Using Multidimensional Arrays

- Such complex arrays are useful for storing data that has multiple indices (i.e., "keys" to look it up)
- Example: hourly temperatures for a weather station over a period of 3 years of measurements, 2015–2017

```java
int years = 3;
int days = 365;
int hours = 24;
double[][][] temps = new double[years][days][hours];
```

Now, e.g., `temps[1][0][14]` can store the temperature for year 2016, 01 January, 1400 hours

Creating Multidimensional Arrays

- Such an array has 2 "rows", each with 3 "columns"
- We can also create such objects as arrays of arrays
- Just as we can declare a one-dimensional array directly:

```java
int[] arr1D = {3, 5, 6};
```

- We can do the same for a 2-dimensional (or more) array:

```java
int[][] arr2D = { {3, 5, 6}, {2, 4, 7} };
```
Using Multidimensional Arrays

Just as a for loop is a nice way to access a 1-dimensional array, nested for loops work well for n-dimensional ones.

```
int[][] twoD = new int[20][15];
for ( int row = 0; row < 20; row++ )
{
    for ( int col = 0; col < 15; col++ )
    {
        int n = twoD[row][col];
        System.out.println(n);
    }
}
```

Loops over all index-pairs

What is “wrong” with this code?

Avoiding “Magic Numbers” in Code

- Hard-coded values lead to fragile code: hard to get right, hard to debug, prone to failure

```
int[][] twoD = new int[20][15];
for ( int row = 0; row < 20; row++ )
{
    for ( int col = 0; col < 15; col++ )
    {
        int n = twoD[row][col];
        System.out.println(n);
    }
}
```

Responsive values make for durable code: easier to get right, easier to debug, easier to modify, less prone to failure

```
int[][] twoD = new int[20][15];
for ( int row = 0; row < twoD.length; row++ )
{
    for ( int col = 0; col < twoD[row].length; col++ )
    {
        int n = twoD[row][col];
        System.out.println(n);
    }
}
```

Number of rows in a two dimensional array is: `<ARRAY NAME>.length`
Each row is itself an array, with length (# columns): `<ARRAY NAME>[row].length`

Avoiding “Magic Numbers” in Code

```java
private Rectangle[][] fillBoard()
{
    Rectangle[][] grid = new Rectangle[8][8];
    for ( int row = 0; row < grid.length; row++ )
    {
        for ( int col = 0; col < grid[row].length; col++ )
        {
            // create a new Rectangle, using col and row to set position
            Rectangle r = new Rectangle( col * 50, row * 50, 50, 50 );
            // alternate white/black and change pattern each row
            if ( (row + col) % 2 == 0 )
                r.setBackground(Color.white);
            grid[row][col] = r;
        }
    }
    return grid;
}
```
Nested Loops & Multidimensional Arrays

```java
private void fillWindow( Rectangle[][] squares )
{
    // again loop over each pair of indices in array;
    // here, we loop over the length of the array (i.e., # of rows)
    for ( int row = 0; row < squares.length; row++ )
    {
        // here we loop over the length of one row (i.e., # of columns)
        for ( int col = 0; col < squares[row].length; col++ )
        {
            // we add each object to window in turn
            window.add( squares[row][col] );
        }
    }
}
```

Overall length of 2-dimensional array (# rows) is:
<ARRAY NAME>.length
Each row is itself an array, with length (# columns):
<ARRAY NAME>[row].length

Creating Multidimensional Arrays

- Such an array also has 2 "rows", each with 3 "columns"
  - Each "row" is itself an array of int objects
    ```java
    int[][] arr2D = { {3, 5, 6}, {2, 4, 7} };
    ```

- What happens if we declare using different sizes of array for each individual row?
  ```java
  int[][] arr2D = { {2, 4}, {7, 3, 5, 6} };
  ```
  This is perfectly OK! (in Java)

Uneven (or “Ragged”) Arrays

- This array consists of two different arrays, one which has size 2, and the other which has size 4
  ```java
  int[][] arr2D = { {2, 4}, {7, 3, 5, 6} };
  ```

- One can also declare the number of rows up-front, leaving the columns in each row to be filled in later
  ```java
  int[][] arr2D = new int[2][];
  arr2D[0] = new int[2];
  arr2D[0][0] = 2;
  arr2D[0][1] = 4;
  arr2D[1] = {7, 3, 5, 6};
  ```
  Leave 2nd index blank
  Two different ways to fill in the rows
Another Reason to Avoid “Magic Numbers”

- As discussed, hard-coded values can lead to errors and make programs fragile and hard to debug.
- With uneven arrays, this won’t even work at all!

```java
int[][] twoD = new int[2][3];
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        int n = twoD[i][j];
        System.out.println(n);
    }
}
```

Error on first row, as twoD[0][2] doesn’t even exist!

Proper Looping with Ragged Arrays

- Since a multi-dimensional array is an array of arrays, each element of the main array also has its own length variable.

```java
int[][] twoD = new int[2][];
twoD[0] = {1, 2};
twoD[1] = {3, 4, 5};
for (int i = 0; i < twoD.length; i++) {
    for (int j = 0; j < twoD[i].length; j++) {
        int n = twoD[i][j];
        System.out.println(n);
    }
}
```

Length != 2 OR 3 (depending on row)

This Week & Next

- **Meetings this week:**
  - Monday/Wednesday: regular classroom
  - Tuesday/Friday: in the CS Lab (16 Wing)

- **Reading 05:** Ch. 6 due Thursday, Oct 31 at Noon
- **Program 05:** due 11:59 PM, Wednesday October 30
- **Midterm 2:** Monday November 11

- **Office Hours:** Wing 212
  - Monday/Friday: 2:15 PM–3:15 PM
  - Tuesday: 2:30 PM–1:30 PM
  - Wednesday: 12:05 PM–1:00 PM