CMSC 131: Chapter 20 (Supplement)
Multidimensional Arrays

Multidimensional Arrays

Multidimensional Arrays: we have discussed the notions of:

Array of primitive types: Consider the declaration:

```
char[ ] c = new char[5];
```

c: is of type **char[ ]** , an **array** of characters.
c[2] and **c[i]**: are of type **char**, a **single** character.

Array of class objects:

```
String[ ] s = new String[10];
```

s: is of type **String[ ]**, an **array** of strings.
s[3] and **s[j]**: are of type **String**, a **single** String.

Can we have an array of arrays? Yes, of course. In Java this is called a multidimensional array.

2-dimensional Arrays

2-dimensional Arrays: Let us first consider representing a page of text for use by a word processor.

- Each **line** of text is an **array of characters** (say, 100 per line).
- Each **page** of text is an **array of lines**, that is, an array of arrays (say, 50 lines per page).

Declarations:

```
char[ ] [ ] page = new char[50][100];
```

or equivalently:

```
char[ ] [ ] page;                    // this declares the variable
page = new char[50][100];            // this allocates storage
```

Access:

page: is of type **char[ ][ ]**, an **array of array** of characters (whole page).
page[4] : is of type **char[ ]**, an **array** of characters (a single line).
page[4][23]: is of type **char**, a **single** character (character 23 of line 4).

Conceptual Layout

Let's be more concrete. Consider the following declaration:

```
char[ ] [ ] a = new char[5][8];
```

Conceptually, this is laid out as the following table:

a[0][0]	a[0][1]	a[0][2]	…	a[0][7]
a[1][0]	a[1][1]	a[1][2]	…	a[1][7]
a[2][0]	a[2][1]	a[2][2]	…	a[2][7]
a[3][0]	a[3][1]	a[3][2]	…	a[3][7]
a[4][0]	a[4][1]	a[4][2]	…	a[4][7]

By convention the 1st index is the row, the 2nd is the column.

Memory Layout

(Figure omitted)

To create an array, Java allocates space for the **array of array references**, and then allocates space for the **individual arrays**.

Multidimensional Array Length

Consider the declaration:

```
char[ ] [ ] a = new char[5][8];
```

What is the meaning of **a.length**?

- 5? 8? 40?
- Undefined?

Ans: 5. This is clear from the illustration on the previous page. To Java, a is an array of 5 references to other arrays.

What is the meaning of **a[2].length**?

Ans: 8, because a[2] is an array of 8 characters.

Example: Blank out the array a:

```
for ( int r = 0; r < a.length; r++ )
   for ( int c = 0; c < a[r].length; c++ )
        a[r][c] = ' ';
```

Ragged Arrays

When you allocate an array of arrays, do all the arrays have to be of the **same size**?

No. When the arrays have different sizes, it is called a **ragged array**. You must explicitly specify their sizes.

```
char[ ][ ] a = new char[4][ ];
a[0] = new char[8];
a[1] = new char[3];
a[2] = new char[5];
a[3] = new char[1];
a[4] = null;
```

Multidimensional Initializers

Recall that a 1-dimensional array can be initialized using:

```
int[ ] quizScores = { 90, 82, 75, 66 };
```

In a similar way, you can initialize a 2-dimensional array:

```
int[ ][ ] quizScores = { { 90, 82, 75, 66 },
                         { 85 },
                         { 45, 77, 99 } };
```

This allocates and initializes an array with 3 rows and 4 columns:

Print the array:

```
for ( int r = 0; r < quizScores.length; r++ ) {
    System.out.print( "Scores for student " + r + ":");
    for ( int c = 0; c < quizScores[r].length; c++ )
        System.out.print( " " + quizScores[r][c] );
    System.out.println( );
}
```