Indexed collections

This chapter introduces collections of data that are ordered by an index value. This includes **arrays** and **array-like constructs** such as `Array` objects and `TypedArray` objects.

For example, consider an array called `emp`, which contains employees' names indexed by their numerical employee number. So `emp[0]` would be employee number zero, `emp[1]` employee number one, and so on.

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods for manipulating arrays, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions.

We will focus on arrays in this article, but many of the same concepts apply to typed arrays as well, since arrays and typed arrays share many similar methods.

Creating an array

The following statements create equivalent arrays:

```
const arr1 = new Array(element0, element1, /* … ,*/ elementN);
const arr2 = Array(element0, element1, /* … ,*/ elementN);
const arr3 = [element0, element1, /* … ,*/ elementN];
```

`element0, element1, …, elementN` is a list of values for the array's elements. When these values are specified, the array is initialized with them as the array's elements. The array's `length` property is set to the number of arguments.

The bracket syntax is called an "array literal" or "array initializer", and is generally preferred.

To create an array with non-zero length, but without any items, either of the following can be used:

```
// This...
const arr1 = new Array(arrayLength);
```

```
// ...results in the same array as this
const arr2 = Array(arrayLength);

// This has exactly the same effect
const arr3 = [];
arr3.length = arrayLength;
```

Note: In the above code, `arrayLength` must be a `Number`. Otherwise, an array with a single element (the provided value) will be created. Calling `arr.length` will return `arrayLength`, but the array doesn't contain any elements. A `for...in` loop will not find any property on the array.

In addition to a newly defined variable as shown above, arrays can also be assigned as a property of a new or an existing object:

```
const obj = {};
// …
obj.prop = [element0, element1, /* … ,*/ elementN];

// OR
const obj = { prop: [element0, element1, /* … ,*/ elementN] };
```

If you wish to initialize an array with a single element, and the element happens to be a `Number`, you must use the bracket syntax. When a single `Number` value is passed to the `Array()` constructor or function, it is interpreted as an `arrayLength`, not as a single element.

```
// This creates an array with only one element: the number 42.
const arr = [42];

// This creates an array with no elements and arr.length set to 42.
const arr = Array(42);
```

Calling `Array(N)` results in a `RangeError`, if `N` is a non-whole number whose fractional portion is non-zero. The following example illustrates this behavior.

```
const arr = Array(9.3); // RangeError: Invalid array length
```

If your code needs to create arrays with single elements of an arbitrary data type, it is safer to use array literals. Alternatively, create an empty array first before adding the single element to it.

You can also use the `Array.of` static method to create arrays with single element.

```
const wisenArray = Array.of(9.3); // wisenArray contains only one element 9.3
```

Referring to array elements

Because elements are also properties, you can access them using property accessors. Suppose you define the following array:

```
const myArray = ["Wind", "Rain", "Fire"];
```

You can refer to the first element of the array as `myArray[0]`, the second element of the array as `myArray[1]`, etc... The index of the elements begins with zero.

Note: You can also use property accessors to access other properties of the array, like with an object.

```
const arr = ["one", "two", "three"];
arr[2]; // three
arr["length"]; // 3
```

Populating an array

You can populate an array by assigning values to its elements. For example:

```
const emp = [];
emp[0] = "Casey Jones";
emp[1] = "Phil Lesh";
emp[2] = "August West";
```

If you supply a non-integer value to the array operator in the code above, a property will be created in the object representing the array, instead of an array element.

```
const arr = [];
arr[3.4] = "Oranges";
console.log(arr.length); // 0
console.log(Object.hasOwn(arr, 3.4)); // true
```

You can also populate an array when you create it:

```
const myArray = new Array("Hello", myVar, 3.14159);
// OR
const myArray = ["Mango", "Apple", "Orange"];
```

Understanding length

At the implementation level, JavaScript's arrays actually store their elements as standard object properties, using the array index as the property name.

The `length` property is special. Its value is always a positive integer greater than the index of the last element if one exists. (In the example below, `'Dusty'` is indexed at `30`, so `cats.length` returns `30 + 1`).

Remember, JavaScript Array indexes are 0-based: they start at `0`, not `1`. This means that the **`length` property will be one more than the highest index stored in the array**:

```
const cats = [];
cats[30] = ["Dusty"];
console.log(cats.length); // 31
```

You can also assign to the `length` property.

Writing a value that is shorter than the number of stored items truncates the array. Writing `0` empties it entirely:

```
const cats = ["Dusty", "Misty", "Twiggy"];
console.log(cats.length); // 3

cats.length = 2;
console.log(cats); // [ 'Dusty', 'Misty' ] - Twiggy has been removed
```

```
cats.length = 0;
console.log(cats); // []; the cats array is empty

cats.length = 3;
console.log(cats); // [ <3 empty items> ]
```

Iterating over arrays

A common operation is to iterate over the values of an array, processing each one in some way. The simplest way to do this is as follows:

```
const colors = ["red", "green", "blue"];
for (let i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

If you know that none of the elements in your array evaluate to `false` in a boolean context—if your array consists only of DOM nodes, for example—you can use a more efficient idiom:

```
const divs = document.getElementsByTagName("div");
for (let i = 0, div; (div = divs[i]); i++) {
  /* Process div in some way */
}
```

This avoids the overhead of checking the length of the array, and ensures that the `div` variable is reassigned to the current item each time around the loop for added convenience.

The `forEach()` method provides another way of iterating over an array:

```
const colors = ["red", "green", "blue"];
colors.forEach((color) => console.log(color));
// red
// green
// blue
```

The function passed to `forEach` is executed once for every item in the array, with the array item passed as the argument to the function. Unassigned values are not iterated in a `forEach` loop.

Note that the elements of an array that are omitted when the array is defined are not listed when iterating by `forEach`, but are listed when `undefined` has been manually assigned to the element:

```
const sparseArray = ["first", "second", , "fourth"];

sparseArray.forEach((element) => {
  console.log(element);
});
// Logs:
// first
// second
// fourth

if (sparseArray[2] === undefined) {
  console.log("sparseArray[2] is undefined"); // true
}

const nonsparseArray = ["first", "second", undefined, "fourth"];

nonsparseArray.forEach((element) => {
  console.log(element);
});
// Logs:
// first
// second
// undefined
// fourth
```

Since JavaScript array elements are saved as standard object properties, it is not advisable to iterate through JavaScript arrays using `for...in` loops, because normal elements and all enumerable properties will be listed.

Array methods

The Array object has the following methods:

`concat()`

The `concat()` method joins two or more arrays and returns a new array.

```
let myArray = ["1", "2", "3"];
myArray = myArray.concat("a", "b", "c");
```

```
// myArray is now ["1", "2", "3", "a", "b", "c"]
```

join()

The `join()` method joins all elements of an array into a string.

```
const myArray = ["Wind", "Rain", "Fire"];
const list = myArray.join(" - "); // list is "Wind - Rain - Fire"
```

push()

The `push()` method adds one or more elements to the end of an array and returns the resulting `length` of the array.

```
const myArray = ["1", "2"];
myArray.push("3"); // myArray is now ["1", "2", "3"]
```

pop()

The `pop()` method removes the last element from an array and returns that element.

```
const myArray = ["1", "2", "3"];
const last = myArray.pop();
// myArray is now ["1", "2"], last = "3"
```

shift()

The `shift()` method removes the first element from an array and returns that element.

```
const myArray = ["1", "2", "3"];
const first = myArray.shift();
// myArray is now ["2", "3"], first is "1"
```

unshift()

The `unshift()` method adds one or more elements to the front of an array and returns the new length of the array.

```
const myArray = ["1", "2", "3"];
myArray.unshift("4", "5");
// myArray becomes ["4", "5", "1", "2", "3"]
```

slice()

The `slice()` method extracts a section of an array and returns a new array.

```
let myArray = ["a", "b", "c", "d", "e"];
myArray = myArray.slice(1, 4); // [ "b", "c", "d"]
// starts at index 1 and extracts all elements
// until index 3
```

splice()

The `splice()` method removes elements from an array and (optionally) replaces them. It returns the items which were removed from the array.

```
const myArray = ["1", "2", "3", "4", "5"];
myArray.splice(1, 3, "a", "b", "c", "d");
// myArray is now ["1", "a", "b", "c", "d", "5"]
// This code started at index one (or where the "2" was),
// removed 3 elements there, and then inserted all consecutive
// elements in its place.
```

at()

The `at()` method returns the element at the specified index in the array, or `undefined` if the index is out of range. It's notably used for negative indices that access elements from the end of the array.

```
const myArray = ["a", "b", "c", "d", "e"];
myArray.at(-2); // "d", the second-last element of myArray
```

`reverse()`

The `reverse()` method transposes the elements of an array, in place: the first array element becomes the last and the last becomes the first. It returns a reference to the array.

```
const myArray = ["1", "2", "3"];
myArray.reverse();
// transposes the array so that myArray = ["3", "2", "1"]
```

`flat()`

The `flat()` method returns a new array with all sub-array elements concatenated into it recursively up to the specified depth.

```
let myArray = [1, 2, [3, 4]];
myArray = myArray.flat();
// myArray is now [1, 2, 3, 4], since the [3, 4] subarray is flattened
```

`sort()`

The `sort()` method sorts the elements of an array in place, and returns a reference to the array.

```
const myArray = ["Wind", "Rain", "Fire"];
myArray.sort();
// sorts the array so that myArray = ["Fire", "Rain", "Wind"]
```

`sort()` can also take a callback function to determine how array elements are compared. The callback function is called with two arguments, which are two values from the array. The function compares these two values and returns a positive number, negative number, or zero, indicating the order of the two values. For instance, the following will sort the array by the last letter of a string:

```
const sortFn = (a, b) => {
  if (a[a.length - 1] < b[b.length - 1]) {
    return -1; // Negative number => a < b, a comes before b
  } else if (a[a.length - 1] > b[b.length - 1]) {
    return 1; // Positive number => a > b, a comes after b
```

```
  }
  return 0; // Zero => a = b, a and b keep their original order
};
myArray.sort(sortFn);
// sorts the array so that myArray = ["Wind","Fire","Rain"]
```

- if `a` is less than `b` by the sorting system, return `-1` (or any negative number)
- if `a` is greater than `b` by the sorting system, return `1` (or any positive number)
- if `a` and `b` are considered equivalent, return `0`.

`indexOf()`

The `indexOf()` method searches the array for `searchElement` and returns the index of the first match.

```
const a = ["a", "b", "a", "b", "a"];
console.log(a.indexOf("b")); // 1

// Now try again, starting from after the last match
console.log(a.indexOf("b", 2)); // 3
console.log(a.indexOf("z")); // -1, because 'z' was not found
```

`lastIndexOf()`

The `lastIndexOf()` method works like `indexOf`, but starts at the end and searches backwards.

```
const a = ["a", "b", "c", "d", "a", "b"];
console.log(a.lastIndexOf("b")); // 5

// Now try again, starting from before the last match
console.log(a.lastIndexOf("b", 4)); // 1
console.log(a.lastIndexOf("z")); // -1
```

`forEach()`

The `forEach()` method executes `callback` on every array item and returns `undefined`.

```
const a = ["a", "b", "c"];
```

```
a.forEach((element) => {
  console.log(element);
});
// Logs:
// a
// b
// c
```

The `forEach` method (and others below) that take a callback are known as *iterative methods*, because they iterate over the entire array in some fashion. Each one takes an optional second argument called `thisArg`. If provided, `thisArg` becomes the value of the `this` keyword inside the body of the callback function. If not provided, as with other cases where a function is invoked outside of an explicit object context, `this` will refer to the global object (`window`, `globalThis`, etc.) when the function is not strict, or `undefined` when the function is strict.

Note: The `sort()` method introduced above is not an iterative method, because its callback function is only used for comparison and may not be called in any particular order based on element order. `sort()` does not accept the `thisArg` parameter either.

`map()`

The `map()` method returns a new array of the return value from executing `callback` on every array item.

```
const a1 = ["a", "b", "c"];
const a2 = a1.map((item) => item.toUpperCase());
console.log(a2); // ['A', 'B', 'C']
```

`flatmap()`

The `flatmap()` method runs `map()` followed by a `flat()` of depth 1.

```
const a1 = ["a", "b", "c"];
const a2 = a1.flatMap((item) => [item.toUpperCase(), item.toLowerCase()]);
console.log(a2); // ['A', 'a', 'B', 'b', 'C', 'c']
```

`filter()`

The `filter()` method returns a new array containing the items for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const a2 = a1.filter((item) => typeof item === "number");
console.log(a2); // [10, 20, 30]
```

`find()`

The `find()` method returns the first item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.find((item) => typeof item === "number");
console.log(i); // 10
```

`findLast()`

The `findLast()` method returns the last item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.findLast((item) => typeof item === "number");
console.log(i); // 30
```

`findIndex()`

The `findIndex()` method returns the index of the first item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.findIndex((item) => typeof item === "number");
console.log(i); // 1
```

`findLastIndex()`

The `findLastIndex()` method returns the index of the last item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.findLastIndex((item) => typeof item === "number");
console.log(i); // 5
```

every()

The `every()` method returns `true` if `callback` returns `true` for every item in the array.

```
function isNumber(value) {
  return typeof value === "number";
}
const a1 = [1, 2, 3];
console.log(a1.every(isNumber)); // true
const a2 = [1, "2", 3];
console.log(a2.every(isNumber)); // false
```

some()

The `some()` method returns `true` if `callback` returns `true` for at least one item in the array.

```
function isNumber(value) {
  return typeof value === "number";
}
const a1 = [1, 2, 3];
console.log(a1.every(isNumber)); // true
const a2 = [1, "2", 3];
console.log(a2.every(isNumber)); // false
```

reduce()

The `reduce()` method applies `callback(accumulator, currentValue, currentIndex, array)` for each value in the array for the purpose of reducing the list of items down to a single value. The `reduce` function returns the final value returned by `callback` function.

If `initialValue` is specified, then `callback` is called with `initialValue` as the first parameter value and the value of the first item in the array as the second parameter value.

If `initialValue` is *not* specified, then `callback`'s first two parameter values will be the first and second elements of the array. On *every* subsequent call, the first parameter's value will be whatever `callback` returned on the previous call, and the second parameter's value will be the next value in the array.

If `callback` needs access to the index of the item being processed, or access to the entire array, they are available as optional parameters.

```
const a = [10, 20, 30];
const total = a.reduce(
  (accumulator, currentValue) => accumulator + currentValue,
  0,
);
console.log(total); // 60
```

`reduceRight()`

The `reduceRight()` method works like `reduce()`, but starts with the last element.

`reduce` and `reduceRight` are the least obvious of the iterative array methods. They should be used for algorithms that combine two values recursively in order to reduce a sequence down to a single value.

Sparse arrays

Arrays can contain "empty slots", which are not the same as slots filled with the value `undefined`. Empty slots can be created in one of the following ways:

```
// Array constructor:
const a = Array(5); // [ <5 empty items> ]

// Consecutive commas in array literal:
const b = [1, 2, , , 5]; // [ 1, 2, <2 empty items>, 5 ]

// Directly setting a slot with index greater than array.length:
const c = [1, 2];
c[4] = 5; // [ 1, 2, <2 empty items>, 5 ]

// Elongating an array by directly setting .length:
const d = [1, 2];
d.length = 5; // [ 1, 2, <3 empty items> ]

// Deleting an element:
const e = [1, 2, 3, 4, 5];
delete e[2]; // [ 1, 2, <1 empty item>, 4, 5 ]
```

In some operations, empty slots behave as if they are filled with `undefined`.

```
const arr = [1, 2, , , 5]; // Create a sparse array

// Indexed access
console.log(arr[2]); // undefined

// For...of
for (const i of arr) {
  console.log(i);
}
// Logs: 1 2 undefined undefined 5

// Spreading
const another = [...arr]; // "another" is [ 1, 2, undefined, undefined, 5 ]
```

But in others (most notably array iteration methods), empty slots are skipped.

```
const mapped = arr.map((i) => i + 1); // [ 2, 3, <2 empty items>, 6 ]
arr.forEach((i) => console.log(i)); // 1 2 5
const filtered = arr.filter(() => true); // [ 1, 2, 5 ]
const hasFalsy = arr.some((k) => !k); // false

// Property enumeration
const keys = Object.keys(arr); // [ '0', '1', '4' ]
for (const key in arr) {
  console.log(key);
}
// Logs: '0' '1' '4'
// Spreading into an object uses property enumeration, not the array's iterator
const objectSpread = { ...arr }; // { '0': 1, '1': 2, '4': 5 }
```

Multi-dimensional arrays

Arrays can be nested, meaning that an array can contain another array as an element. Using this characteristic of JavaScript arrays, multi-dimensional arrays can be created.

The following code creates a two-dimensional array.

```
const a = new Array(4);
for (let i = 0; i < 4; i++) {
```

```
    a[i] = new Array(4);
    for (let j = 0; j < 4; j++) {
      a[i][j] = `[${i}, ${j}]`;
    }
  }
```

This example creates an array with the following rows:

```
Row 0: [0, 0] [0, 1] [0, 2] [0, 3]
Row 1: [1, 0] [1, 1] [1, 2] [1, 3]
Row 2: [2, 0] [2, 1] [2, 2] [2, 3]
Row 3: [3, 0] [3, 1] [3, 2] [3, 3]
```

Using arrays to store other properties

Arrays can also be used like objects, to store related information.

```
const arr = [1, 2, 3];
arr.property = "value";
console.log(arr.property); // "value"
```

For example, when an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `RegExp.prototype.exec()`, `String.prototype.match()`, and `String.prototype.split()`. For information on using arrays with regular expressions, see Regular Expressions.

Working with array-like objects

Some JavaScript objects, such as the `NodeList` returned by `document.getElementsByTagName()` or the `arguments` object made available within the body of a function, look and behave like arrays on the surface but do not share all of their methods. The `arguments` object provides a `length` attribute but does not implement array methods like `forEach()`.

Array methods cannot be called directly on array-like objects.

```
function printArguments() {
  arguments.forEach((item) => {
    console.log(item);
  }); // TypeError: arguments.forEach is not a function
}
```

But you can call them indirectly using `Function.prototype.call()`.

```
function printArguments() {
  Array.prototype.forEach.call(arguments, (item) => {
    console.log(item);
  });
}
```

Array prototype methods can be used on strings as well, since they provide sequential access to their characters in a similar way to arrays:

```
Array.prototype.forEach.call("a string", (chr) => {
  console.log(chr);
});
```