

# Arrays

## 1 Modeling Aggregates

As you've seen, you can get pretty far with "scalar" data. But many phenomena we wish to model computationally are aggregates, or collections, for example:

- scores on assignments in a class,
- word counts in a document, or
- pixel colors in a bitmap image.

Today we'll learn Java's most basic facility for modeling aggregates: arrays.

## 2 Arrays

Java Arrays ([JLS §10](#)):

- are objects,
- are dynamically allocated (e.g., with operator `new`), and
- have a fixed number of elements of the same type.

## 3 Creating Arrays

Consider the following *array creation expression* ([JLS §10.3](#)):

```
double[] scores = new double[5];
```

This declaration:

- allocates a 5-element array,

- the 5 in the example above can be any expression that is unary promotable to an `int` (JLS §5.6.1)
- stores the address of this new array in `scores`, and
- initializes each value to its default value (0 for numeric types, `false` for boolean types, and `null` for references, JLS §4.12.5).

## 4 Array Declarations

The preceding array definition

```
double[] scores = new double[5];
```

could be split into a declaration and initialization:

```
double[] scores;
scores = new double[5];
```

Also, you can put the `[]` on the type or the variable name when declaring an array. These two declarations are equivalent:

```
double[] scores;
double scores[];
```

Generally, it's better style to put the `[]` on the type.

## 5 Mixed Declarations

Note that you can mix array declarations with declarations of variables having the same element type. The declaration line:

```
double scores[], average;
```

creates

- an array of double reference named `scores`, and
- a double variable named `average`

What's the size of the `scores` array declared above?

## 6 Array Objects

After the definition:

```
double[] scores = new double[5];
```

`scores` points to an array object in memory that can be visualized as:

0	1	2	3	4
0.0	0.0	0.0	0.0	0.0

The *indexes* of `scores` range from 0 to 4. The size of arrays are stored in a public final instance variable named `length`

```
scores.length == 5;
```

What is the type and value of the expression above?

## 7 Accessing Array Elements

Array elements are accessed with an `int`-promotable expression enclosed in square brackets (`[]`)

```
double[] scores = new double[5];
scores[0] = 89;
scores[1] = 100;
scores[2] = 95.6;
scores[3] = 84.5;
scores[4] = 91;
scores[scores.length - 1] = 99.2;
```

Will this line compile? If so, what will happen at runtime?

```
scores[scores.length] = 100;
```

## 8 Initializing Arrays

You can provide initial values for (small) arrays

```
String[] validSuits = {"diamonds", "clubs", "hearts", "spades"};
```

- What is `validSuits.length`?
- What is `validSuits[1]`?

You can also use a loop to initialize the values of an array:

```
int[] squares = new int[5];
for (int i = 0; i < squares.length; ++i) {
    squares[i] = i*i;
}
```

What is `squares[4]`?

## 9 Traversing Arrays

Arrays and `for` statements go hand-in-hand:

```
double[] scores = new double[5];
for (int i = 0; i < 5; ++i) {
    System.out.printf("scores[%d] = %.2f\n", i, scores[i]);
}
```

You can also use the “enhanced” `for` loop:

```
for (double score: scores) {
    System.out.println(score);
}
```

Read the enhanced `for` loop as “for each element of the array ...”.

Why use `for-each` instead of traditional `for`? ...

## 10 Traditional `for` Versus `for-each`

In cases where you don’t need the index, use the enhanced `for` loop. Consider:

```
double sum = 0.0;
for (int i = 0; i < scores.length; ++i) {
    sum += scores[i];
}
```

In the code above, `scores.length` is used only for bounding the array traversal, and the index `i` is only used for sequential array access. Those are two things we can mess up. The enhanced `for` loop is cleaner:

```
double sum = 0.0;
for (double score: scores) {
    sum += score;
}
```

Also note how our naming conventions help to make the code clear. You can read the loop above as “for each score in scores”.

## 11 Array Initialization and Access Gotchas

Because arrays are allocated dynamically, this will compile:

```
double[] scores = new double[-5];
```

but will produce an error at run-time:

```
Exception in thread "main" java.lang.NegativeArraySizeException
    at ArrayBasics.main(ArrayBasics.java:4)
```

Also, array access expressions are evaluated and checked at run-time. So, in the same way that accessing an array with an index  $\geq$  the size of the array produces a run-time error, negative indexes like:

```
scores[-1] = 100;
```

produce:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
    at ArrayBasics.main(ArrayBasics.java:23)
```

## 12 Arrays as Method Parameters - main

We've already seen an array parameter:

```
public static void main(String[] args)
```

We can use this array just like we use any other array.

```
public class Shout {
    public static void main(String[] args) {
        for (String arg: args) {
            System.out.print(arg.toUpperCase() + " ");
        }
        System.out.println();
    }
}
```

See also [CourseAverage.java](#)

## 13 Variable Arity Parameters

- The *arity* of a method is its number of formal parameters.
- So far, all the methods we've written have fixed arity.
- The last parameter to a method may be a *variable arity parameter*, a.k.a. *var args* parameter ([JLS §8.4.1](#)), whose syntax is simply to add an ellipse ( . . . ) after the type name.
- The var args parameter is accessed as an array inside the method.

For example:

```
public static int max(int ... numbers) {
    int max = numbers[0];
    for (int i = 1; i < numbers.length; ++i) {
        if (numbers[i] > max) max = numbers[i];
    }
    return max;
}
```

```
}
```

## 14 Multidimensional Arrays

You can create arrays of any number of dimensions simply by adding additional square brackets for dimensions and sizes. For example:

```
char[][] grid;
```

The declaration statement above:

- Declares a 2-dimensional array of `char`.
- As with one-dimensional arrays, `char` is the base type.
- Each element of `grid`, which is indexed by two `int` expressions, is a `char` variable.

## 15 Initializing Multidimensional Arrays

Initialization of 2-dimensional arrays can be done with `new`:

```
grid = new char[10][10];
```

or with literal initialization syntax:

```
char[][] grid = {{' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}};
```

Notice that a 2-dimensional array is an array of 1-dimensional arrays (and a 3-dimensional array is an array of 2-dimensional arrays, and so on).

## 16 Visualizing Multidimensional Arrays

Our 2-dimensional `grid` array can be visualized as a 2-d grid of cells.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
grid[0]	' '	' '	' '	' '	' '	' '	' '	' '	' '	' '
grid[1]	' '	' '	' '	' '	' '	' '	' '	' '	' '	' '
grid[2]	' '	'*'	'*'	' '	' '	' '	' '	'*'	'*'	' '
grid[3]	' '	'*'	'**'	' '	' '	' '	' '	'*'	'*'	' '
grid[4]	' '	' '	' '	' '	'*'	'*'	' '	' '	' '	' '
grid[5]	' '	' '	' '	' '	'*'	'*'	' '	' '	' '	' '
grid[6]	' '	'*'	' '	' '	' '	' '	' '	' '	'*'	' '
grid[7]	' '	' '	'*'	' '	' '	' '	' '	'*'	' '	' '
grid[8]	' '	' '	' '	'*'	'*'	'*'	'*'	' '	' '	' '
grid[9]	' '	' '	' '	' '	' '	' '	' '	' '	' '	' '

And an individual cell can be accessed by supplying two indices:

```
grid[3][2] == '*' ; // true
```

## 17 Traversing Multidimensional Arrays

Traverse 2-dimensional array by nesting loops. The key to getting it right is to use the right lengths.

```
for (int row = 0; row < grid.length; ++row) {
    for (int col = 0; col < grid[row].length; ++col) {
        System.out.print(grid[row][col]);
    }
    System.out.println();
}
```

Note that the for loops above traverse the grid in row-major order. We can traverse the grid in column-major order by reversing the nesting of the for loops:

```
for (int col = 0; col < grid[0].length; ++col) {
    for (int row = 0; row < grid.length; ++row) {
        System.out.print(grid[row][col]);
    }
    System.out.println();
}
```

See [Smiley.java](#)

## 18 Closing Thoughts

- Arrays are our first “collection classes” (but are not Java `Collection` classes).
- Arrays are objects, so array objects are created with operator `new` and array variables can have the value `null`.
- Arrays have sugar to add convenience and make them syntactically similar to C’s arrays.