Two-dimensional arrays

The arrays you have been using so far have only held one column of data. But you can set up an array to hold more than one column. These are called two-dimensional (2d) arrays. As an example, think of a spreadsheet with rows and columns. If you have 6 rows and 5 columns then your spreadsheet can hold 30 numbers.

Contents

Simple view on 2d arrays

To use a 2d array in programs, like usual with arrays and objects we must separate *declarations* that prepare names from *definitions* that reserve space in memory. The following constructs are used for 2d arrays:

2d array declaration:

Syntax:

<type> `[][]` <name> ;
(A type, followed by two pairs of brackets, followed by an identifier, followed by a semicolon.)

Semantics:

Prepare the name on the right so that it can later refer to a 2d array where each element is of the type indicated on the left.

2d array definition:

Syntax:

`new` <type> `[` <height> `][` <width> `]`

Semantics:

Reserve space for a 2d array with the indicated number of rows and columns.

For example:

```
int[][] a;
a = new int[6][5];
```

This prepares the name "`a`" then binds it to a newly allocated 2d array of 6 rows and 5 columns, with each element of type `int`.

Like usual, you can combine declaration and definition as follows:

Combined 2d array declaration and definition

Syntax:

<type> `[][]` <name> `=` `new` <type> `[` <height> `][` <width> `] ;`

Semantics:

Combines the corresponding declaration and definition.

Once you have a 2d array you can use them in program using two pairs of brackets to access individual elements, like this:

2d element access

Syntax:

<expression> `[` <rowindex> `][` <columnindex> `]`

Semantics:

Evaluates the expression on the left, which must give a 2d array; then access the element indicated by the row index and column index in the 2d array.

For example:

```
void printDiagonal(int[][] array)
{
   int i;
   for (i = 0; i < array.length; i++)
       System.out.println(array[i][i]);
}
```

Note

If "`array`" is a name that refers to a 2d array, "`array.length`" gives the number of rows in the array. To access the number of columns, use "`array[0].length`" (length of the 1st row).

By using only the definitions in this first section, you can pass the course and use 2d arrays successfully in your own programs.

Advanced views on 2d arrays

If we look at the Java language definition in more details, we can see that there is no such thing as a real "2d array". Instead the notion of a "2d array" in Java is built entirely using the more simple single-dimensional array and the difference between name and definition already seen previously. The various constructs described in the section above are actually simplified abbreviations for compound uses of single-dimensional arrays.

To understand this fully, it is necessary to reveal a dark, sad and unfortunate corner of the Java language, that was inherited from C.

Left and right parts in a type

In Java, like in C before it, the syntactic form of a type must be separated into a left part and a right part.

All primitive and class types have only a left part. That is, the syntactic form of the type "`int`" has a left part "`int`" and an empty right part.

All array types, in contrast, have both a left and a right part. The left part is the element type. The right part is the array-ness or number of elements. For example, the syntactic form of the type "`int[]`" has a left part "`int`" and a right part "`[]`".

Left and right parts in a declaration

The separation in left and right parts are important to understand the real ("secret") form of a declaration in Java:

Array declaration (for realzzz)

Syntax:

> <leftpart> `[]` <rightpart> <name> ;
> (The left part of a type, followed by `[]`, followed by the right part, followed by a name, followed by a semicolon.)

Semantics:

> Prepares a name that can later refer to a 1d array where the element type is formed by placing the left part and right part together.

For example, in:

```
int[] a;
```

The left part is "`int`", and the right part is empty. So "`a`" refers to a 1d array where the element type is of type "`int`".

In contrast, consider:

```
double[][] b;
```

The left part is "`double`" and the right part is "`[]`". So "`b`" refers to a 1d array where the element type is of type "`double[]`".

Left and right parts in a definition

Likewise, the separation in left and right parts is necessary to understand the real form of an array definition:

Array definition (for realzzz):

Syntax:

> `new` <leftpart> `[` <size> `]` <rightpart>
> (The keyword "`new`" followed by the left part of a type, followed by a size between brackets, followed by the right part of a type.)

Semantics:

> Reserves spaces for a 1d array with the number of element indicated between the brackets, where each element is of the type defined by taking the left and right part together.
> If the element type is an array type, the definition only *prepares the names* for the individual element arrays, and does not actually reserve the space.

For example:

```
new int[3]
```

The left part is "`int`", the right part is empty, so an array of 3 elements of type "`int`" is defined.

Another example:

```
new int[3][]
```

The left part is "`int`", the right part is "`[]`", so an array of 3 elements of type "`int[]`" is defined. The 3 elements are merely *names*.

It is important to understand that a definition with `new xxx[n][]` only prepares names for the rows, so the following code produces an error:

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

System.out.println(a[0][1]);
```

The error is that `a[0]` is a name which has not yet been bound to an actual array, so `a[0][1]` cannot access a valid element.

Putting things together

Using the constructs from the previous sections it becomes possible to write the following:

```
double[][] r;
r = new double[6][];
r[0] = new double[5];
r[1] = new double[5];
r[2] = new double[5];
r[3] = new double[5];
r[4] = new double[5];
r[5] = new double[5];
r[2][3] = 42;
```

The first line declares a name "`r`" that can refer to an 1d array where every element is a name for an 1d array where every element is a `double`.

The second line defines an array of 6 elements, where each element is a name that can refer to an array of `double`.

The 3rd line defines an array of 5 elements of type `double`, then sets the name in 1st position of `r` to that new array.

The 3th line defines another array of 5 elements, and binds the name in 2nd position of `r` to it.

And so on, until the last line. This accesses first `r[2]`, which gives a name for an array of double; then it accesses the item at position 3 (4th item from the beginning) and changes it to 42.

Abbrevation for multiple definitions

The example above uses a construct which is quite common:

```
type[][] r = new type[n][];
r[0] = new double[m];
r[1] = new double[m];
...
r[n-1] = new double[m];
```

As usual, when a construct is commonly found, the language designers provide a shorter form to simplify the code. In this case we use the following:

Multi-dimensional array definition:

Syntax:

`new` <leftpart> `[` <size1> `][` <size2> `]` <rightpart>

Semantics:

> First defines a new array with <size1> elements of type "<leftpart>[]<rightpart>" (they are all names). Then for every position in this first array, create a new array of <size2> elements of type "<leftpart><rightpart>", then bind the name in the first array to the newly created array.

For example:

```
int[][] r;
r = new int[3][5];
```

is equivalent to:

```
int[][] r;
r = new int[3][];
r[0] = new int[5];
r[1] = new int[5];
r[2] = new int[5];
```

This generalizes directly to more dimensions. For example:

```
int[][][] r;
r = new int[2][3][4];
```

is equivalent to:

```
int[][][] r;
r = new int[2][][];
r[0] = new int[3][4];
r[1] = new int[3][4];
```

which itself is equivalent to:

```
int[][][] r;
r = new int[2][][];
r[0] = new int[3][];
r[0][0] = new int[4];
r[0][1] = new int[4];
r[0][2] = new int[4];
r[1] = new int[3][];
r[1][0] = new int[4];
r[1][1] = new int[4];
r[1][2] = new int[4];
```

So as you can see, the combined construct eventually expands to only uses of single-dimensional (1d) arrays.

Triangular and other heterogeneous arrays

Since each element in an array of type `int[][]` is a *name* for an array of type `int[]`, there is nothing remaining that forces every name to refer to arrays of the same sizes.

Actually, it is quite common to exploit this mechanism to define *triangular arrays*:

```
double[][] r;
r = new double[4][];
r[0] = new double[1];
r[1] = new double[2];
r[2] = new double[3];
r[3] = new double[4];
```

This feature is commonly used in programs that perform linear algebra on symmetrical matrices, since this way only half of the space is needed in memory to store all the useful values.

In general, if a 1d array has an array type as element type, and different elements point to arrays of different sizes, the array is said to be an *heterogeneous multi-dimensional array*.

Important concepts

- 2d array declaration and definition (simple view)
- (optional) separation in left and right part of a type syntax;
- (optional) how 2d arrays are formed using 1d arrays;
- (optional) triangular and heterogeneous 2d arrays.

Further reading

- Programming in Java, section 3.8.5 (pp. 120-122) & section 7.5 (pp. 363-366)
- Absolute Java, section 6.4 (pp. 399-425)

Copyright and licensing