Introduction to arrays

Regardless of the programming language, we often distinguish between *scalar* data types, which can only hold one piece of data, and *structured* types, which can contain multiple data elements. *Records* or *objects* (again, in non-language-specific terms) are structured types containing multiple data elements, each one potentially of a distinct type, and with each element referenced by its name. But a more fundamental structured type is a *sequence*, containing multiple data elements of the same type, referenced by position or index. In Java (as in many other languages), the most basic—and most important—kind of sequence is an *array*.

Concepts

We can picture an array as a row of post office boxes, numbered consecutively and starting at 0. We can open any individual box and see what it contains; we can also open any individual box and put something inside it. In programming, we usually refer to these boxes as *elements*, and the box numbers as *indices* or positions.

Each box is the same size and shape; therefore, each box can contain the same type of content. Continuing the post office box analogy, imagine a row of very small, letter-sized boxes: We could put a letter in each one, but not a package.

The most basic arrays are defined to contain a simple, scalar value in each box. For example, here is a visual representation of an array (of length 5) of *integers* (numbers without a decimal portion).

index	***0***	***1***	***2***	***3***	***4***
value	2	7	1	8	2

Some important points to note:

- The length of the array (the number of elements) is 5; however, since we start counting at 0, the index position of the last element in the array is 4.
- In this case, and most others, the value in each element (box) is not necessarily the same as the index or position of the element (i.e. the box number).

Declaring arrays in Java

We can define an array like the one above in Java code quite simply—especially since we know ahead of time the values in all of the elements:

```java
int[] digitsOfE = {2, 7, 1, 8, 2};
```

- We've called the array `digitsOfE` here; as a rule, variables should be given meaningful names, helping anyone that reads your code later (including yourself) understand what those variables are used for.
- The array is declared with the type `int[]`. The square brackets after `int` indicate that this is an array, and each element of the array can hold a value of type `int`.
- Because low-level Java syntax is based on C, we could express this same declaration in terms that while not commonly used in Java, are quite common in C code, and are supported in Java. For example, instead of `int[] digitsOfE`, we could write `int digitsOfE[]`, and it would have the same meaning.

The above statement is a *declaration-with-assignment*: the variable `digitsOfE` is declared (on the left side of the `=` sign) and assigned a value (taken from the right side of `=`) in the same statement. In this form, where the element values are enclosed in braces, the expression in braces is called an *array initializer*. The Java compiler is able to infer a lot of things when we use this form, but it's not always possible to use it. Another form with the same result is:

```java
int[] digitsOfE;
digitsOfE = new int[]{2, 7, 1, 8, 2};
```

The Java compiler is still inferring the size of the array, based on the number of values in the *array initializer* (the curly braces and values they enclose); but in this case, we have to tell the compiler to allocate memory for the array, using `new int[]`.

Accessing elements in an array

The above code fragments show examples of declaring and assigning all values of an array at once. But what do we do if we want to access (either to read or write) a single element? For that, we must use square brackets enclosing index positions. In fact, we might assign all the values of the array in this fashion:

```java
int[] digitsOfE = new int[5];
digitsOfE[0] = 2;
digitsOfE[1] = 7;
digitsOfE[2] = 1;
digitsOfE[3] = 8;
digitsOfE[4] = 2;
// The contents of digitsOfE are now {2, 7, 1, 8, 2}.
```

Again, we start with a *declaration-with-assignment* statement: The declaration is on the left side of the `=`, and the value being assigned is on the right. But rather than assigning individual element values, we're simply allocating space for the array.

Here, we're using the square brackets for two different (though both array-related) purposes:

- To specify the size of the array—that is, the number of elements—we enclose that size in square brackets after the element type and the `new` keyword. This allocates space for the array, and—because the element type is `int` (a numeric primitive type)—sets all the element values to 0.

- In this fragment, each element is referenced by its index, one at a time, and a new value is assigned to that element. After all of those assignments are complete, the contents of the array are exactly as they were in the code fragments in the previous section.

Alternatively, we might use one of the approaches shown earlier to assign all the values at once, and then modify a single element value using a bracketed reference:

```java
int[] digitsOfE = {2, 7, 1, 8, 3};
digitsOfE[4] = 2;
// The contents of digitsOfE are now {2, 7, 1, 8, 2}.
```

We can also use brackets to read individual element values. For example the following fragment creates and initializes the values as before, and then prints the value of element 3, which is 8.

```java
int[] digitsOfE = {2, 7, 1, 8, 2};
System.out.println(digitsOfE[3]);
```

In many cases, we want to iterate over an array—i.e. perform some operation on every element in the array in turn. Thus, we often see arrays paired with the `for` statement. The following fragment uses a `for` loop to print the value of each element in an array.

```java
int[] digitsOfE = {2, 7, 1, 8, 2};
for (int i = 0; i < digitsOfE.length; i++) {
  System.out.println(digitsOfE[i]);
}
```

Note that the loop control variable `i` is only being used to count up from 0 to the last index in the array, and to read the element value at each position. A simpler way to accomplish the same thing is the *enhanced-for* loop, which doesn't use an explicit loop control variable as an index:

```java
int[] digitsOfE = {2, 7, 1, 8, 2};
for (int digit : digitsOfE) {
  System.out.println(digit);
}
```

Default element values

In one of the examples above, it was mentioned that immediately after allocation of the array, all of its element values were set to 0. In fact, we can generalize that statement as follows:

- An array declared with elements of any primitive numeric type will have the value `0` assigned to all of its elements upon allocation, unless an array initializer is used.

- An array declared with elements of the `boolean` primitive type will have the value `false` assigned to all of its elements upon allocation, unless an array initializer is used.

- An array declared with elements of any object type will have the value `null` (i.e. a reference to nothing, or a non-existent object) assigned to all of its elements upon allocation, unless an array initializer is used.

```
double[] prices = new double[4]; // Contains {0.0, 0.0, 0.0, 0.0}.
long[] distances = new long[3]; // Contains {0, 0, 0}.
boolean[] flags = new boolean[5]; // Contains {false, false, false, false, false}.
String[] names = new String[2]; // Contains {null, null}.
```

Points to remember

- The declaration of an array **never** includes a size. The size is specified in the allocation expression (the part starting with `new`), or is inferred from an array initializer used in *declaration-with-assignment*.

- An array allocation performed with `new` **always** includes a size.

- Java arrays are fixed-length: once space is allocated for an array, it can't be expanded or contracted.

- While a Java array can be declared to contain other primitive types besides `int`, as well as object types, arrays are *homogeneous*: all elements of any given array must be of the same type. (This is not as restrictive as it might seem, particularly when the elements are objects.)

- 2- (and higher) dimensional arrays are possible in Java. However, their use can be a little confusing at the start, since they are actually *arrays of arrays* (or *arrays of arrays of arrays*, for a 3-dimensional array, and so on). More descriptively, a 2-dimensional array in Java is an array of objects, where each element is itself a reference to an array.

- Every array is actually a special kind of object in Java; however, the methods that can be invoked directly on an array are limited. Instead, we usually resort to methods in the `Arrays` or `System` class to perform specialized operations on arrays.

- Though it's not a syntactical requirement, we recommend giving arrays plural names as a general practice.