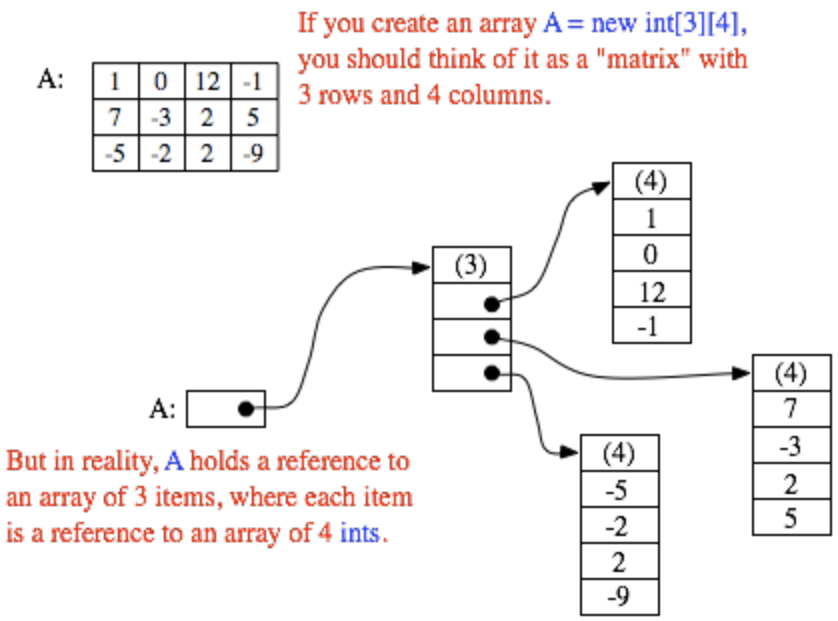
MULTI-DIMENSIONAL ARRAYS

ANY TYPE CAN BE USED as the base type of an array. You can have an array of ints, an array of *Strings*, an array of *Objects*, and so on. In particular, since an array type is a first-class Java type, you can have an array of arrays. For example, an array of ints has type `int[]`. This means that there is automatically another type, `int[][]`, which represents an "array of arrays of `ints`". Such an array is said to be a two-dimensional array. Of course once you have the type `int[][]`, there is nothing to stop you from forming the type `int[][][]`, which represents a three-dimensional array -- and so on. There is no limit on the number of dimensions that an array type can have. However, arrays of dimension three or higher are fairly uncommon, and I concentrate here mainly on two-dimensional arrays. The type `BaseType[][]` is usually read "two-dimensional array of *BaseType*" or "*BaseType* array array".

Creating Two-dimensional Arrays

The declaration statement "`int[][] A;`" declares a variable named `A` of type `int[][]`. This variable can hold a reference to an object of type `int[][]`. The assignment statement "`A = new int[3][4];`" creates a new two-dimensional array object and sets `A` to point to the newly created object. As usual, the declaration and assignment could be combined in a single declaration statement "`int[][] A = new int[3][4];`". The newly created object is an array of arrays-of-ints. The notation `int[3][4]` indicates that there are 3 arrays-of-`ints` in the array `A`, and that there are 4 ints in each array-of-ints. However, trying to think in such terms can get a bit confusing -- as you might have already noticed. So it is customary to think of a two-dimensional array of items as a rectangular grid or matrix of items. The notation "`new int[3][4]`" can then be taken to describe a grid of `ints` with 3 rows and 4 columns. The following picture might help:



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[]`.

The notation `A[1]` refers to one of the rows of the array `A`. Since `A[1]` is itself an array of ints, you can use another subscript to refer to one of the positions in that row. For example, `A[1][3]` refers to item number 3 in row number 1. Keep in mind, of course, that both rows and columns are numbered starting from zero. So, in the above example, `A[1][3]` is 5. More generally, `A[i][j]` refers to the grid position in row number `i` and column number `j`. The 12 items in `A` are named as follows:

```
A[0][0]    A[0][1]    A[0][2]    A[0][3]
A[1][0]    A[1][1]    A[1][2]    A[1][3]
A[2][0]    A[2][1]    A[2][2]    A[2][3]
```

`A[i][j]` is actually a variable of type int. You can assign integer values to it or use it in any other context where an integer variable is allowed.

It might be worth noting that `A.length` gives the number of rows of `A`. To get the number of columns in `A`, you have to ask how many `ints` there are in a row; this number would be given by `A[0].length`, or equivalently by `A[1].length` or `A[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.)

Three-dimensional arrays are treated similarly. For example, a three-dimensional array of `int`s could be created with the declaration statement "`int[][][] B = new int[7][5][11];`". It's possible to visualize the value of `B` as a solid 7-by-5-by-11 block of cells. Each cell holds an int and represents one position in the three-dimensional array. Individual positions in the array can be referred to with variable names of the form `B[i][j][k]`. Higher-dimensional arrays follow the same pattern, although for dimensions greater than three, there is no easy way to visualize the structure of the array.

It's possible to fill a multi-dimensional array with specified items at the time it is declared. Recall that when an ordinary one-dimensional array variable is declared, it can be assigned an "array initializer," which is just a list of values enclosed between braces, { and }. Array initializers can also be used when a multi-dimensional array is declared. An initializer for a two-dimensional array consists of a list of one-dimensional array initializers, one for each row in the two-dimensional array. For example, the array `A` shown in the picture above could be created with:

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

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

Using Two-dimensional Arrays

Just as in the case of one-dimensional arrays, two-dimensional arrays are often processed using `for` statements. To process all the items in a two-dimensional array, you have to use one `for` statement nested inside another. If the array `A` is declared as

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

then you could store a 17 into each location in `A` with:

```
for (int row = 0;  row < 3;  row++) {
   for (int column = 0;  column < 4;  column++) {
      A[row][column] = 17;
```

```
   }
}
```

The first time the outer `for` loop executes (with `row` = 0), the inner for loop fills in the four values in the first row of A, namely `A[0][0] = 17`, `A[0][1] = 17`, `A[0][2] = 17`, and `A[0][3] = 17`. The next execution of the outer `for` loop fills in the second row of `A`. And the third and final execution of the outer loop fills in the final row of `A`.

Similarly, you could add up all the items in `A` with:

```
int sum = 0;
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 4; j++)
      sum = sum + A[i][j];
```

This could even be done with nested for-each loops. Keep in mind that the elements in `A` are objects of type `int[]`, while the elements in each row of `A` are of type `int`:

```
int sum = 0;
for ( int[] row : A ) {       // For each row in A...
   for ( int item : row )     // For each item in that row...
      sum = sum + item;       // Add item to the sum.
}
```

To process a three-dimensional array, you would, of course, use triply nested `for` loops.

A two-dimensional array can be used whenever the data that is being represented can be arranged into rows and columns in a natural way. Often, the grid is built into the problem. For example, a chess board is a grid with 8 rows and 8 columns. If a class named *ChessPiece* is available to represent individual chess pieces, then the contents of a chess board could be represented by a two-dimensional array:

```
ChessPiece[][]  board  =  new ChessPiece[8][8];
```

Or consider the "mosaic" of colored rectangles used in an example in Subsection 4.6.2. The mosaic is implemented by a class named *MosaicCanvas.java*. The data about the color of each of the rectangles in the mosaic is stored in an instance variable named `grid` of type `Color[][]`. Each position in this grid is occupied by a value of type *Color*. There is one position in the grid for each colored rectangle in the mosaic. The actual two-dimensional array is created by the statement:

```
grid  =  new Color[ROWS][COLUMNS];
```

where ROWS is the number of rows of rectangles in the mosaic and COLUMNS is the number of columns. The value of the Color variable grid[i][j] is the color of the rectangle in row number i and column number j. When the color of that rectangle is changed to some color, c, the value stored in grid[i][j] is changed with a statement of the form "grid[i][j] = c;". When the mosaic is redrawn, the values stored in the two-dimensional array are used to decide what color to make each rectangle. Here is a simplified version of the code from the *MosaicCanvas* class that draws all the colored rectangles in the grid. You can see how it uses the array:

```
int rowHeight = getHeight() / ROWS;
int colWidth = getWidth() / COLUMNS;
for (int row = 0; row < ROWS; row++) {
   for (int col = 0; col < COLUMNS; col++) {
       g.setColor( grid[row][col] ); // Get color from array.
       g.fillRect( col*colWidth, row*rowHeight,
                                      colWidth, rowHeight );
   }
}
```

Sometimes two-dimensional arrays are used in problems in which the grid is not so visually obvious. Consider a company that owns 25 stores. Suppose that the company has data about the profit earned at each store for each month in the year 2010. If the stores are numbered from 0 to 24, and if the twelve months from January '10 through December '10 are numbered from 0 to 11, then the profit data could be stored in an array, profit, constructed as follows:

```
double[][]  profit  =  new double[25][12];
```

profit[3][2] would be the amount of profit earned at store number 3 in March, and more generally, profit[storeNum][monthNum] would be the amount of profit earned in store number storeNum in month number monthNum. In this example, the one-dimensional array profit[storeNum] has a very useful meaning: It is just the profit data for one particular store for all the months in the whole year.

Let's assume that the profit array has already been filled with data. This data can be processed in a lot of interesting ways. For example, the total profit for the company -- for the whole year from all its stores -- can be calculated by adding up all the entries in the array:

```
double totalProfit;  // Company's total profit in 2010.

totalProfit = 0;
```

```
for (int store = 0; store < 25; store++) {
   for (int month = 0; month < 12; month++)
      totalProfit += profit[store][month];
}
```

Sometimes it is necessary to process a single row or a single column of an array, not the entire array. For example, to compute the total profit earned by the company in December, that is, in month number 11, you could use the loop:

```
double decemberProfit = 0.0;
for (storeNum = 0; storeNum < 25; storeNum++)
   decemberProfit += profit[storeNum][11];
```

Let's extend this idea to create a one-dimensional array that contains the total profit for each month of the year:

```
double[] monthlyProfit;  // Holds profit for each month.
monthlyProfit = new double[12];

for (int month = 0; month < 12; month++) {
   // compute the total profit from all stores in this month.
   monthlyProfit[month] = 0.0;
   for (int store = 0; store < 25; store++) {
         // Add the profit from this store in this month
         // into the total profit figure for the month.
      monthlyProfit[month] += profit[store][month];
   }
}
```

As a final example of processing the profit array, suppose that we wanted to know which store generated the most profit over the course of the year. To do this, we have to add up the monthly profits for each store. In array terms, this means that we want to find the sum of each row in the array. As we do this, we need to keep track of which row produces the largest total.

```
double maxProfit; // Maximum profit earned by a store.
int bestStore;    // The number of the store with the
                  //   maximum profit.

double total;    // Total profit for one store.

// First compute the profit from store number 0.

total = 0.0;
for (month = 0;  month < 12; month++)
   total += profit[0][month];

bestStore = 0;      // Start by assuming that the best
maxProfit = total;  //     store is store number 0.

// Now, go through the other stores, and whenever we
// find one with a bigger profit than maxProfit, revise
```

```
// the assumptions about bestStore and maxProfit.

for (store = 1; store < 25; store++) {

    // Compute this store's profit for the year.

    total = 0.0;
    for (month = 0; month < 12; month++)
       total += profit[store][month];

    // Compare this store's profits with the highest
    // profit we have seen among the preceding stores.

    if (total > maxProfit) {
       maxProfit = total;    // Best profit seen so far!
       bestStore = store;    // It came from this store.
    }

} // end for

// At this point, maxProfit is the best profit of any
// of the 25 stores, and bestStore is a store that
// generated that profit.  (Note that there could also be
// other stores that generated exactly the same profit.)
```

Source : http://math.hws.edu/javanotes/c7/s5.html