10 Managing Collections with Arrays

[Close all sections] Only visible sections will be included when printing this document.

References: L&L 8.1–8.3, 8.6, 11.1, 11.2

And later: utility methods in `java.util.Arrays` class: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

▽ Arrays: your first data structure

So far in the unit we've looked at representing individual pieces of data as primitive values or Strings, but there are also times when we need to store a (potentially large) number of values of the same type. One way to do this is to use a "Collection" class (there are many alternatives), which we don't cover in this unit, but which you can learn about yourself later. The other way is to use an **array**. (In fact, most of the Collection classes actually use arrays internally.)

An array is a fixed-length, ordered (and so, indexed) list of data of one type. Each *element* of an array is located at a position between 0 (zero) and the length of the array minus 1, just like the characters of a String are indexed from 0. Below is a diagrammatic representation of an array of 10 `int`s.

Index	0	1	2	3	4	5	6	7	8	9
Value	23	6	90	19	27	1	0	59	2	5

Syntax for declaring an array

An array variable is declared by appending square brackets `[]` to the name of the data type you want it to hold:

- `int[] ages;` declares a reference to an array of `int`s, called `ages`
- `String[] names;` declares a reference to an array of `String`s, called `names`
- `double[] heights;` declares a reference to an array of `double`s, called `heights`

Allocating space for an array

Arrays are a special kind of object, so merely declaring an array variable doesn't allocate any space for the array. Space is allocated using the `new` keyword, using the following syntax:

```
new type[size];
```

So, given the declarations above, we can allocate space for 10 `int`s and 20 `String`s with:

- `ages = new int[10];`
- `names = new String[20];`

The initial value of each element is set to a default when the array is created, according to the following table

General type	Example(s)	Default value
numerical primitives, including characters	`int`, `double`, `char`	0 (or 0.0)
boolean	`boolean`	`false`
objects	`String`, `Color`	`null`

Tip: If you've decided how large to make an array at design time then make the size a constant, as in:

```
final int CAPACITY = 20;
//and then
String[] names = new String[CAPACITY];
```

Initialising an array with a list of values

If you know at design time that an array will have a fixed set of values, or should start with a pre-defined set of values (that may later change) then you can initialise it when you declare it:

```
type[] identifier = { list, of, values };
```

For example, we could declare an array of playing card suits with:

```
String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
```

Array literals

The array literal representation shown above can only used when initialising an array's value. Writing `{ "A", "B" "C" }` elsewhere in a program would *not* create a three-element array of Strings, and wouldn't even compile. If you *do* need to create an unnamed array, filled with data then prefix it with `new element_type[]`, so `new String[]{ "A", "B", "C" }` would be valid. However, the need for this is rare and usually restricted to quickly testing a piece of array-related code.

Accessing a single value in an array

Array elements are like any other individual variables. To use or modify a particular value in an array, use its index in square brackets after the array variable's name, as in:

- `ages[2]` refers to the value at the third position in the `ages` array, so
  - `ages[2] = 50;` *writes* the value 50 to that position; and
  - `int aValue = ages[2];` *reads* the value at position 2 and assigns it to the variable `aValue`
- `names[0]` refers to the first value in the `names` array, so
  - `names[0] = "Fred"` assigns the value `"Fred"` to the first position in the array `names`; and
  - `String someName = names[0];` assigns the value in the first position in `names` to `someName`

The index can be *any integer expression*, so integer literals like `0` and `2` are allowed, as are expressions like `10/2` and `i` or `i/2` (assuming `i` was declared as an `int` at some point before).

Discovering the length of an array later

Every array has a `length` property (a read-only `int` instance variable) that holds the value of its length. This means we can discover an array's length after space for it has been allocated elsewhere in the program.

- `ages.length` is 10 (given how space was allocated above)
- `names.length` is 20

Note: An array's `length` property is *not* a method (like the similarly-named property of the `String` class), so never put parentheses after it.

Trying to access an element at an index below 0 or the array's length and above results in a runtime error that will generally halt your program.

Arrays and objects are *references*

Passing arrays to methods

Array variables are like objects in that they are *references* to the actual array in memory. This means that when they are passed to a method as an argument the *reference* is copied into the parameter that the method will use internally. And that means that the method has direct access to the data in the array. The practical implication of this is that any changes to the elements of an array made by a method persist after the method has finished.

Arrays of objects

As described above, when space for an array is allocated the elements are set to a default value based on their type. Class types are set to `null`, meaning they refer to nothing at all. So,

```
String[] words = new String[10];
```

allocates space for 10 String references, but doesn't create them. The elements only become non-null when values are assigned, as in

```
words[0] = "alpha";
words[1] = "beta";
```

At this point the array is similar to this picture (only 'similar' because the values in positions 0 and 1 are actually *references* to the Strings "alpha" and "beta"):

Index	0	1	2	3	4	5	6	7	8	9
Value	"alpha"	"beta"	`null`	`null`	`null`	`null`	`null`	`null`	`null`	`null`

▽ Tracing code involving arrays

You can apply your existing code tracing skills to code that involves arrays. The main complication that arrays introduce is an increase in the number of variables to track. Here are two approaches you may like to try, depending on what you're most interested in tracking as the code executes.

1. **If the code appears to be reading the array only** then sketch the array as a series of boxes off to the side of the tracing table, with the indices sitting above, similar to the visualisations in these notes.
   For example, given the code:

```
1  int[] vals = { 1, 2, 4 };
2  int sum = 0;
3  int i = 0;
4
5  while (i < vals.length) {
6      sum += vals[i];
7      i++;
8  }
```

   You could construct the tracing table:

Line	sum	i
2	0	
3		0
6	1	
7		1
6	3	
7		2
6	7	
7		3

	0	*1*	*2*
vals	1	2	4

2. **If the code is modifying the array's contents** then incorporate it into the table by placing the arrays name across multiple columns representing the position indices.
   For example, given the code:

```
1  int[] vals = { 1, 2, 4 };
2  int i = 0;
3
4  while (i < vals.length) {
5      vals[i] = 2 * vals[i];
6      i++;
7  }
```

   You could construct the tracing table:

		vals		
Line	i	0	1	2
1		1	2	4
2	0			
5		2		
6	1			
5			4	
6	2			
5				8
6	3			

   And if you wish, at the end you can always sketch the final state of the array by reading down the columns:

	0	*1*	*2*
vals	2	4	8

▽ Working with arrays

If working on the entire array, write a method (it can be reused on different arrays of the same type). To work out what to write consider the following three things:

- **Data in:**
  - Always the array
  - Anything else required, e.g.,
    - number of filled positions
    - a value to search for
    - value to multiply each element by
- **Data out:** depends on the task, e.g.,
  - sum of elements
  - index of found value
  - maybe nothing
- **The Algorithm:** simplest is to traverse the array and do "something" with each element. This means a loop with a variable representing the current index to work with:
  - I: Set index to 0
  - T: index < length of array
  - B: Do "something" with element at that index
  - U: increment index

Working with partially-filled arrays

Arrays have a fixed length, but that doesn't mean that every element holds a useful value. Your program might, for instance, allow a user to add elements one at a time until they wish to stop (or the array is full). The key to working with partially-filled arrays is to have two variables, one for the array and an `int` representing how full it is, as in:

```
String[] words = new String[10];
int count = 0;
```

Then to add an item to the "end" of the array:

```
words[count] = "some String to add";
count = count + 1;
```

Methods for dealing with partially-filled arrays take an additional parameter: the number of filled positions. And if they modify the array's contents by adding or deleting an entry then it's common for them to return the new number of filled positions. Here are two examples that illustrate this common pattern:

```
/** Display the first count elements of the array data, one per line. */
public static void printArray(String[] data, int count) {
    for (int i = 0; i < count; i++) { //<-- note count instead of data.length
        System.out.println(data[i]);
    }
}

/**
 * Adds the given value to the array, currently of count elements,
 * and returns the new number of elements. Does not add anything if
 * the array is full.
 */
public static int add(String[] data, int count, String value) {
     if (count < data.length) { //<-- only add if space available
        data[count] = value; //<-- count will be index of next empty space
        count++;
    }
    return count; //<-- if added then count is 1 higher; if not then unchanged
}
```

To use `add()` you then call it with the value to append and update the current `count` with its result:

```
count = add(words, count, "Another");
```

▽ Arrays utilities in the Java Standard Library

The java.util package contains a utility class called `Arrays`, which you can import to you program with `import java.util.Arrays;`. It holds a large number of utility methods, which are accessed statically, meaning `Arrays.methodName`. Some of its useful functions are the following, which have been ordered approximately by how advanced they are:

Arrays method	Purpose
equals(*array1*, *array2*)	Compares two arrays for value equality (same length, with same values in same positions)
fill(*array*, *value*)	Fills an array with a given value. There are variants for all element types
toString(*array*)	Creates a String representation of the array that is suitable for debugging purposes, but not ideal for presentation to humans. For example, the array `{"a", "Bee", "sea"}` is represented as `"[a, Bee, sea]"`
sort(*array*)	Sorts the contents of the array into ascending order. Works with arrays of numbers, characters, Strings (and other Comparable object types).
binarySearch(*array*, *value*)	Efficiently searches a previously sorted array for the given value and returns its position if it's found and -1 if it's not present.
copyOf(*array*, *new length*)	Creates a copy of an array, possibly with a different length to the original. So it can be used to 'grow' an array by making a larger copy of the original and then referring to that copy.

Making use of the fact that arrays allow direct access

Arrays are also useful for directly accessing individual elements that are identified by their index in the array. For instance, we could write a die roll simulator that stores the frequency with which each face comes up. The face value can be the index in the array of that face's roll frequency. This is also illustrated in the 2D array example next.

▽ Advanced topics

Multidimensional arrays

Arrays can be multidimensional, similar to mathematical matrices or tables. Two-dimensional arrays are quite common and are declared with

```
type[][] identifier;
```

and allocated with

```
new type[rows][columns];
```

For instance, if we wanted to represent the number of different species of animals observed in different adjacent regions on a map, divided into 10 rows and 15 columns, we could declare and allocate the array with

```
int[][] speciesCounts = new int[10][15];
```

To refer to the value at a particular position, simply refer to it by its row and column, so the species count at row 1, column 4 would be `speciesCounts[1][4]`.

Processing the elements of a 2D array

To process every element of a 2D array, use a nested loop for the columns, as in:

```
public void display2D(int[][] m) {
    for (int row = 0; row < m.length; row++) {
        for (int col = 0; col < m[row].length; col++) {
            System.out.print(m[row][col] + " ");
        }
        System.out.println(); //adds the newline
    }
}
```

`m.length` and `m[row].length` can be replaced by constants such as `ROWS` and `COLS` if these are known to your program.

Ragged arrays

Tip: 'Ragged' is pronounced like ra-gid, not rag'd.

A 2D array in Java is really an array *of arrays*. This means that the rows of a 2D array can be instantiated with different lengths by first allocating space for the first dimension of the array (the rows) and later instantiating each row:

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

which creates an array similar to the following diagram:

row	
0	`null`
2	`null`
3	`null`

then individual rows could be allocated with the necessary sizes:

```
assignmentMarks[0] = new int [10];
assignmentMarks[1] = new int[4];
```

which creates a 2D array similar to this diagram:

row\col	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0						
2	`null`									

Processing the elements of a ragged array

The example `display2D()` method above will work with ragged arrays because the nested for-loop checks the length of the current row.

Java's "for-each" loop

In late 2004 Java introduced a "for-each" loop that provides a shorter way of expressing that you want to iterate over all the elements in an array (or a Collection). The syntax is as follows:

```
for (element_type value_identifier : array_name ) {
    statements
}
```

For example, we could print all the elements in an array of `int`s called `ages` with:

```
for (int anAge : ages) {
    System.out.println(anAge);
}
```

This construct is useful if you want to use every value in the array in some way, but it has some limitations:

- it will go through *every* element in the array, even if only the first few spaces hold sensible values;
- inside the loop there is no way of knowing which index is currently being processed; and
- there is no way to modify the current value.

If you *do* need to do any of those things then use a typical for-loop instead.

Into the future: Collections

In the future you'll find yourself using a mixture of arrays and Collection objects. Arrays can provide exceptionally fast access, but are tedious to work with if you don't know how much data needs to be stored ahead of time. Collections (which use arrays and other data structures internally) take care of "growing" arrays as needed (actually creating a completely new, larger array and copying all the elements across). One Collection type you'll use frequently is the `Vector`, from the `java.util` package. You do not need to learn about it now, but make a note to come back to it in the future.

If you're interested now, see https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html and https://docs.oracle.com/javase/tutorial/collections/intro/index.html.