Chapter 4. Arrays

This chapter documents arrays, a fundamental datatype in JavaScript and in most other programming languages. An *array* is an ordered collection of values. Each value is called an *element*, and each element has a numeric position in the array, known as its *index*. JavaScript arrays are *untyped*: an array element may be of any type, and different elements of the same array may be of different types. Array elements may even be objects or other arrays, which allows you to create complex data structures, such as arrays of objects and arrays of arrays. JavaScript arrays are *zero-based* and use 32-bit indexes: the index of the first element is 0, and the highest possible index is 4294967294 ($2^{32}-2$), for a maximum array size of 4,294,967,295 elements. JavaScript arrays are *dynamic*: they grow or shrink as needed and there is no need to declare a fixed size for the array when you create it or to reallocate it when the size changes. JavaScript arrays may be *sparse*: the elements need not have contiguous indexes and there may be gaps. Every JavaScript array has a `length` property. For nonsparse arrays, this property specifies the number of elements in the array. For sparse arrays, `length` is larger than the index of all elements.

JavaScript arrays are a specialized form of JavaScript object, and array indexes are really little more than property names that happen to be integers. We'll talk more about the specializations of arrays elsewhere in this chapter. Implementations typically optimize arrays so that access to numerically indexed array elements is generally significantly faster than access to regular object properties.

Arrays inherit properties from `Array.prototype`, which defines a rich set of array manipulation methods, covered in §4.8. Most of these methods are *generic*, which means that they work correctly not only for true arrays, but for any "array-like object." We'll discuss array-like objects in §4.9. Finally, JavaScript strings behave like arrays of characters, and we'll discuss this in §4.10.

ECMAScript 6 introduces a set of new array classes known collectively as "typed arrays". Unlike regular JavaScript arrays, typed arrays have a fixed length and a fixed numeric element type. They offer high performance and byte-level access to binary data.

4.1 Creating Arrays

There are several ways to create arrays. The subsections that follow explain how to create arrays with

- array literals
- the `...` spread operator on an iterable object
- the `Array()` constructor
- the `Array.of()` and `Array.from()` factory methods

4.1.1 Array Literals

By far the simplest way to create an array is with an array literal, which is simply a comma-separated list of array elements within square brackets. For example:

```
let empty = [];                 // An array with no elements
let primes = [2, 3, 5, 7, 11];  // An array with 5 numeric elements
let misc = [ 1.1, true, "a", ]; // 3 elements of various types + trail
```

The values in an array literal need not be constants; they may be arbitrary expressions:

```
let base = 1024;
let table = [base, base+1, base+2, base+3];
```

Array literals can contain object literals or other array literals:

```
let b = [[1,{x:1, y:2}], [2, {x:3, y:4}]];
```

If an array literal contains multiple commas in a row, with no value between, the array is sparse (see 7.3). Array elements for which values are omitted do not exist, but appear to be `undefined` if you query them:

```
let count = [1,,3]; // Elements at indexes 0 and 2. No element at inde
let undefs = [,,];  // An array with no elements but a length of 2
```

Array literal syntax allows an optional trailing comma, so `[,,]` has a length of 2, not 3.

4.1.2 The Spread Operator

In ECMAScript 6 and later you can use the "spread operator" `...` to include the elements of one array within an array literal:

```
let a = [1, 2, 3];
let b = [0, ...a, 4];  // b == [0, 1, 2, 3, 4]
```

The three dots "spread" the array `a` so that its elements become elements within the array literal that is being created. It is as if the `...a` was replaced by the elements of the array `a`, listed literally as part of the enclosing array literal. (Note that although we call these three dots a "spread operator" this is not a true operator because it can only be used in array literals and, as we'll see later in the book, function invocations.)

The spread operator is a convenient way to create a (shallow) copy an array:

```
let original = [1,2,3];
let copy = [...original];
copy[0] = 0;  // Modifying the copy does not change the original
original[0]   // => 1
```

The spread operator is not intended to just spread arrays. In fact, it works to on any iterable object. (*Iterable* objects are what the `for/of` loop iterates over; we'll see more about them in Chapter 7.) Strings are iterable, so you can use a spread operator to turn any string into an array of single-character strings:

```
let digits = [..."0123456789ABCDEF"];
// digits == ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','
```

Set objects are iterable, so an easy way to remove duplicate elements from an array is to convert the array to a set and then immediately convert the set back to an array using the spread operator:

```
let letters = [...'hello world'];
[...new Set(letters)]  // => ['h','e','l','o',' ','w','r','d']
```

4.1.3 The Array() Constructor

Another way to create an array is with the `Array()` constructor. You can invoke this constructor in three distinct ways:

- Call it with no arguments:

  ```
  let a = new Array();
  ```

  This method creates an empty array with no elements and is equivalent to the array literal `[]`.

- Call it with a single numeric argument, which specifies a length:

  ```
  let a = new Array(10);
  ```

  This technique creates an array with the specified length. This form of the `Array()` constructor can be used to preallocate an array when you know in advance how many elements will be required. Note that no values are stored in the array, and the array index properties "0", "1", and so on are not even defined for the array.

- Explicitly specify two or more array elements or a single non-numeric element for the array:

  ```
  let a = new Array(5, 4, 3, 2, 1, "testing, testing");
  ```

  In this form, the constructor arguments become the elements of the new array. Using an array literal is almost always simpler than this usage of the `Array()` constructor.

4.1.4 Array.of()

When the `Array()` constructor function is invoked with one numeric argument, it uses that argument as an array length. But when invoked with more than one numeric argument, it treats those arguments as elements for the array to be

created. This means that the `Array()` constructor cannot be used to create an array with a single numeric element.

In ES6, the `Array.of()` function addresses this problem: it is a factory method that creates and returns a new array, using its argument values (regardless of how many of them there are) as the array elements:

```
Array.of()          // => []; returns empty array with no arguments
Array.of(10)        // => [10]; can create arrays with a single numeric
Array.of(1,2,3)     // => [1, 2, 3]
```

With this introduction of `Array.of()` there should rarely be any reason to use the Array constructor directly.

4.1.5 Array.from()

`Array.from` is another array factory method introduced in ES6. It expect an iterable or array-like object as its first argument, and returns a new array that contains the elements of that object. With an iterable argument, `Array.from(iterable)` works like the spread operator `[...iterable]` does. It is also a simple way to make a copy of an array:

```
let copy = Array.from(original);
```

`Array.from()` is also important because it defines a way to make a true-array copy of an array-like object. Array-like objects are non-array objects that have a numeric length property and have values stored with properties whose names happen to be integers. When working with client side JavaScript, the return values of some web browser methods are array-like, and it can be easier to work with them if you first convert them to true arrays:

```
let truearray = Array.from(arraylike);
```

`Array.from()` also accepts an optional second argument. If you pass a function as the second argument, then as the new array is being built, each element from the source object will be passed to the function you specify, and the return value of the function will be stored in the array instead of the original value. (This is very much like the array `map()` method that will be introduced later in the chapter, but is is more efficient to perform the mapping while the array is being built than it is to build the array and then map it to another new array.)

4.2 Reading and Writing Array Elements

You access an element of an array using the `[]` operator. A reference to the array should appear to the left of the brackets. An arbitrary expression that has a non-negative integer value should be inside the brackets. You can use this syntax to both read and write the value of an element of an array. Thus, the following are all legal JavaScript statements:

```
let a = ["world"];      // Start with a one-element array
let value = a[0];       // Read element 0
a[1] = 3.14;            // Write element 1
i = 2;
a[i] = 3;               // Write element 2
a[i + 1] = "hello";     // Write element 3
a[a[i]] = a[0];         // Read elements 0 and 2, write element 3
```

What is special about arrays is that when you use property names that are non-negative integers less than 2^{32}-1, the array automatically maintains the value of the `length` property for you. Above, for example, we created an array `a` with a single element. We then assigned values at indexes 1, 2, and 3. The `length` property of the array changed as we did so:

```
a.length        // => 4
```

Remember that arrays are a specialized kind of object. The square brackets used to access array elements work just like the square brackets used to access object properties. JavaScript converts the numeric array index you specify to a string—the index `1` becomes the string `"1"`—then uses that string as a property name. There is nothing special about the conversion of the index from a number to a string: you can do that with regular objects, too:

```
let o = {};     // Create a plain object
o[1] = "one";   // Index it with an integer
o["1"]          // => "one"; numeric and string property names are the
```

It is helpful to clearly distinguish an *array index* from an *object property name*. All indexes are property names, but only property names that are integers between 0 and 2^{32}-2 are indexes. All arrays are objects, and you can create properties of any name on them. If you use properties that are array indexes, however, arrays have the special behavior of updating their `length` property as needed.

Note that you can index an array using numbers that are negative or that are not integers. When you do this, the number is converted to a string, and that string is used as the property name. Since the name is not a non-negative integer, it is treated as a regular object property, not an array index. Also, if you index an array with a string that happens to be a non-negative integer, it behaves as an

array index, not an object property. The same is true if you use a floating-point number that is the same as an integer:

```
a[-1.23] = true;  // This creates a property named "-1.23"
a["1000"] = 0;    // This the 1001st element of the array
a[1.000]          // Array index 1.  Same as a[1]
```

The fact that array indexes are simply a special type of object property name means that JavaScript arrays have no notion of an "out of bounds" error. When you try to query a nonexistent property of any object, you don't get an error, you simply get `undefined`. This is just as true for arrays as it is for objects:

```
let a = [true, false]; // This array has elements at indexes 0 and 1
a[2]                   // => undefined; no element at this index.
a[-1]                  // => undefined; no property with this name.
```

4.3 Sparse Arrays

A *sparse* array is one in which the elements do not have contiguous indexes starting at 0. Normally, the `length` property of an array specifies the number of elements in the array. If the array is sparse, the value of the `length` property is greater than the number of elements. Sparse arrays can be created with the `Array()` constructor or simply by assigning to an array index larger than the current array `length`.

```
a = new Array(5);   // No elements, but a.length is 5.
a = [];             // Create an array with no elements and length = 0
a[1000] = 0;        // Assignment adds one element but sets length to
```

We'll see later that you can also make an array sparse with the `delete` operator.

Arrays that are sufficiently sparse are typically implemented in a slower, more memory-efficient way than dense arrays are, and looking up elements in such an array will take about as much time as regular object property lookup.

Note that when you omit a value in an array literal (using repeated commas as in `[1,,3]`) the resulting array is sparse and the omitted elements simply do not exist:

```
let a1 = [,];           // This array has no elements and length 1
let a2 = [undefined];   // This array has one undefined element
0 in a1                 // => false: a1 has no element with index 0
0 in a2                 // => true: a2 has the undefined value at ind
```

Some older implementations (such as Firefox 3) incorrectly insert the undefined values in array literals with omitted values. In these implementations `[1,,3]` is the same as `[1,undefined,3]`.

Understanding sparse arrays is an important part of understanding the true nature of JavaScript arrays. In practice, however, most JavaScript arrays you will work with will not be sparse. And, if you do have to work with a sparse array, your code will probably treat it just as it would treat a nonsparse array with `undefined` elements.

4.4 Array Length

Every array has a `length` property, and it is this property that makes arrays different from regular JavaScript objects. For arrays that are dense (i.e., not sparse), the `length` property specifies the number of elements in the array. Its value is one more than the highest index in the array:

```
[].length             // => 0: the array has no elements
['a','b','c'].length  // => 3: highest index is 2, length is 3
```

When an array is sparse, the `length` property is greater than the number of elements, and all we can say about it is that `length` is guaranteed to be larger than the index of every element in the array. Or, put another way, an array (sparse or not) will never have an element whose index is greater than or equal to its `length`. In order to maintain this invariant, arrays have two special behaviors. The first was described above: if you assign a value to an array element whose index `i` is greater than or equal to the array's current `length`, the value of the `length` property is set to `i+1`.

The second special behavior that arrays implement in order to maintain the length invariant is that if you set the `length` property to a non-negative integer `n` smaller than its current value, any array elements whose index is greater than or equal to `n` are deleted from the array:

```
a = [1,2,3,4,5];      // Start with a 5-element array.
a.length = 3;         // a is now [1,2,3].
a.length = 0;         // Delete all elements.  a is [].
a.length = 5;         // Length is 5, but no elements, like new Array(5
```

You can also set the `length` property of an array to a value larger than its current value. Doing this does not actually add any new elements to the array, it simply creates a sparse area at the end of the array.

4.5 Adding and Deleting Array Elements

We've already seen the simplest way to add elements to an array: just assign values to new indexes:

```
a = []            // Start with an empty array.
a[0] = "zero";    // And add elements to it.
a[1] = "one";
```

You can also use the `push()` method to add one or more values to the end of an array:

```
a = [];               // Start with an empty array
a.push("zero")        // Add a value at the end.  a = ["zero"]
a.push("one", "two") // Add two more values.  a = ["zero", "one", "two
```

Pushing a value onto an array `a` is the same as assigning the value to `a[a.length]`. You can use the `unshift()` method (described in §4.8) to insert a value at the beginning of an array, shifting the existing array elements to higher indexes. The `pop()` method is the opposite of `push()`: it removes the last element of the array and returns it, reducing the length of an array by 1. Similarly, the `shift()` method removes and returns the first element of the array, reducing the length by 1 and shifting all elements down to an index one lower than their current index. See §4.8 for more on these methods.

You can delete array elements with the `delete` operator, just as you can delete object properties:

```
a = [1,2,3];
delete a[2];   // a now has no element at index 2
2 in a         // => false: no array index 2 is defined
a.length       // => 3: delete does not affect array length
```

Deleting an array element is similar to (but subtly different than) assigning `undefined` to that element. Note that using `delete` on an array element does not alter the `length` property and does not shift elements with higher indexes down to fill in the gap that is left by the deleted property. If you delete an element from an array, the array becomes sparse.

As we saw above, you can also remove elements from the end of an array simply by setting the `length` property to the new desired length.

Finally, `splice()` is the general-purpose method for inserting, deleting, or replacing array elements. It alters the `length` property and shifts array elements to higher or lower indexes as needed. See §4.8 for details.

4.6 Iterating Arrays

As of ECMAScript 6, the easiest way to loop through each of the elements of an array (or any iterable object) is with the `for/of` loop.

```
let letters = [..."Hello world"];  // An array of characters for testi
let string = ""
for(let letter of letters) {
    string += letter
}
string  // => "Hello world"; we reassembled the original text
```

The built-in array iterator that the `for/of` loop uses returns the elements of an array in ascending order. It has no special behavior for sparse arrays and simply returns `undefined` for any array elements that do not exist.

If you want to use a `for/of` loop for an array need to know the index of each array element, use the `entries()` method of the array, along with destructuring assignment, like this:

```
let everyother = ""
for(let [index, letter] of letters.entries()) {
    if (index % 2 == 0) everyother += letter;  // Letters at even inde
}
everyother  // => "HLowrd"
```

Another good way to iterate arrays is with `forEach()` ("forEach()"). This is not a new form of the `for` loop, but an array method that offers a functional approach to array iteration. You pass a function to the `forEach()` method of an array, and `forEach()` invokes your function once on each element of the array:

```
let uppercase = ""
letters.forEach(letter => {  // Note arrow function syntax here
    uppercase += letter.toUpperCase();
});
uppercase  // => "HELLO WORLD"
```

As you would expect, `forEach()` iterates the array in order and it actually passes the array index to your function as a second argument, which is occasionally useful. Unlike the `for/of` loop, the `forEach()` is aware of sparse arrays and does not invoke your function for elements that are not there.

You can also loop through the elements of an array with a good old-fashioned `for` loop, which is likely to be slightly faster than the alternatives described

above:

```
let vowels = ""
for(let i = 0; i < letters.length; i++) { // For each index in the arr
    let letter = letters[i];              // Get the element at that
    if (/[aeiou]/.test(letter)) {         // Use a regular expression
        vowels += letter;                 // If it is a vowel, rememb
    }
}
vowels  // => "eoo"
```

In nested loops, or other contexts where performance is critical, you may sometimes see this basic array iteration loop written so that the array length is only looked up once rather than on each iteration. Both of the following `for` loop forms are idiomatic, though not particularly common, and with modern JavaScript interpreters it is not at all clear that they have any performance impact:

```
// Save the array length into a local variable
for(let i = 0, len = letters.length; i < len; i++) {
  // Loop body remains the same
}

// Iterate backwards from the end of the array to the start
for(let i = letters.length-1; i >= 0; i--) {
  // Loop body remains the same
}
```

These examples assume that the array is dense and that all elements contain valid data. If this is not the case, you should test the array elements before using them. If you want to skip undefined and nonexistent elements, you might write:

```
for(let i = 0; i < a.length; i++) {
    if (a[i] === undefined) continue; // Skip undefined + nonexistent
    // Loop body here
}
```

Changing the `===` comparison in the code above to `==` will also skip `null` elements.

4.7 Multidimensional Arrays

JavaScript does not support true multidimensional arrays, but you can approximate them with arrays of arrays. To access a value in an array of arrays, simply use the `[]` operator twice. For example, suppose the variable `matrix` is an array of arrays of numbers. Every element in `matrix[x]` is an array of numbers. To access a particular number within this array, you would write `matrix[x][y]`. Here is a concrete example that uses a two-dimensional array as a multiplication table:

```
// Create a multidimensional array
let table = new Array(10);                // 10 rows of the table
for(let i = 0; i < table.length; i++)
    table[i] = new Array(10);             // Each row has 10 columns

// Initialize the array
for(let row = 0; row < table.length; row++) {
    for(col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}

// Use the multidimensional array to compute 5*7
table[5][7]  // => 35
```

4.8 Array Methods

The sections above have focused on basic JavaScript syntax for working with arrays. In general, though, it is the methods defined by the Array class that are the most powerful. The sections below document these methods. While reading about these methods, keep in mind that some of them modify the array they are called on and some of them leave the array unchanged. A number of the methods return an array: sometimes this is a new array and the original is unchanged. Other times, a method will modify the array in place and will also return a reference to the modified array.

Each of the sub-sections that follows covers a group of related array methods:

- iterator methods loop over the elements of an array, typically invoking a function that you specify on each of those elements.
- stack and queue methods add and remove array elements to and from the beginning and the end of an array.
- subarray methods are for extracting, deleting, inserting, filling, and copying contiguous regions of a larger array.
- searching and sorting methods are for locating elements within an array and for sorting the elements of an array.

The subsections below also cover the static methods of the Array class and a few miscellaneous methods for concatenating arrays and converting arrays to strings.

4.8.1 Array Iterator Methods

The methods described in this section iterate over arrays by passing array elements, in order, to a function you supply, and they provide convenient ways to iterate, map, filter, test and reduce arrays.

Before we explain the methods in detail, however, it is worth making some generalizations about them. First, all of these methods accept a function as their first argument and invoke that function once for each element (or some elements) of the array. If the array is sparse, the function you pass is not invoked for nonexistent elements. In most cases, the function you supply is invoked with three arguments: the value of the array element, the index of the array element, and the array itself. Often, you only need the first of these argument values and can ignore the second and third values. Most of these methods accept an optional second argument. If specified, the function is invoked as if it is a method of this second argument. That is, the second argument you pass becomes the value of the `this` keyword inside of the function you pass as the first argument. The return value of the function you pass is usually important, but different methods handle the return value in different ways. None of the methods described here modify the array on which they are invoked (though the function you pass can modify the array, of course).

Each of these functions is invoked with a function as its first argument, and it is very common to define that function inline as part of the method invocation expression instead of using an existing function that is defined elsewhere. Arrow function syntax (see §5.1.3) works particularly well with these methods, and we will use it in the examples below.

FOREACH()

The `forEach()` method iterates through an array, invoking a function you specify for each element. As described above, you pass the function as the first argument to `forEach()`. `forEach()` then invokes your function with three arguments: the value of the array element, the index of the array element, and the array itself. If you only care about the value of the array element, you can write a function with only one parameter—the additional arguments will be ignored:

```
let data = [1,2,3,4,5], sum = 0;
// Compute the sum of the elements of the array
data.forEach(value => { sum += value; });           // sum == 15

// Now increment each array element
data.forEach(function(v, i, a) { a[i] = v + 1; }); // data == [2,3,4,
```

Note that `forEach()` does not provide a way to terminate iteration before all elements have been passed to the function. That is, there is no equivalent of the `break` statement you can use with a regular `for` loop.

MAP()

The `map()` method passes each element of the array on which it is invoked to the function you specify, and returns an array containing the values returned by your function. For example:

```
let a = [1, 2, 3];
a.map(x => x*x)     // => [1, 4, 9]
```

The function you pass to `map()` is invoked in the same way as a function passed to `forEach()`. For the `map()` method, however, the function you pass should return a value. Note that `map()` returns a new array: it does not modify the array it is invoked on. If that array is sparse, your function will not be called for the missing elements, but the returned array will be sparse in the same way as the original array: it will have the same length and the same missing elements.

FILTER()

The `filter()` method returns an array containing a subset of the elements of the array on which it is invoked. The function you pass to it should be predicate: a function that returns `true` or `false`. The predicate is invoked just as for `forEach()` and `map()`. If the return value is `true`, or a value that converts to `true`, then the element passed to the predicate is a member of the subset and is added to the array that will become the return value. Examples:

```
a = [5, 4, 3, 2, 1];
a.filter(x => x < 3)        // => [2, 1]; values less than 3
a.filter((x,i) => i%2 == 0) // => [5, 3, 1]; every other value
```

Note that `filter()` skips missing elements in sparse arrays, and that its return value is always dense. To close the gaps in a sparse array, you can do this:

```
let dense = sparse.filter(() => true);
```

And to close gaps and remove undefined and null elements you can use `filter` like this:

```
a = a.filter(x => x !== undefined && x !== null);
```

FIND() AND FINDINDEX()

The `find()` and `findIndex()` methods are like `filter()` in that they iterate through your array looking for elements for which your predicate function

returns a truthy value. Unlike `filter()`, however, these two methods stop iterating the first time the predicate finds an element. When that happens, `find()` returns the matching element and `findIndex()` returns the index of the matching element. If no matching element is found, `find()` returns `undefined` and `findIndex()` returns `-1`:

```
a = [1,2,3,4,5];
a.findIndex(x => x == 3);    // => 2; the value 3 appears at index 2
a.findIndex(x => x < 0);     // => -1; no negative numbers in the array
a.find(x => x % 5 == 0);     // => 5: this is a multiple of 5
a.find(x => x % 7 == 0);     // => undefined: no multiples of 7 in the ar
```

EVERY() AND SOME()

The `every()` and `some()` methods are array predicates: they apply a predicate function you specify to the elements of the array, and then return `true` or `false`.

The `every()` method is like the mathematical "for all" quantifier ∀: it returns `true` if and only if your predicate function returns `true` for all elements in the array:

```
a = [1,2,3,4,5];
a.every(x => x < 10)       // => true: all values are < 10.
a.every(x => x % 2 === 0) // => false: not all values are even.
```

The `some()` method is like the mathematical "there exists" quantifier ∃: it returns `true` if there exists at least one element in the array for which the predicate returns `true`, and returns `false` if and only if the predicate returns `false` for all elements of the array:

```
a = [1,2,3,4,5];
a.some(x => x%2===0)  // => true; a has some even numbers.
a.some(isNaN)         // => false; a has no non-numbers.
```

Note that both `every()` and `some()` stop iterating array elements as soon as they know what value to return. `some()` returns `true` the first time your predicate returns true, and only iterates through the entire array if your predicate always returns `false`. `every()` is the opposite: it returns `false` the first time your predicate returns `false`, and only iterates all elements if your predicate always returns `true`. Note also that by mathematical convention, `every()` returns `true` and `some` returns `false` when invoked on an empty array.

REDUCE() AND REDUCERIGHT()

The `reduce()` and `reduceRight()` methods combine the elements of an array, using the function you specify, to produce a single value. This is a common operation in functional programming and also goes by the names "inject" and "fold." Examples help illustrate how it works:

```
let a = [1,2,3,4,5]
a.reduce((x,y) => x+y, 0)          // => 15; the sum of the values
a.reduce((x,y) => x*y, 1)          // => 120; the product of the valu
a.reduce((x,y) => (x > y) ? x : y) // => 5; the largest of the values
```

`reduce()` takes two arguments. The first is the function that performs the reduction operation. The task of this reduction function is to somehow combine or reduce two values into a single value, and to return that reduced value. In the examples above, the functions combine two values by adding them, multiplying them, and choosing the largest. The second (optional) argument is an initial value to pass to the function.

Functions used with `reduce()` are different than the functions used with `forEach()` and `map()`. The familiar value, index, and array values are passed as the second, third, and fourth arguments. The first argument is the accumulated result of the reduction so far. On the first call to the function, this first argument is the initial value you passed as the second argument to `reduce()`. On subsequent calls, it is the value returned by the previous invocation of the function. In the first example above, the reduction function is first called with arguments 0 and 1. It adds these and returns 1. It is then called again with arguments 1 and 2 and it returns 3. Next it computes 3+3=6, then 6+4=10, and finally 10+5=15. This final value, 15, becomes the return value of `reduce()`.

You may have noticed that the third call to `reduce()` above has only a single argument: there is no initial value specified. When you invoke `reduce()` like this with no initial value, it uses the first element of the array as the initial value. This means that the first call to the reduction function will have the first and second array elements as its first and second arguments. In the sum and product examples above, we could have omitted the initial value argument.

Calling `reduce()` on an empty array with no initial value argument causes a TypeError. If you call it with only one value—either an array with one element and no initial value or an empty array and an initial value—it simply returns that one value without ever calling the reduction function.

`reduceRight()` works just like `reduce()`, except that it processes the array from highest index to lowest (right-to-left), rather than from lowest to highest. You might want to do this if the reduction operation has right-to-left associativity, for example:

```
// Compute 2^(3^4).  Exponentiation has right-to-left precedence
let a = [2, 3, 4]
a.reduceRight((acc,val) => Math.pow(val,acc)) // => 2.4178516392292583
```

Note that neither `reduce()` nor `reduceRight()` accepts an optional argument that specifies the `this` value on which the reduction function is to be invoked. The optional initial value argument takes its place. See the `Function.bind()` method if you need your reduction function invoked as a method of a particular object.

The examples shown so far have been numeric for simplicity, but `reduce()` and `reduceRight()` are not intended solely for mathematical computations. Any function that can combine two values (such as two objects) into one value of the same type can be used as a reduction function. On the other hand, algorithms expressed using array reductions can quickly become complex and hard to understand, and you may find that it is easier to read, write, and reason about your code if you use regular looping constructs to process your arrays.

4.8.2 Adding arrays with concat()

The `concat()` method creates and returns a new array that contains the elements of the original array on which `concat()` was invoked, followed by each of the arguments to `concat()`. If any of these arguments is itself an array, then it is the array elements that are concatenated, not the array itself. Note, however, that `concat()` does not recursively flatten arrays of arrays. `concat()` does not modify the array on which it is invoked:

```
let a = [1,2,3];
a.concat(4, 5)          // => [1,2,3,4,5]
a.concat([4,5],[6,7])   // => [1,2,3,4,5,6,7]; arrays are flattened
a.concat(4, [5,[6,7]])  // => [1,2,3,4,5,[6,7]]; but not nested arrays
a                       // => [1,2,3]; the original array is unmodifi
```

Note that `concat()` makes a new copy of the array it is called on. In many cases this is the right thing to do, but it is an expensive operation. If you find yourself writing code like `a = a.concat(x)` then you should think about modifying your array in place with `push()` or `splice()` instead of creating a new one.

4.8.3 Stacks and Queues with push(), pop(), shift() and unshift()

The `push()` and `pop()` methods allow you to work with arrays as if they were stacks. The `push()` method appends one or more new elements to the end of an array and returns the new length of the array. Unlike `concat()`, `push()` does not flatter array. The `pop()` method does the reverse: it deletes the last element of an array, decrements the array length, and returns the value that it removed. Note that both methods modify the array in place. The combination of `push()` and `pop()` allows you to use a JavaScript array to implement a first-in, last-out stack. For example:

```
let stack = [];       // stack == []
stack.push(1,2);      // stack == [1,2];
stack.pop();          // stack == [1]; returns 2
stack.push(3);        // stack == [1,3]
stack.pop();          // stack == [1]; returns 3
stack.push([4,5]);    // stack == [1,[4,5]]
stack.pop()           // stack == [1]; returns [4,5]
stack.pop();          // stack == []; returns 1
```

The `push()` method does not flatten an array you pass to it, but if you want to push all of the elements from one array onto another array, you can use the spread operator to flatten it explicitly:

```
a.push(...values)
```

The `unshift()` and `shift()` methods behave much like `push()` and `pop()`, except that they insert and remove elements from the beginning of an array rather than from the end. `unshift()` adds an element or elements to the beginning of the array, shifts the existing array elements up to higher indexes to make room, and returns the new length of the array. `shift()` removes and returns the first element of the array, shifting all subsequent elements down one place to occupy the newly vacant space at the start of the array. You could use `unshift()` and `shift()` to implement a stack but it would be less efficient than using `push()` and `pop()` because the array elements need to be shifted up or down every time an element is added or removed at the start of the array. Instead, though, you can implement a queue data structure by using `push()` to add elements at the end of an array and `shift()` to remove them from the start of the array:

```
let q = [];           // q == []
q.push(1,2);          // q == [1,2]
q.shift();            // q == [2]; returns 1
q.push(3)             // q == [2, 3]
q.shift()             // q == [3]; returns 2
q.shift()             // q == []; returns 3
```

There is one feature of `unshift()` that is worth calling out because you may find it surprising. When pass multiple arguments to unshift(), they are inserted all at once, which means that they end up in the array in a different order than they would be if you inserted them one at a time:

```
let a = [];             // a == []
a.unshift(1)            // a == [1]
a.unshift(2)            // a == [2, 1]
a = [];                 // a == []
a.unshift(1,2)          // a == [1, 2]
```

4.8.4 Subarrays with slice(), splice(), fill() and copyWithin()

Arrays define a number of methods that work on contiguous regions, or subarrays or "slices" of an array. The following sections describe method for extracting, replacing, filling and copying slices.

SLICE()

The `slice()` method returns a *slice*, or subarray, of the specified array. Its two arguments specify the start and end of the slice to be returned. The returned array contains the element specified by the first argument and all subsequent elements up to, but not including, the element specified by the second argument. If only one argument is specified, the returned array contains all elements from the start position to the end of the array. If either argument is negative, it specifies an array element relative to the last element in the array. An argument of -1, for example, specifies the last element in the array, and an argument of -3 specifies the third from last element of the array. Note that `slice()` does not modify the array on which it is invoked. Here are some examples:

```
let a = [1,2,3,4,5];
a.slice(0,3);     // Returns [1,2,3]
a.slice(3);       // Returns [4,5]
a.slice(1,-1);    // Returns [2,3,4]
a.slice(-3,-2);   // Returns [3]
```

SPLICE()

`splice()` is a general-purpose method for inserting or removing elements from an array. Unlike `slice()` and `concat()`, `splice()` modifies the array on which it is invoked. Note that `splice()` and `slice()` have very similar names but perform substantially different operations.

`splice()` can delete elements from an array, insert new elements into an array, or perform both operations at the same time. Elements of the array that come after the insertion or deletion point have their indexes increased or decreased as necessary so that they remain contiguous with the rest of the array. The first argument to `splice()` specifies the array position at which the insertion and/or deletion is to begin. The second argument specifies the number of elements that should be deleted from (spliced out of) the array. (Note that this is another difference between these two methods. The second argument to `slice()` is an end position. The second argument to `splice()` is a length.) If this second argument is omitted, all array elements from the start element to the end of the array are removed. `splice()` returns an array of the deleted elements, or an empty array if no elements were deleted. For example:

```
let a = [1,2,3,4,5,6,7,8];
a.splice(4)     // => [5,6,7,8]; a is now [1,2,3,4]
a.splice(1,2)   // => [2,3]; a is now [1,4]
a.splice(1,1)   // => [4]; a is now [1]
```

The first two arguments to `splice()` specify which array elements are to be deleted. These arguments may be followed by any number of additional arguments that specify elements to be inserted into the array, starting at the position specified by the first argument. For example:

```
let a = [1,2,3,4,5];
a.splice(2,0,'a','b')  // => []; a is now [1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3)  // => ['a','b']; a is now [1,2,[1,2],3,3,4,5]
```

Note that, unlike `concat()`, `splice()` inserts arrays themselves, not the elements of those arrays.

FILL()

The `fill()` method sets the elements of an array, or a slice of an array, to a specified value. It mutates the array it is called on, and also returns the modified array:

```
let a = new Array(5);   // Start with no elements and length 5
a.fill(0)               // => [0,0,0,0,0]; fill the array with zeros
a.fill(9, 1)            // => [0,9,9,9,9]; fill with 9 starting at in
a.fill(8, 2, -1)        // => [0,9,8,8,9]; fill with 8 at indexes 2, 3
```

The first argument to `fill()` is the value to set array elements to. The optional second argument specifies the starting index. If omitted, filling starts at index 0. The optional third argument specifies the ending index–array elements up to, but not including, this index will be filled. If this argument is omitted, then the array is filled from the start index to the end. You can specify indexes relative to the end of the array by passing negative numbers, just as you can for `slice()`.

COPYWITHIN()

`copyWithin()` copies a slice of an array to a new position within the array. It modifies the array in place, and returns the modified array, but it will not change the length of the array. The first argument specifies the destination index to which the first element will be copied. The second argument specifies the index of the first element to be copied. If this second argument is omitted, 0 is used. The third argument specifies the end of the slice of elements to be copied. If omitted, the length of the array is used. Elements from the start index up to, but not including the end index will be copied. You can specify indexes relative to the end of the array by passing negative numbers, just as you can for `slice()`.

```
let a = [1,2,3,4,5]
a.copyWithin(1)        // => [1,1,2,3,4]: copy array elements up one
a.copyWithin(2, 3, 5) // => [1,1,3,4,4]: copy last 2 elements to index
a.copyWithin(0, -2)    // => [4,4,3,4,4]: negative offsets work, too
```

`copyWithin()` is intended as a high-performance method that is particularly useful with typed arrays. It is modelled after the `memmove()` function from the C standard library. Note that the copy will work correctly even if there is overlap between the source and destination regions.

4.8.5 Array Searching and Sorting Methods

Arrays implement `indexOf()`, `lastIndexOf()` and `includes()` methods that are similar to the same-named methods of strings. There are also `sort()` and `reverse()` methods for reordering the elements of an array. These methods are described in the sub-sections below.

INDEXOF() AND LASTINDEXOF()

`indexOf()` and `lastIndexOf()` search an array for an element with a specified value, and return the index of the first such element found, or `-1` if none is found. `indexOf()` searches the array from beginning to end, and `lastIndexOf()` searches from end to beginning.

```
a = [0,1,2,1,0];
a.indexOf(1)       // => 1: a[1] is 1
a.lastIndexOf(1)   // => 3: a[3] is 1
a.indexOf(3)       // => -1: no element has value 3
```

These methods compare their argument to the array indexes using the equivalent of the `===` operator. If your array contains objects instead of primitive values will only do a shallow equality test to see if two references both refer to exactly the same object. If you want to actually look at the content of an object, try using the `find()` method with your own custom predicate function instead.

`indexOf()` and `lastIndexOf()` take an optional second argument that specifies the array index at which to begin the search. If this argument is omitted, `indexOf()` starts at the beginning and `lastIndexOf()` starts at the end. Negative values are allowed for the second argument and are treated as an offset from the end of the array, as they are for the `slice()` method: a value of –1, for example, specifies the last element of the array.

The following function searches an array for a specified value and returns an array of *all* matching indexes. This demonstrates how the second argument to `indexOf()` can be used to find matches beyond the first.

```
// Find all occurrences of a value x in an array a and return an array
// of matching indexes
function findall(a, x) {
    let results = [],             // The array of indexes we'll return
        len = a.length,           // The length of the array to be sea
        pos = 0;                  // The position to search from
    while(pos < len) {            // While more elements to search...
        pos = a.indexOf(x, pos); // Search
        if (pos === -1) break;    // If nothing found, we're done.
        results.push(pos);        // Otherwise, store index in array
        pos = pos + 1;            // And start next search at next ele
    }
    return results;               // Return array of indexes
}
```

Note that strings have `indexOf()` and `lastIndexOf()` methods that work like these array methods, except that a negative second argument is treated as zero.

INCLUDES()

The `includes()` method takes a single argument and returns `true` if the array contains that value or `false` otherwise. It does not tell you the index of the value, only whether it exists. The Set class also defines a similar `includes()` method, and this is effectively a set membership tests for arrays. (Note however that arrays are not an efficient representation for sets, and if you are working with more than a few elements you should use a real Set object.)

The `includes()` method is slightly different than the `indexOf()` method in one important way. `indexOf()` tests equality using the same algorithm that the `===` operator does, and that equality algorithm considers the Not-a-number value to be different from every other value but itself. `includes()` uses a slightly different version of equality that does consider NaN to be equal to itself. This means that `indexOf()` will not detect the NaN value in an array, but `includes()` will:

```
let a = [1,true,3,NaN];
a.includes(true)            // => true
a.includes(2)               // => false
a.includes(NaN)             // => true
a.indexOf(NaN)              // => -1; indexOf can't find NaN
```

SORT()

`sort()` sorts the elements of an array in place and returns the sorted array. When `sort()` is called with no arguments, it sorts the array elements in alphabetical order (temporarily converting them to strings to perform the comparison, if necessary):

```
let a = ["banana", "cherry", "apple"];
a.sort(); // a == ["apple", "banana", "cherry"]
```

If an array contains undefined elements, they are sorted to the end of the array.

To sort an array into some order other than alphabetical, you must pass a comparison function as an argument to `sort()`. This function decides which of its two arguments should appear first in the sorted array. If the first argument should appear before the second, the comparison function should return a number less than zero. If the first argument should appear after the second in the sorted array, the function should return a number greater than zero. And if the two values are equivalent (i.e., if their order is irrelevant), the comparison function should return 0. So, for example, to sort array elements into numerical rather than alphabetical order, you might do this:

```
let a = [33, 4, 1111, 222];
a.sort();                   // a == [1111, 222, 33, 4]; alphabetical or
a.sort(function(a,b) {      // Pass a comparator function
           return a-b;      // Returns < 0, 0, or > 0, depending on orde
       });                   // a == [4, 33, 222, 1111]; numerical order
a.sort((a,b) => b-a);       // a == [1111, 222, 33, 4]; reverse numerica
```

As another example of sorting array items, you might perform a case-insensitive alphabetical sort on an array of strings by passing a comparison function that converts both of its arguments to lowercase (with the `toLowerCase()` method) before comparing them:

```
a = ['ant', 'Bug', 'cat', 'Dog']
a.sort();    // a == ['Bug','Dog','ant','cat']; case-sensitive sort
a.sort(function(s,t) {
           let a = s.toLowerCase();
           let b = t.toLowerCase();
           if (a < b) return -1;
           if (a > b) return 1;
           return 0;
       });   // a == ['ant','Bug','cat','Dog']; case-insensitive sort
```

REVERSE()

The `reverse()` method reverses the order of the elements of an array and returns the reversed array. It does this in place; in other words, it doesn't create a new array with the elements rearranged but instead rearranges them in the already existing array:

```
let a = [1,2,3];
a.reverse();   // a == [3,2,1]
```

4.8.6 Array to String Conversions

The Array class defines three methods that can convert arrays to strings, which is generally something you might do when creating log and error messages. (If you want to save the contents of an array in textual form for later reuse, serialize the array with `JSON.stringify()` instead of using the methods described here.)

The `join()` method converts all the elements of an array to strings and concatenates them, returning the resulting string. You can specify an optional string that separates the elements in the resulting string. If no separator string is specified, a comma is used:

```
let a = [1, 2, 3];
a.join()                // => "1,2,3"
a.join(" ")             // => "1 2 3"
a.join("")              // => "123"
let b = new Array(10)   // An array of length 10 with no elements
b.join('-')             // => '---------': a string of 9 hyphens
```

The `join()` method is the inverse of the `String.split()` method, which creates an array by breaking a string into pieces.

Arrays, like all JavaScript objects, have a `toString()` method. For an array, this method works just like the `join()` method with no arguments:

```
[1,2,3].toString()          // => '1,2,3'
["a", "b", "c"].toString()  // => 'a,b,c'
[1, [2,'c']].toString()     // => '1,2,c'
```

Note that the output does not include square brackets or any other sort of delimiter around the array value.

`toLocaleString()` is the localized version of `toString()`. It converts each array element to a string by calling the `toLocaleString()` method of the element, and then it concatenates the resulting strings using a locale-specific (and implementation-defined) separator string.

4.8.7 Static Array Functions

In addition to the Array methods documented above, the Array class also defines three static functions that you can invoke through the `Array` constructor rather than on arrays. `Array.of()` and `Array.from()` are factory methods for creating new arrays. They were documented in §4.1.4 and §4.1.5.

The one other static Array function is `Array.isArray()` which is useful for determinging whether an unknown value is an array or not:

```
Array.isArray([])      // => true
Array.isArray({})      // => false
```

4.9 Array-Like Objects

As we've seen, JavaScript arrays have some special features that other objects do not have:

- The `length` property is automatically updated as new elements are added to the list.
- Setting `length` to a smaller value truncates the array.
- Arrays inherit useful methods from `Array.prototype`.
- Arrays have a *class* attribute of "Array".

These are the features that make JavaScript arrays distinct from regular objects. But they are not the essential features that define an array. It is often perfectly reasonable to treat any object with a numeric `length` property and corresponding non-negative integer properties as a kind of array.

These "array-like" objects actually do occasionally appear in practice, and although you cannot directly invoke array methods on them or expect special behavior from the `length` property, you can still iterate through them with the same code you'd use for a true array. It turns out that many array algorithms work just as well with array-like objects as they do with real arrays. This is especially true if your algorithms treat the array as read-only or if they at least leave the array length unchanged.

The following code takes a regular object, adds properties to make it an array-like object, and then iterates through the "elements" of the resulting pseudo-array:

```
let a = {};  // Start with a regular empty object

// Add properties to make it "array-like"
let i = 0;
while(i < 10) {
    a[i] = i * i;
    i++;
}
a.length = i;

// Now iterate through it as if it were a real array
let total = 0;
for(let j = 0; j < a.length; j++)
    total += a[j];
```

In client-side JavaScript, a number of DOM methods, such as `document.querySelectorAll()`, return array-like objects. Here's a function you might use to test for objects that work like arrays:

```
// Determine if o is an array-like object.
// Strings and functions have numeric length properties, but are
// excluded by the typeof test. In client-side JavaScript, DOM text
// nodes have a numeric length property, and may need to be excluded
// with an additional o.nodeType != 3 test.
function isArrayLike(o) {
    if (o &&                                // o is not null, undef
        typeof o === "object" &&            // o is an object
        isFinite(o.length) &&               // o.length is a finite
        o.length >= 0 &&                    // o.length is non-negat
        o.length===Math.floor(o.length) &&  // o.length is an integer
        o.length < 4294967295) {            // o.length < 2^32 - 1
        return true;                        // Then o is array-like
    } else {
        return false;                       // Otherwise it is not
    }
}
```

We'll see in §4.10 that strings behave like arrays. Nevertheless, tests like the one above for array-like objects typically return `false` for strings—they are usually best handled as strings, not as arrays.

Most JavaScript array methods are purposely defined to be generic, so that they work correctly when applied to array-like objects in addition to true arrays. Since array-like objects do not inherit from `Array.prototype`, you cannot invoke

array methods on them directly. You can invoke them indirectly using the `Function.call` method, however (see §5.7.4 for details):

```
let a = {"0":"a", "1":"b", "2":"c", length:3};      // An array-like ob
Array.prototype.join.call(a, "+")  // => "a+b+c"
Array.prototype.map.call(a, x => x.toUpperCase())  // => ["A","B","C"]
Array.prototype.slice.call(a, 0)    // => ["a","b","c"]: true array cop
```

The last line of code above invokes the Array `slice()` method on an array-like object in order to copy the elements of that object into a true array object. This is an idiomatic trick that exists in much legacy code, but is now more easy to do with `Array.from()`.

4.10 Strings As Arrays

Javascript strings behave like read-only arrays of UTF-16 Unicode characters. Instead of accessing individual characters with the `charAt()` method, you can use square brackets:

```
let s = "test";
s.charAt(0)     // => "t"
s[1]            // => "e"
```

The `typeof` operator still returns "string" for strings, of course, and the `Array.isArray()` method returns `false` if you pass it a string.

The primary benefit of indexable strings is simply that we can replace calls to `charAt()` with square brackets, which are more concise and readable, and potentially more efficient. The fact that strings behave like arrays also means, however, that we can apply generic array methods to them. For example:

```
Array.prototype.join.call("JavaScript", " ")  // => "J a v a S c r i p
```

Keep in mind that strings are immutable values, so when they are treated as arrays, they are read-only arrays. Array methods like `push()`, `sort()`, `reverse()`, and `splice()` modify an array in place and do not work on strings. Attempting to modify a string using an array method does not, however, cause an error: it simply fails silently.

4.11 Summary

This chapter has covered JavaScript arrays in depth, including esoteric details about sparse arrays and array-like objects. The main points to take from this chapter are:

- Array literals are written as comma-separated lists of values within square brackets.
- Individual array elements are accessed by specifying the desired array index within square brackets.
- The `for/of` loop and `...` spread operator introduced in ES6 are particularly useful ways to iterate arrays.
- The Array class defines a rich set of methods for manipulating arrays, and you should be sure to familiarize yourself with the Array API.