(b) Switch the values at either end of the array.

(c) Change any negative values to positive values (of the same magnitude).

(d) Set the variable `sampleSum` to the sum of the values of all the elements.

(e) Print the contents of the odd-numbered locations.

3. Write a method `max` that has one `double` array parameter. The method should return the value of the largest element in the array.

4. Complete the definition of the method `equals` so that it returns true if and only if its two array parameters contain equal elements.

```
public static boolean equals (double[] a, double[] b)
```

5. Write a program that repeatedly prompts the user to supply scores (out of 10) on a test. The program should continue to ask the user for marks until a negative value is supplied. Any values greater than ten should be ignored. Once the program has read all the scores, it should produce a table with the following headings:

```
Score          # of Occurrences
```

The program should then calculate the mean score, rounded to one decimal place.

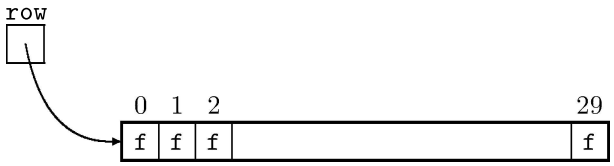
8.3 Multi-Dimensional Arrays

Suppose that, because of your growing reputation as an expert with computers, you have been asked to assist the local little theatre group in maintaining records of seats sold for their upcoming play. The auditorium that the group uses has twenty-five rows, each of which contain thirty seats. To keep track of sales, it would be useful to have an array of boolean values in which an element has the value `true` if and only if the seat has been sold.

Sales of seats in a single row could be monitored using a boolean array of size thirty. This could be declared with the statement

```
boolean[] row = new boolean[30];
```

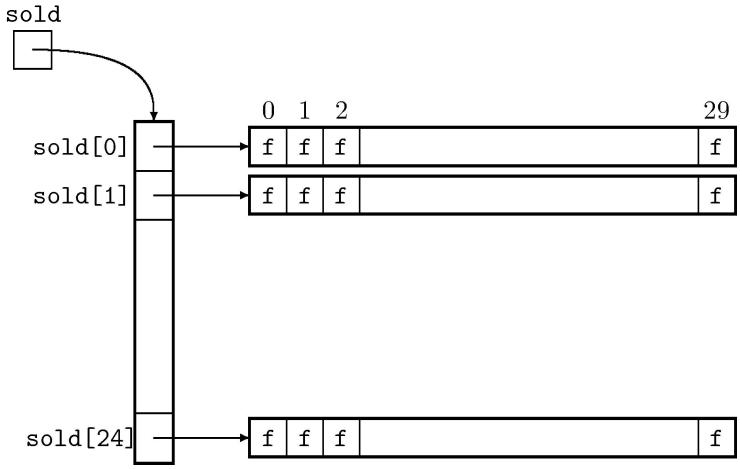
to produce the following result.



To monitor sales for the entire auditorium, we want one of these arrays for each row. We can do this easily by writing

```
boolean[][] sold = new boolean[25][30];
```

The results of this declaration are shown in the next diagram.



Although this may be surprising at first, analysis may make it seem perfectly reasonable. Recall that the general form of a declaration of an array was

<type>[] <identifier> = new <type>[<expression>];

Comparing this form with our declaration of the `sold` array, the <type> of the components of `sold` is `boolean[]`; each element of the `sold` array is of type `boolean[]`, a reference to an array of boolean values. As the diagram indicates, the identifiers of these references are `sold[0]`, `sold[1]`, ..., `sold[24]`.

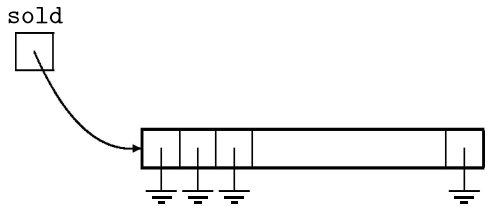
Each `sold[i]` value refers to a boolean array, each of whose elements are initialized to `false` (indicated by the letter `f` in the diagram). The elements of the boolean arrays are identified by two indices — the first for the index of the array reference and the second for the index of the element

within its array. For example, the element in the upper left hand corner of the diagram has identifier `sold[0][0]` while the element in the lower right hand corner has identifier `sold[24][29]`.

We could have created this structure in stages. As a first step, we could have written

```
boolean[][] sold = new boolean[25][];
```

to create `sold`, a reference to an array of 25 variables of type `boolean[]`. As usual, when we create a new array, the values are initialized. Since the type of the values is a reference type, each one is initialized to `null`.



Now, to create the arrays to represent each row, we can use a loop.

```
for (int i = 0; i < sold.length; i++)
  sold[i] = new boolean[30];
```

The loop would be executed 25 times (the value of `sold.length`). Each time around, one more array of 30 elements would be generated and initialized. The final result would be what we want: 25 arrays of 30 boolean elements representing the seats sold in the auditorium. An array of arrays like this one is known as a *two-dimensional array.*

Often, we want to use two-dimensional arrays to represent values in a simple rectangular table. In such cases, we can think of a two-dimensional array as consisting of rows and columns. If we do this, it is customary to think of the index representing a row preceding that representing a column.

Example 1

A table of `int` values with 3 rows and 6 columns can be created using the declaration

```
int[][] table = new int[3][6];
```

As we have seen, this really creates three arrays each containing six elements but we can think of it as shown in the following diagram.

		columns					
		0	1	2	3	4	5
rows	0	0	0	0	0	0	0
	1	0	0	0	0	0	0
	2	0	0	0	0	0	0

Just as `for` statements are usually used to index through a one-dimensional array, nested `for` statements are usually used to index through a two-dimensional array. For the array `table` shown in Example 1 we can initialize the elements to one using the following fragment.

```
for (int row = 0; row < table.length; row++)
  for (int col = 0; col < table[0].length; col++)
    table[row][col] = 1;
```

Notice the way in which the upper bounds of the loops are written. The number of rows in `table` is given by `table.length` while the number of columns in row zero (and all other rows) is given by `table[0].length`.

Two-dimensional arrays can be initialized as they are being declared, in a manner similar to that used for one-dimensional arrays.

Example 2

Suppose we want to create a table with three rows and four columns, initialized as shown in the diagram.

5	2	7	3
6	8	7	9
4	5	2	3

We can do this with the following statement.

```
int[][] scores = {{5,2,7,3},
                  {6,8,7,9},
                  {4,5,2,3}};
```

On the right side of the assignment statement, the elements of each row of the array are enclosed by brace brackets and separated by commas. Similarly, the sets of values for each row are themselves enclosed by brace brackets and separated by commas. The statement could have been written

on one line but we wrote it the way that we did to make the structure of the array clearer to a human reader.

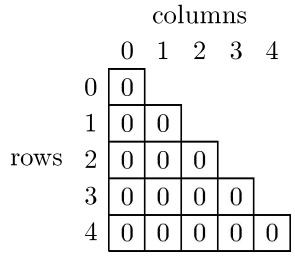
The one-dimensional arrays that compose a two-dimensional array need not all be of the same length, as the next example illustrates.

Example 3

The following fragment creates a triangular array of `double` values with one element in the first row, two in the second, and so on for a total of five rows.

```
double[][] triTable = new double[5][];
for (int row = 0; row < triTable.length; row++)
  triTable[row] = new double[row+1];
```

We can picture the resulting array as follows:



Array declarations like the one for `triTable` in Example 3, in which the second dimension's size was unspecified, are legal. However, it is *not* legal to have the first dimension's size unspecified. For example, the declaration

```
int[][] badArray = new int[][20];
```

is not permitted.

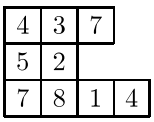
Non-rectangular two-dimensional arrays are sometimes called *ragged arrays*. We can initialize ragged arrays in their declarations, just as we did for rectangular arrays.

Example 4

The statement

```
int[][] ragged = {{4,3,7},
                  {5,2},
                  {7,8,1,4}};
```

creates the array whose elements are illustrated in the following diagram.



To process the elements of a ragged array, we must change our loop structure slightly to take account of the fact that the length of each row may be different.

Example 5

The following fragment could be used to find the sum of the elements of a ragged two-dimensional array like the one shown in the previous example. Notice that the upper bound of the inner loop now depends on the length of each row of the array.

```
int sum = 0;
for (int row = 0; row < ragged.length; row++)
  for (int col = 0; col < ragged[row].length; col++)
    sum += ragged[row][col];
```

Multi-dimensional arrays are not limited to two dimensions. We can create three-dimensional arrays, four-dimensional arrays, or arrays of even higher dimensions. These higher-dimensional arrays are usually rectangular, but they need not be. We suggested that a two-dimensional rectangular array could be visualized as a table with rows and columns. Similarly, a rectangular three-dimensional array can be visualized as a set of tables,

perhaps a book containing a number of tables, one on each page. The indices could then be thought of as representing the pages, rows, and columns (in that order). A four-dimensional array could then be thought of as a number of volumes of books, all of the same size, with an index specifying a volume, page, row, and column. As with two-dimensional arrays, we need not specify all the dimensions in a declaration of a higher-dimensional array.

Example 6

The following declarations are valid.

(a) `char[][][] goodOne = new char[5][6][3];`
This creates an array of $5 \times 6 \times 3 = 90$ `char` elements or, more accurately, five sets of six sets of three-element `char` arrays.

(b) `byte[][][] goodTwo = new byte[20][][];`
This creates 20 arrays of type `byte[][]`. Each of these 20 arrays (called `goodTwo[0]`, `goodTwo[1]` and so on) will be initialized to `null`.

(c) `double[][][] goodThree = new double[50][100][];`
This creates an array of 50 references to arrays of 100 references to arrays with elements of type `double`. These 5000 array references will be set to `null`.

Example 7

These declarations are not valid.

(a) `float[][] badOne = new float[][50];`
This attempt to create an unspecified number of arrays that are each of type `float[50]` will fail. The first dimension must always be specified.

(b) `boolean[][][] badTwo = new boolean[10][][25];`
Although the first dimension has been specified here, the second and third dimension show the same pattern that was exhibited in `badOne`.

Finally, we note that, just as one-dimensional array references can be used as parameters and return values of methods, so too can multi-dimensional array references. Of course, the dimensions of arguments and parameters must match and their types must be appropriate. It is also possible to pass sub-arrays to methods, as the next example illustrates.

Example 8

Suppose that a program had a method with the following header:

```
public static void process (double[] row)
```

Suppose also that in our `main` method, say, we had made the following declaration:

```
double[][] table = new double[40][60];
```

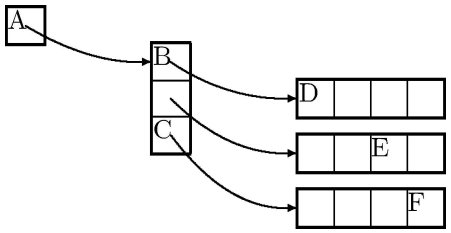
We could then call `process` from `main` by writing

```
process(table[20]);
```

This works because `table[20]` is a reference to a one-dimensional array of `double` values and that exactly matches the type of the parameter `row`.

Exercises 8.3

1. The diagram shows an array declared by the statement
   ```
   int[][] a = new int[3][4];
   ```
   State the identifier of each cell marked by a letter.



2. How many elements would there be in each of the arrays created by the following declarations?

   (a) `double[][] first = new double[25][40];`

(b) `boolean[][][] second = new boolean[3][6][50]`

(c) `char[][] third = new char[60][40];`

(d) `long[][][] fourth = new long[5][10][20];`

3. Suppose that the following declarations have been made:

```
int a[][] = {{4,2,7},
             {3,9,1}};
int i, j;
```

Determine what would be printed by each fragment.

(a)
```
for (i = 0; i < a.length; i++)
{
  for (j = 0; j < a[0].length; j++)
    System.out.print(a[i][j]);
  System.out.println();
}
```

(b)
```
for (i = 0; i < a[0].length; i++)
{
  for (j = 0; j < a.length; j++)
    System.out.print(a[j][i]);
  System.out.println();
}
```

(c)
```
for (i = a.length - 1; i >= 0; i--)
{
  for (j = 0; j < a[0].length; j++)
    System.out.print(a[i][j]);
  System.out.println();
}
```

4. For the array given in the previous question, write a fragment that would print the elements of the array in the form

```
17
92
34
```

5. Write a method `sum` having one `double[][]` parameter. The method should return the sum of the elements of the array passed to it. You may assume that the array is rectangular.

6. Write a method `max` that will return the maximum value of the elements in a two-dimensional array of `int` values. Do *not* assume that the array is rectangular.

7. Write a method `print` that could be used to print a two-dimensional ragged array of `int` values. Each row of elements should be printed on its own line with one blank between each element.

8. Write a method `size` that has one `int[][][]` parameter. The method should return the number of elements in the array. Do not make any assumptions about regularity of the array.

8.4 Arrays of Objects

We said earlier that the elements of an array could be of any type. This includes the possibility that elements can be objects. To illustrate this, consider a class that could be used to represent complex numbers. We could begin to define such a class by writing

```
class Complex
{
  private double re;
  private double im;
}
```

To create an array of `Complex` objects, we could proceed as follows. In our `main` method, say, we could write

```
Complex[] points = new Complex[100];
```

This would create an array of 100 values (`points[0]`, `points[1]`, ..., `points[99]`) of type reference to `Complex`. As with the declaration of any array of references, each of these would be initialized to `null`. To actually create 100 `Complex` objects, we could use the following loop.

```
for (int i = 0; i < points.length; i++)
  points[i] = new Complex();
```

Assuming that the `Complex` class has a `toString` method, we could print the value of the fifth object in the array by writing