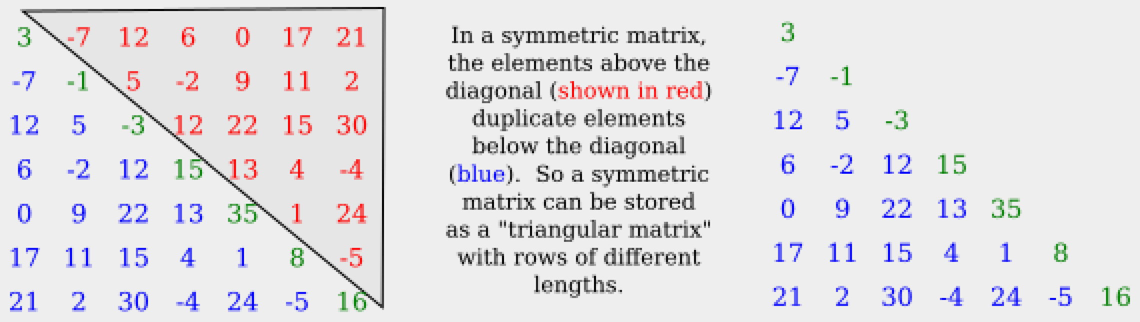
For the most part, you can ignore the reality and keep the picture of a grid in mind. Sometimes, though, you will need to remember that each row in the grid is really an array in itself. These arrays can be referred to as A[0], A[1], and A[2]. Each row is in fact a value of type int[]. It could, for example, be passed to a subroutine that asks for a parameter of type int[].

Some of the consequences of this structure are a little subtle. For example, thinking of a 2D array, A, as an array of arrays, we see that A.length makes sense and is equal to the number of rows of A. If A has the usual shape for a 2D array, then the number of columns in A would be the same as the number of elements in the first row, that is, A[0].length. But there is no rule that says that all of the rows of A must have the same length (although an array created with new BaseType[rows][columns] will always have that form). Each row in a 2D array is a separate one-dimensional array, and each of those arrays can have a different length. In fact, it's even possible for a row to be `null`. For example, the statement

```
A = new int[3][];
```

with no number in the second set of brackets, creates an array of 3 elements where all the elements are `null`. There are places for three rows, but no actual rows have been created. You can then create the rows `A[0]`, `A[1]`, and `A[2]` individually.

As an example, consider a symmetric matrix. A symmetric matrix, `M`, is a two-dimensional array in which the number of rows is equal to the number of columns and satisfying `M[i][j]` equals `M[j][i]` for all `i` and `j`. Because of this equality, we only really need to store `M[i][j]` for `i >= j`. We can store the data in a "triangular matrix":



In a symmetric matrix, the elements above the diagonal (shown in red) duplicate elements below the diagonal (blue). So a symmetric matrix can be stored as a "triangular matrix" with rows of different lengths.

It's easy enough to make a triangular array, if we create each row separately. To create a 7-by-7 triangular array of double, we can use the code segment

```
double[][] matrix = new double[7][]; // rows have not yet been
created!
for (int i = 0; i < 7; i++) {
    matrix[i] = new double[i+1];  // Create row i with i + 1 elements.
}
```

We just have to remember that if we want to know the value of the matrix at `(i,j)`, and if `i < j`, then we actually have to get the value of `matrix[j][i]` in the triangular matrix. And similarly for setting values. It's easy to write a class to represent symmetric matrices:

```
/**
 * Represents symmetric n-by-n matrices of real numbers.
 */
public class SymmetricMatrix {

    private double[][] matrix;  // A triangular matrix to hold the
data.
```

```
    /**
     * Creates an n-by-n symmetric matrix in which all entries are 0.
     */
    public SymmetricMatrix(int n) {
        matrix = new double[n][];
        for (int i = 0; i < n; i++)
            matrix[i] = new double[i+1];
    }

    /**
     * Returns the matrix entry at position (row,col).  (If row < col,
     * the value is actually stored at position (col,row).)
     */
    public double get( int row, int col ) {
        if (row >= col)
            return matrix[row][col];
        else
            return matrix[col][row];
    }

    /**
     * Sets the value of the matrix entry at (row,col).  (If row <
col,
     * the value is actually stored at position (col,row).)
     */
    public void set( int row, int col, double value ) {
        if (row >= col)
            matrix[row][col] = value;
        else
            matrix[col][row] = value;
    }

    /**
     * Returns the number of rows and columns in the matrix.
     */
    public int size() {
        return matrix.length;  // The size is the number of rows.
    }

} // end class SymmetricMatrix
```

This class is in the file *SymmetricMatrix.java*, and a small program to test it can be found in *TestSymmetricMatrix.java*.

By the way, the standard function `Arrays.copyOf()` can't make a full copy of a 2D array in a single step. To do that, you need to copy each row separately. To make a copy of a two-dimensional array of int, for example:

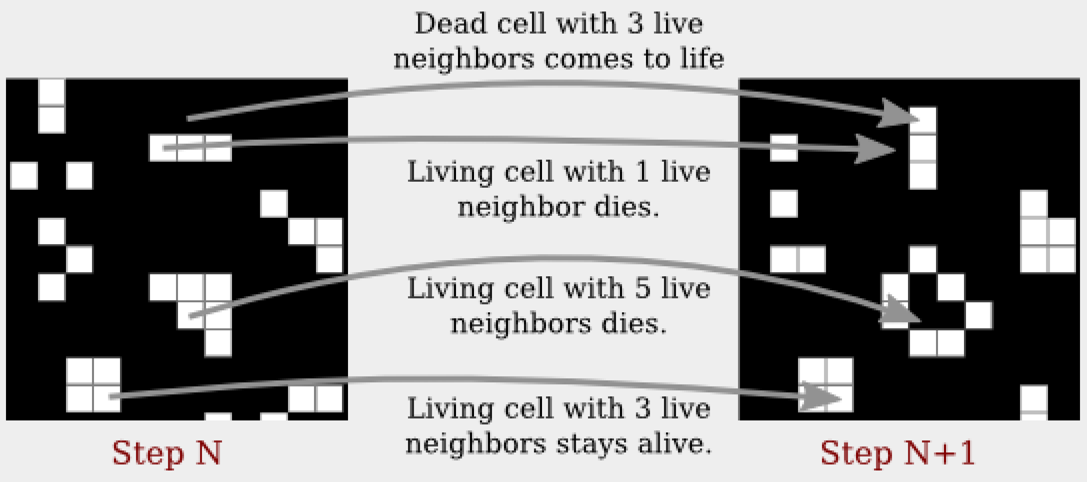
```
int[][] B = new int[A.length][];  // B has as many rows as A.
for (int i = 0; i < A.length; i++) {
    B[i] = Arrays.copyOf(A[i], A[i].length)); // Copy row i.
}
```

7.5.2 Conway's Game Of Life

As an example of more typical 2D array processing, let's look at a very well-known example: John Conway's Game of Life, invented by mathematician John Horton Conway in 1970. This Game of Life is not really a game (although sometimes it's referred to as a "zero-person game" that plays itself). It's a "two-dimensional cellular automaton." This just means that it's a grid of cells whose content changes over time according to definite, deterministic rules. In Life, a cell can only have two possible contents: It can be "alive" or "dead." We will use a 2D array to represent the grid, with each element of the array representing the content of one cell in the grid. In the game, an initial grid is set up in which each cell is marked as either alive or dead. After that, the game "plays itself." The grid evolves through a series of time steps. The contents of the grid at each time step are completely determined by the contents at the previous time step, according to simple rules: Each cell in the grid looks at its eight neighbors (horizontal, vertical, and diagonal) and counts how many of its neighbors are alive. Then the state of the cell in the next step is determined by the rules:

- If the cell is alive in the current time step: If the cell has 2 or 3 living neighbors, then the cell remains alive in the next time step; otherwise, it dies. (A living cell dies of loneliness if it has 0 or 1 living neighbor, and of overcrowding if it has more than 3 living neighbors.)
- If the cell is dead in the current time step: If the cell has 3 living neighbors, then the cell becomes alive in the next time step; otherwise, it remains dead. (Three living cells give birth to a new living cell.)

Here's a picture of part of a Life board, showing the same board before and after the rules have been applied. The rules are applied to every cell in the grid. The picture shows how they apply to four of the cells:


Dead cell with 3 live neighbors comes to life
Living cell with 1 live neighbor dies.
Living cell with 5 live neighbors dies.
Living cell with 3 live neighbors stays alive.
Step N
Step N+1

The Game of Life is interesting because it gives rise to many interesting and surprising patterns. (Look it up on Wikipedia.) Here, we are just interested in writing a program to simulate the game. The complete program can be found in the file *Life.java*. In the program, the life grid is shown as a grid of squares in which dead squares are black and living squares are white. (The program uses *MosaicCanvas.java* from Section 4.7 to represent the grid, so you will also need that file to compile and run the program.) In the program, you can fill the life board randomly with dead and alive cells, or you can use the mouse to set up the game board. There is a "Step" button that will compute one time-step of the game, and a "Start" button that will run time steps as an animation.

We'll look at some of the array processing involved in implementing the Game of Life for this program. Since a cell can only be alive or dead, it is natural to use a two-dimensional array of `boolean[][]` to represent the states of all the cells. The array is named `alive`, and `alive[r][c]` is true when the cell in row `r`, column `c` is alive. The number of rows and the number of columns are equal and are given by a constant, `GRID_SIZE`. So, for example, to fill the Life grid with random values, the program uses simple nested `for` loops:

```
for (int r = 0; r < GRID_SIZE; r++) {
    for (int c = 0; c < GRID_SIZE; c++) {
           // Use a 25% probability that the cell is alive.
        alive[r][c] = (Math.random() < 0.25);
    }
}
```

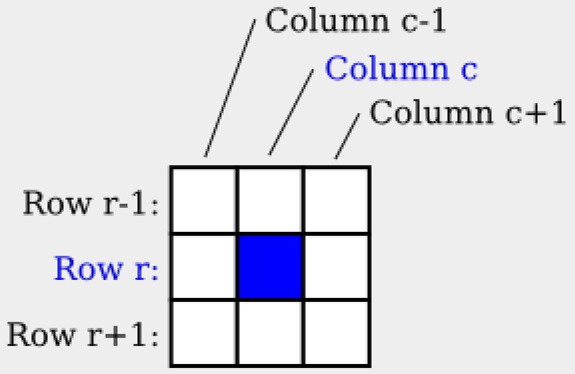
Note that the expression `(Math.random() < 0.25)` is a true/false value that can be assigned to a boolean array element. The array is also used to set the color of the cells on the screen. Since the grid of cells is displayed on screen as a *MosaicCanvas*, setting the colors is done using the MosaicCanvas API. Note that the actual drawing is done in the *MosaicCanvas* class (which has its own 2D array of type `Color[][]` to keep track of the colors of each cell). The Life program just has to set the colors in the mosaic, using the MosaicCanvas API. This is done in the program in a method named `showBoard()` that is called each time the board changes. Again, simple nested `for` loops are used to set the color of each square in the grid:

```
for (int r = 0; r < GRID_SIZE; r++) {
    for (int c = 0; c < GRID_SIZE; c++) {
        if (alive[r][c])
            display.setColor(r,c,Color.WHITE);
        else
            display.setColor(r,c,null);  // Shows the background
color, black.
    }
}
```

Of course, the most interesting part of the program is computing the new state of the board by applying the rules to the current state. The rules apply to each individual cell, so again we can use nested for loops to work through all the cells on the board, but this time the processing is more complicated. Note first that we can't make changes to the values in the array as we work through it, since we will need to know the old state of a cell when processing its neighboring cells. In fact, the program uses a second array to hold the new board as it is being created. When the new board is finished, it can be substituted for the old board. The algorithm goes like this in pseudocode:

```
let newboard be a new boolean[][] array
for each row r:
    for each column c:
        Let N be the number of neighbors of cell (r,c) in the alive
array
        if ((N is 3) or (N is 2 and alive[r][c]))
            newboard[r][c] = true;
        else
            newboard[r][c] = false;
alive = newboard
```

Note that at the end of the process, `alive` is pointing to a new array. This doesn't matter as long as the contents of the array represent the new state of the game. The old array will be garbage collected. The test for whether `newboard[r][c]` should be `true` or `false` might not be obvious, but it implements the rules correctly. We still need to work on counting the neighbors. Consider the cell in row `r` and column `c`. If it's not at an edge of the board, then it's clear where its neighbors are:



The row above row number `r` is row number `r-1`, and the row below is `r+1`. Similarly for the columns. We just have to look at the values of `alive[r-1][c-1]`, `alive[r-1][c]`, `alive[r-1][c+1]`, `alive[r][c-1]`, `alive[r][c+1]`, `alive[r+1][c-1]`, `alive[r+1][c]`, and `alive[r+1][c+1]`, and count the number that are `true`. (You should make sure that you understand how the array indexing works here.)

But there is a problem when the cell is along one of the edges of the grid. In that case, some of the array elements in the list don't exist, and an attempt to use them will cause an exception. To avoid the exception, we have to give special consideration to cells along the edges. One idea is that before referencing any array element, check that the array element actually exists. In that case, the code for neighbor counting becomes

```
if (r-1 >= 0 && c-1 >= 0 && alive[r-1][c-1])
    N++; // A cell at position (r-1,c-1) exists and is alive.
if (r-1 >= 0 && alive[r-1][c])
    N++; // A cell at position (r-1,c) exists and is alive.
if (r-1 >= 0 && c+1 <= GRID_SIZE && alive[r-1][c+1])
    N++; // A cell at position (r-1,c+1) exists and is alive.
// and so on...
```

All the possible exceptions are avoided. But in my program, I actually do something that is common in 2D computer games—I pretend that the left edge of the board is attached to the right edge and the top edge to the bottom edge. For example, for a cell in row 0, we say that the row "above" is actually the bottom row, row number `GRID_SIZE-1`. I use variables to represent the positions above, below, left, and right of a given cell. The code turns out to be simpler than the code shown above. Here is the complete method for computing the new board:

```
private void doFrame() { // Compute the new state of the Life board.
    boolean[][] newboard = new boolean[GRID_SIZE][GRID_SIZE];
    for ( int r = 0; r < GRID_SIZE; r++ ) {
        int above, below; // rows considered above and below row
number r
        int left, right;  // columns considered left and right of
column c
        above = r > 0 ? r-1 : GRID_SIZE-1;  // (for "?:" see
Subsection 2.5.5)
        below = r < GRID_SIZE-1 ? r+1 : 0;
        for ( int c = 0; c < GRID_SIZE; c++ ) {
            left =  c > 0 ? c-1 : GRID_SIZE-1;
            right = c < GRID_SIZE-1 ? c+1 : 0;
            int n = 0; // number of alive cells in the 8 neighboring
cells
            if (alive[above][left])
                n++;
            if (alive[above][c])
                n++;
            if (alive[above][right])
                n++;
            if (alive[r][left])
                n++;
            if (alive[r][right])
                n++;
            if (alive[below][left])
                n++;
            if (alive[below][c])
                n++;
            if (alive[below][right])
                n++;
            if (n == 3 || (alive[r][c] && n == 2))
                newboard[r][c] = true;
            else
                newboard[r][c] = false;
        }
    }
    alive = newboard;
}
```

Again, I urge you to check out the source code, *Life.java*, and try the program. Don't forget that you will also need *MosaicCanvas.java*.
