Chapter 12: Arrays

Section 12.1: Converting Array-like Objects to Arrays

What are Array-like Objects?

JavaScript has "Array-like Objects", which are Object representations of Arrays with a length property. For example:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

Common examples of Array-like Objects are the `arguments` object in functions and `HTMLCollection` or `NodeList` objects returned from methods like `document.getElementsByTagName` or `document.querySelectorAll`.

However, one key difference between Arrays and Array-like Objects is that Array-like objects inherit from `Object.prototype` instead of `Array.prototype`. This means that Array-like Objects can't access common Array prototype methods like `forEach()`, `push()`, `map()`, `filter()`, and `slice()`:

```
var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // Error! indexOf is not defined.
domList.forEach(function() {
  arguments.map(/* Stuff here */) // Error! map is not defined.
}); // Error! forEach is not defined.

function func() {
  console.log(arguments);
}
func(1, 2, 3);   // → [1, 2, 3]
```

Convert Array-like Objects to Arrays in ES6

1. `Array.from`:

Version ≥ 6

```
const arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
arrayLike.forEach(value => {/* Do something */}); // Errors
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* Do something */}); // Works
```

2. `for...of`:

Version ≥ 6

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

3. Spread operator:

Version ≥ 6

```
[...arrayLike]
```

4. `Object.values`:

Version ≥ 7

```
var realArray = Object.values(arrayLike);
```

5. `Object.keys`:

Version ≥ 6

```
var realArray = Object
   .keys(arrayLike)
   .map((key) => arrayLike[key]);
```

Convert Array-like Objects to Arrays in ≤ ES5

Use `Array.prototype.slice` like so:

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // Shorter version

realArray.indexOf('Value 1'); // Wow! this works
```

You can also use `Function.prototype.call` to call `Array.prototype` methods on Array-like objects directly, without converting them:

Version ≥ 5.1

```
var domList = document.querySelectorAll('#myDropdown option');

domList.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

Array.prototype.forEach.call(domList, function() {
  // Do stuff
}); // Wow! this works
```

You can also use `[].method.bind( arrayLikeObject )` to borrow array methods and glom them on to your object:

Version ≥ 5.1

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};

arrayLike.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

[].forEach.bind(arrayLike)(function(val){
  // Do stuff with val
```

```
}); // Wow! this works
```

Modifying Items During Conversion

In ES6, while using `Array.from`, we can specify a map function that returns a mapped value for the new array being created.

Version ≥ 6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

See Arrays are Objects for a detailed analysis.

Section 12.2: Reducing values

Version ≥ 5.1

The `reduce()` method applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value.

Array Sum

This method can be used to condense all values of an array into a single value:

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});
// → 10
```

Optional second parameter can be passed to `reduce()`. Its value will be used as the first argument (specified as `a`) for the first call to the callback (specified as `function(a, b)`).

```
[2].reduce(function(a, b) {
  console.log(a, b); // prints: 1 2
  return a + b;
}, 1);
// → 3
```

Version ≥ 5.1

Flatten Array of Objects

The example below shows how to flatten an array of objects into a single object.

```
var array = [{
    key: 'one',
    value: 1
}, {
    key: 'two',
    value: 2
}, {
    key: 'three',
    value: 3
}];
```

Version ≤ 5.1

```
array.reduce(function(obj, current) {
  obj[current.key] = current.value;
  return obj;
}, {});
```

Version ≥ 6

```
array.reduce((obj, current) => Object.assign(obj, {
  [current.key]: current.value
}), {});
```

Version ≥ 7

```
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});
```

Note that the Rest/Spread Properties is not in the list of finished proposals of ES2016. It isn't supported by ES2016. But we can use babel plugin babel-plugin-transform-object-rest-spread to support it.

All of the above examples for Flatten Array result in:

```
{
  one: 1,
  two: 2,
  three: 3
}
```

Version ≥ 5.1

Map Using Reduce

As another example of using the *initial value* parameter, consider the task of calling a function on an array of items, returning the results in a new array. Since arrays are ordinary values and list concatenation is an ordinary function, we can use `reduce` to accumulate a list, as the following example demonstrates:

```
function map(list, fn) {
  return list.reduce(function(newList, item) {
    return newList.concat(fn(item));
  }, []);
}

// Usage:
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]
```

Note that this is for illustration (of the initial value parameter) only, use the native `map` for working with list transformations (see Mapping values for the details).

Version ≥ 5.1

Find Min or Max Value

We can use the accumulator to keep track of an array element as well. Here is an example leveraging this to find the min value:

```
var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10
```

Version ≥ 6

Find Unique Values

Here is an example that uses reduce to return the unique numbers to an array. An empty array is passed as the second argument and is referenced by `prev`.

```
var arr = [1, 2, 1, 5, 9, 5];

arr.reduce((prev, number) => {
  if(prev.indexOf(number) === -1) {
    prev.push(number);
  }
  return prev;
}, []);
// → [1, 2, 5, 9]
```

Section 12.3: Mapping values

It is often necessary to generate a new array based on the values of an existing array.

For example, to generate an array of string lengths from an array of strings:

Version ≥ 5.1

```
['one', 'two', 'three', 'four'].map(function(value, index, arr) {
  return value.length;
});
// → [3, 3, 5, 4]
```

Version ≥ 6

```
['one', 'two', 'three', 'four'].map(value => value.length);
// → [3, 3, 5, 4]
```

In this example, an anonymous function is provided to the `map()` function, and the map function will call it for every element in the array, providing the following parameters, in this order:

- The element itself
- The index of the element (0, 1...)
- The entire array

Additionally, `map()` provides an *optional* second parameter in order to set the value of **`this`** in the mapping function. Depending on the execution environment, the default value of **`this`** might vary:

In a browser, the default value of **`this`** is always `window`:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (the default value in browsers)
  return value.length;
});
```

You can change it to any custom object like this:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

Section 12.4: Filtering Object Arrays

The `filter()` method accepts a test function, and returns a new array containing only the elements of the original array that pass the test provided.

```
// Suppose we want to get all odd number in an array:
var numbers = [5, 32, 43, 4];
```

Version ≥ 5.1

```
var odd = numbers.filter(function(n) {
  return n % 2 !== 0;
});
```

Version ≥ 6

```
let odd = numbers.filter(n => n % 2 !== 0); // can be shortened to (n => n % 2)
```

`odd` would contain the following array: `[5, 43]`.

It also works on an array of objects:

```
var people = [{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}, {
  id: 3,
  name: "Peter",
  age: 55
}];
```

Version ≥ 5.1

```
var young = people.filter(function(person) {
  return person.age < 35;
});
```

Version ≥ 6

```
let young = people.filter(person => person.age < 35);
```

`young` would contain the following array:

```
[{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}]
```

You can search in the whole array for a value like this:

```
var young = people.filter((obj) => {
  var flag = false;
  Object.values(obj).forEach((val) => {
    if(String(val).indexOf("J") > -1) {
      flag = true;
      return;
    }
  });
  if(flag) return obj;
});
```

This returns:

```
[{
  id: 1,
  name: "John",
  age: 28
},{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Section 12.5: Sorting Arrays

The `.sort()` method sorts the elements of an array. The default method will sort the array according to string Unicode code points. To sort an array numerically the `.sort()` method needs to have a `compareFunction` passed to it.

> **Note:** The `.sort()` method is impure. `.sort()` will sort the array **in-place**, i.e., instead of creating a sorted copy of the original array, it will re-order the original array and return it.

Default Sort

Sorts the array in UNICODE order.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Results in:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

> **Note:** The uppercase characters have moved above lowercase. The array is not in alphabetical order, and numbers are not in numerical order.

Alphabetical Sort

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {
  return a.localeCompare(b);
});
```

Results in:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

> **Note:** The above sort will throw an error if any array items are not a string. If you know that the array may contain items that are not strings use the safe version below.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {
  return a.toString().localeCompare(b);
});
```

String sorting by length (longest first)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return b.length - a.length;
});
```

Results in

```
["elephants", "penguins", "zebras", "dogs"];
```

String sorting by length (shortest first)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return a.length - b.length;
});
```

Results in

```
["dogs", "zebras", "penguins", "elephants"];
```

Numerical Sort (ascending)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return a - b;
});
```

Results in:

```
[1, 10, 100, 1000, 10000]
```

Numerical Sort (descending)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return b - a;
});
```

Results in:

```
[10000, 1000, 100, 10, 1]
```

Sorting array by even and odd numbers

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {
    return (a & 1) - (b & 1) || a - b;
});
```

Results in:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Date Sort (descending)

```
var dates = [
  new Date(2007, 11, 10),
  new Date(2014, 2, 21),
```

```
  new Date(2009, 6, 11),
  new Date(2016, 7, 23)
];

dates.sort(function(a, b) {
  if (a > b) return -1;
  if (a < b) return 1;
  return 0;
});

// the date objects can also sort by its difference
// the same way that numbers array is sorting
dates.sort(function(a, b) {
  return b-a;
});
```

Results in:

```
[
  "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",
  "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",
  "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",
  "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"
]
```

Section 12.6: Iteration

A traditional `for`-loop

A traditional `for` loop has three components:

1. **The initialization:** executed before the look block is executed the first time
2. **The condition:** checks a condition every time before the loop block is executed, and quits the loop if false
3. **The afterthought:** performed every time after the loop block is executed

These three components are separated from each other by a `;` symbol. Content for each of these three components is optional, which means that the following is the most minimal `for` loop possible:

```
for (;;) {
    // Do stuff
}
```

Of course, you will need to include an `if(condition === true) { break; }` or an `if(condition === true) { return; }` somewhere inside that `for`-loop to get it to stop running.

Usually, though, the initialization is used to declare an index, the condition is used to compare that index with a minimum or maximum value, and the afterthought is used to increment the index:

```
for (var i = 0, length = 10; i < length; i++) {
    console.log(i);
}
```

Using a traditional `for` loop to loop through an array

The traditional way to loop through an array, is this:

```
for (var i = 0, length = myArray.length; i < length; i++) {
```

```
    console.log(myArray[i]);
}
```

Or, if you prefer to loop backwards, you do this:

```
for (var i = myArray.length - 1; i > -1; i--) {
    console.log(myArray[i]);
}
```

There are, however, many variations possible, like for example this one:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =
myArray[++key]) {
    console.log(value);
}
```

... or this one ...

```
var i = 0, length = myArray.length;
for (; i < length;) {
    console.log(myArray[i]);
    i++;
}
```

... or this one:

```
var key = 0, value;
for (; value = myArray[key++];){
    console.log(value);
}
```

Whichever works best is largely a matter of both personal taste and the specific use case you're implementing.

Note that each of these variations is supported by all browsers, including very very old ones!

A `while` loop

One alternative to a `for` loop is a `while` loop. To loop through an array, you could do this:

```
var key = 0;
while(value = myArray[key++]){
    console.log(value);
}
```

Like traditional `for` loops, `while` loops are supported by even the oldest of browsers.

Also, note that every while loop can be rewritten as a `for` loop. For example, the `while` loop hereabove behaves the exact same way as this `for`-loop:

```
for(var key = 0; value = myArray[key++];){
    console.log(value);
}
```

`for...in`

In JavaScript, you can also do this:

```
for (i in myArray) {
    console.log(myArray[i]);
}
```

This should be used with care, however, as it doesn't behave the same as a traditional `for` loop in all cases, and there are potential side-effects that need to be considered. See **Why is using "for...in" with array iteration a bad idea?** for more details.

`for...of`

In ES 6, the `for-of` loop is the recommended way of iterating over a the values of an array:

Version ≥ 6

```
let myArray = [1, 2, 3, 4];
for (let value of myArray) {
  let twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
}
```

The following example shows the difference between a `for...of` loop and a `for...in` loop:

Version ≥ 6

```
let myArray = [3, 5, 7];
myArray.foo = "hello";

for (var i in myArray) {
  console.log(i); // logs 0, 1, 2, "foo"
}

for (var i of myArray) {
  console.log(i); // logs 3, 5, 7
}
```

`Array.prototype.keys()`

The `Array.prototype.keys()` method can be used to iterate over indices like this:

Version ≥ 6

```
let myArray = [1, 2, 3, 4];
for (let i of myArray.keys()) {
  let twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

`Array.prototype.forEach()`

The `.forEach(...)` method is an option in ES 5 and above. It is supported by all modern browsers, as well as Internet Explorer 9 and later.

Version ≥ 5

```
[1, 2, 3, 4].forEach(function(value, index, arr) {
  var twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
});
```

Comparing with the traditional `for` loop, we can't jump out of the loop in `.forEach()`. In this case, use the `for` loop, or use partial iteration presented below.

In all versions of JavaScript, it is possible to iterate through the indices of an array using a traditional C-style **for** loop.

```
var myArray = [1, 2, 3, 4];
for(var i = 0; i < myArray.length; ++i) {
  var twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

It's also possible to use `while` loop:

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
  sum += i;
}
console.log(sum);
```

Array.prototype.every

Since ES5, if you want to iterate over a portion of an array, you can use Array.**prototype**.every, which iterates until we return **false**:

Version ≥ 5

```
// [].every() stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].every(function(value, index, arr) {
  console.log(value);
  return value % 2 === 0; // iterate until an odd number is found
});
```

Equivalent in any JavaScript version:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // iterate until an odd number is
found
  console.log(arr[i]);
}
```

Array.prototype.some

Array.**prototype**.some iterates until we return **true**:

Version ≥ 5

```
// [].some stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].some(function(value, index, arr) {
  console.log(value);
  return value === 7; // iterate until we find value 7
});
```

Equivalent in any JavaScript version:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
  console.log(arr[i]);
}
```

Libraries

Finally, many utility libraries also have their own `foreach` variation. Three of the most popular ones are these:

`jQuery.each()`, in jQuery:

```
$.each(myArray, function(key, value) {
    console.log(value);
});
```

`_.each()`, in Underscore.js:

```
_.each(myArray, function(value, key, myArray) {
    console.log(value);
});
```

`_.forEach()`, in Lodash.js:

```
_.forEach(myArray, function(value, key) {
    console.log(value);
});
```

See also the following question on SO, where much of this information was originally posted:

- Loop through an array in JavaScript

Section 12.7: Destructuring an array

Version ≥ 6

An array can be destructured when being assigned to a new variable.

```
const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;     // → true
height === 4;     // → true
hypotneuse === 5; // → true
```

Elements can be skipped

```
const [,b,,c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4
```

Rest operator can be used too

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

An array can also be destructured if it's an argument to a function.

```
function area([length, height]) {
  return (length * height) / 2;
}
```

```
const triangle = [3, 4, 5];

area(triangle); // → 6
```

Notice the third argument is not named in the function because it's not needed.

Learn more about destructuring syntax.

Section 12.8: Removing duplicate elements

From ES5.1 onwards, you can use the native method `Array.prototype.filter` to loop through an array and leave only entries that pass a given callback function.

In the following example, our callback checks if the given value occurs in the array. If it does, it is a duplicate and will not be copied to the resulting array.

Version ≥ 5.1

```
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
  return self.indexOf(value) === index;
}); // returns ['a', 1, 2, '1']
```

If your environment supports ES6, you can also use the Set object. This object lets you store unique values of any type, whether primitive values or object references:

Version ≥ 6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Section 12.9: Array comparison

For simple array comparison you can use JSON stringify and compare the output strings:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

> **Note:** that this will only work if both objects are JSON serializable and do not contain cyclic references. It may throw `TypeError: Converting circular structure to JSON`

You can use a recursive function to compare arrays.

```
function compareArrays(array1, array2) {
  var i, isA1, isA2;
  isA1 = Array.isArray(array1);
  isA2 = Array.isArray(array2);

  if (isA1 !== isA2) { // is one an array and the other not?
    return false;      // yes then can not be the same
  }
  if (! (isA1 && isA2)) {      // Are both not arrays
    return array1 === array2;  // return strict equality
  }
  if (array1.length !== array2.length) { // if lengths differ then can not be the same
    return false;
  }
  // iterate arrays and compare them
  for (i = 0; i < array1.length; i += 1) {
```

```
    if (!compareArrays(array1[i], array2[i])) { // Do items compare recursively
      return false;
    }
  }
  return true; // must be equal
}
```

WARNING: Using the above function is dangerous and should be wrapped in a `try catch` if you suspect there is a chance the array has cyclic references (a reference to an array that contains a reference to itself)

```
a = [0] ;
a[1] = a;
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded
```

> **Note:** The function uses the strict equality operator `===` to compare non array items `{a: 0} === {a: 0}` is `false`

Section 12.10: Reversing arrays

`.reverse` is used to reverse the order of items inside an array.

Example for `.reverse`:

```
[1, 2, 3, 4].reverse();
```

Results in:

```
[4, 3, 2, 1]
```

> **Note**: Please note that `.reverse`(`Array.prototype.reverse`) will reverse the array *in place*. Instead of returning a reversed copy, it will return the same array, reversed.
>
> ```
> var arr1 = [11, 22, 33];
> var arr2 = arr1.reverse();
> console.log(arr2); // [33, 22, 11]
> console.log(arr1); // [33, 22, 11]
> ```

You can also reverse an array 'deeply' by:

```
function deepReverse(arr) {
  arr.reverse().forEach(elem => {
    if(Array.isArray(elem)) {
      deepReverse(elem);
    }
  });
  return arr;
}
```

Example for deepReverse:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];
```

```
deepReverse(arr);
```

Results in:

```
arr // -> [[['c','b','a'], 3, 2, 1], 3, 2, 1]
```

Section 12.11: Shallow cloning an array

Sometimes, you need to work with an array while ensuring you don't modify the original. Instead of a `clone` method, arrays have a `slice` method that lets you perform a shallow copy of any part of an array. Keep in mind that this only clones the first level. This works well with primitive types, like numbers and strings, but not objects.

To shallow-clone an array (i.e. have a new array instance but with the same elements), you can use the following one-liner:

```
var clone = arrayToClone.slice();
```

This calls the built-in JavaScript `Array.prototype.slice` method. If you pass arguments to `slice`, you can get more complicated behaviors that create shallow clones of only part of an array, but for our purposes just calling `slice()` will create a shallow copy of the entire array.

All method used to convert array like objects to array are applicable to clone an array:

Version ≥ 6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // the shortest way
```

Version ≤ 5.1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

Section 12.12: Concatenating Arrays

Two Arrays

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
```

Version ≥ 3

```
var array3 = array1.concat(array2);  // returns a new array
```

Version ≥ 6

```
var array3 = [...array1, ...array2]
```

Results in a new `Array`:

```
[1, 2, 3, 4, 5]
```

Multiple Arrays

```
var array1 = ["a", "b"],
    array2 = ["c", "d"],
    array3 = ["e", "f"],
    array4 = ["g", "h"];
```

Version ≥ 3

Provide more Array arguments to `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

Version ≥ 6

Provide more arguments to `[]`

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Results in a new `Array`:

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Without Copying the First Array

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],
    shortArray = [9, 10];
```

Version ≥ 3

Provide the elements of `shortArray` as parameters to push using `Function.prototype.apply`

```
longArray.push.apply(longArray, shortArray);
```

Version ≥ 6

Use the spread operator to pass the elements of `shortArray` as separate arguments to `push`

```
longArray.push(...shortArray)
```

The value of `longArray` is now:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note that if the second array is too long (>100,000 entries), you may get a stack overflow error (because of how `apply` works). To be safe, you can iterate instead:

```
shortArray.forEach(function (elem) {
    longArray.push(elem);
});
```

Array and non-array values

```
var array = ["a", "b"];
```

Version ≥ 3

```
var arrConc = array.concat("c", "d");
```

Version ≥ 6

```
var arrConc = [...array, "c", "d"]
```

Results in a new `Array`:

```
["a", "b", "c", "d"]
```

You can also mix arrays with non-arrays

```
var arr1 = ["a","b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

Results in a new `Array`:

```
["a", "b", "c", "d", "e", "f"]
```

Section 12.13: Merge two array as key value pair

When we have two separate array and we want to make key value pair from that two array, we can use array's reduce function like below:

```
var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result =  rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {})

console.log(result);
```

Output:

```
{
  Date: "2001",
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}
```

Section 12.14: Array spread / rest

Spread operator

Version ≥ 6

With ES6, you can use spreads to separate individual elements into a comma-separated syntax:

```
let arr = [1, 2, 3, ...[4, 5, 6]];  // [1, 2, 3, 4, 5, 6]

// in ES < 6, the operations above are equivalent to
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

The spread operator also acts upon strings, separating each individual character into a new string element. Therefore, using an array function for converting these into integers, the array created above is equivalent to the one below:

```
let arr = [1, 2, 3, ...[..."456"].map(x=>parseInt(x))]; // [1, 2, 3, 4, 5, 6]
```

Or, using a single string, this could be simplified to:

```
let arr = [..."123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

If the mapping is not performed then:

```
let arr = [..."123456"]; // ["1", "2", "3", "4", "5", "6"]
```

The spread operator can also be used to spread arguments into a function:

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];

myFunction(...args);

// in ES < 6, this would be equivalent to:
myFunction.apply(null, args);
```

Rest operator

The rest operator does the opposite of the spread operator by coalescing several elements into a single one

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Collect arguments of a function:

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Section 12.15: Filtering values

The `filter()` method creates an array filled with all array elements that pass a test provided as a function.

Version ≥ 5.1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {
  return value > 2;
});
```

Version ≥ 6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

Results in a new array:

```
[3, 4, 5]
```

Filter falsy values

Version ≥ 5.1

```
var filtered = [ 0, undefined, {}, null, '', true, 5].filter(Boolean);
```

Since Boolean is a native JavaScript function/constructor that takes [one optional parameter] and the filter method also takes a function and passes it the current array item as parameter, you could read it like the following:

1. `Boolean(0)` returns false
2. `Boolean(undefined)` returns false
3. `Boolean({})` returns **true** which means push it to the returned array
4. `Boolean(null)` returns false
5. `Boolean('')` returns false
6. `Boolean(true)` returns **true** which means push it to the returned array
7. `Boolean(5)` returns **true** which means push it to the returned array

so the overall process will result

```
[ {}, true, 5 ]
```

Another simple example

This example utilises the same concept of passing a function that takes one argument

Version ≥ 5.1

```
function startsWithLetterA(str) {
    if(str && str[0].toLowerCase() == 'a') {
        return true
    }
    return false;
}

var str              = 'Since Boolean is a native javascript function/constructor that takes [one
optional parameter] and the filter method also takes a function and passes it the current array
item as a parameter, you could read it like the following';
var strArray         = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);
//["a", "and", "also", "a", "and", "array", "as"]
```

Section 12.16: Searching an Array

The recommended way (Since ES5) is to use Array.prototype.find:

```
let people = [
  { name: "bob" },
  { name: "john" }
];

let bob = people.find(person => person.name === "bob");

// Or, more verbose
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

In any version of JavaScript, a standard `for` loop can be used as well:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

FindIndex

The findIndex() method returns an index in the array, if an element in the array satisfies the provided testing function. Otherwise -1 is returned.

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
```

```
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

Section 12.17: Convert a String to an Array

The `.split()` method splits a string into an array of substrings. By default `.split()` will break the string into substrings on spaces (`" "`), which is equivalent to calling `.split(" ")`.

The parameter passed to `.split()` specifies the character, or the regular expression, to use for splitting the string.

To split a string into an array call `.split` with an empty string (`""`). **Important Note:** This only works if all of your characters fit in the Unicode lower range characters, which covers most English and most European languages. For languages that require 3 and 4 byte Unicode characters, `slice("")` will separate them.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

Version ≥ 6

Using the spread operator (`...`), to convert a `string` into an `array`.

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Section 12.18: Removing items from an array

Shift

Use `.shift` to remove the first item of an array.

For example:

```
var array = [1, 2, 3, 4];
array.shift();
```

array results in:

```
[2, 3, 4]
```

Pop

Further `.pop` is used to remove the last item from an array.

For example:

```
var array = [1, 2, 3];
array.pop();
```

array results in:

```
[1, 2]
```

Both methods return the removed item;

Splice

Use `.splice()` to remove a series of elements from an array. `.splice()` accepts two parameters, the starting index, and an optional number of elements to delete. If the second parameter is left out `.splice()` will remove all elements from the starting index through the end of the array.

For example:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

leaves `array` containing:

```
[1, 4]
```

The return of `array.splice()` is a new array containing the removed elements. For the example above, the return would be:

```
[2, 3]
```

Thus, omitting the second parameter effectively splits the array into two arrays, with the original ending before the index specified:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

...leaves `array` containing `[1, 2]` and returns `[3, 4]`.

Delete

Use `delete` to remove item from array without changing the length of array:

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

Array.prototype.length

Assigning value to `length` of array changes the length to given value. If new value is less than array length items will be removed from the end of value.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Section 12.19: Removing all elements

```
var arr = [1, 2, 3, 4];
```

Method 1

Creates a new array and overwrites the existing array reference with a new one.

```
arr = [];
```

Care must be taken as this does not remove any items from the original array. The array may have been closed over when passed to a function. The array will remain in memory for the life of the function though you may not be aware of this. This is a common source of memory leaks.

Example of a memory leak resulting from bad array clearing:

```
var count = 0;

function addListener(arr) { // arr is closed over
  var b = document.body.querySelector("#foo" + (count++));
  b.addEventListener("click", function(e) { // this functions reference keeps
    // the closure current while the
    // event is active
    // do something but does not need arr
  });
}

arr = ["big data"];
var i = 100;
while (i > 0) {
  addListener(arr); // the array is passed to the function
  arr = []; // only removes the reference, the original array remains
  array.push("some large data"); // more memory allocated
  i--;
}
// there are now 100 arrays closed over, each referencing a different array
// no a single item has been deleted
```

To prevent the risk of a memory leak use the one of the following 2 methods to empty the array in the above example's while loop.

Method 2

Setting the length property deletes all array element from the new array length to the old array length. It is the most efficient way to remove and dereference all items in the array. Keeps the reference to the original array

```
arr.length = 0;
```

Method 3

Similar to method 2 but returns a new array containing the removed items. If you do not need the items this method is inefficient as the new array is still created only to be immediately dereferenced.

```
arr.splice(0); // should not use if you don't want the removed items
// only use this method if you do the following
var keepArr = arr.splice(0); // empties the array and creates a new array containing the
                             // removed items
```

Related question.

Section 12.20: Finding the minimum or maximum element

If your array or array-like object is *numeric*, that is, if all its elements are numbers, then you can use `Math.min.apply` or `Math.max.apply` by passing **`null`** as the first argument, and your array as the second.

```
var myArray = [1, 2, 3, 4];

Math.min.apply(null, myArray); // 1
```

```
Math.max.apply(null, myArray); // 4
```

Version ≥ 6

In ES6 you can use the `...` operator to spread an array and take the minimum or maximum element.

```
var myArray = [1, 2, 3, 4, 99, 20];

var maxValue = Math.max(...myArray); // 99
var minValue = Math.min(...myArray); // 1
```

The following example uses a `for` loop:

```
var maxValue = myArray[0];
for(var i = 1; i < myArray.length; i++) {
   var currentValue = myArray[i];
   if(currentValue > maxValue) {
      maxValue = currentValue;
   }
}
```

Version ≥ 5.1

The following example uses `Array.prototype.reduce()` to find the minimum or maximum:

```
var myArray = [1, 2, 3, 4];

myArray.reduce(function(a, b) {
  return Math.min(a, b);
}); // 1

myArray.reduce(function(a, b) {
  return Math.max(a, b);
}); // 4
```

Version ≥ 6

or using arrow functions:

```
myArray.reduce((a, b) => Math.min(a, b)); // 1
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

Version ≥ 5.1

To generalize the `reduce` version we'd have to pass in an *initial value* to cover the empty list case:

```
function myMax(array) {
  return array.reduce(function(maxSoFar, element) {
    return Math.max(maxSoFar, element);
  }, -Infinity);
}

myMax([3, 5]);              // 5
myMax([]);                  // -Infinity
Math.max.apply(null, []);   // -Infinity
```

For the details on how to properly use `reduce` see Reducing values.

Section 12.21: Standard array initialization

There are many ways to create arrays. The most common are to use array literals, or the Array constructor:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

If the Array constructor is used with no arguments, an empty array is created.

```
var arr3 = new Array();
```

results in:

```
[]
```

Note that if it's used with exactly one argument and that argument is a `number`, an array of that length with all `undefined` values will be created instead:

```
var arr4 = new Array(4);
```

results in:

```
[undefined, undefined, undefined, undefined]
```

That does not apply if the single argument is non-numeric:

```
var arr5 = new Array("foo");
```

results in:

```
["foo"]
```

Version ≥ 6

Similar to an array literal, `Array.of` can be used to create a new `Array` instance given a number of arguments:

```
Array.of(21, "Hello", "World");
```

results in:

```
[21, "Hello", "World"]
```

In contrast to the Array constructor, creating an array with a single number such as `Array.of(23)` will create a new array `[23]`, rather than an Array with length 23.

The other way to create and initialize an array would be `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

will result:

```
[0, 1, 16, 81, 256]
```

Section 12.22: Joining array elements in a string

To join all of an array's elements into a string, you can use the `join` method:

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
```

```
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

As you can see in the second line, items that are not strings will be converted first.

Section 12.23: Removing/Adding elements using splice()

The `splice()`method can be used to remove elements from an array. In this example, we remove the first `3` from the array.

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
  values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

The `splice()` method can also be used to add elements to an array. In this example, we will insert the numbers 6, 7, and 8 to the end of the array.

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

The first argument of the `splice()` method is the index at which to remove/insert elements. The second argument is the number of elements to remove. The third argument and onwards are the values to insert into the array.

Section 12.24: The entries() method

The `entries()` method returns a new Array Iterator object that contains the key/value pairs for each index in the array.

Version ≥ 6

```
var letters = ['a','b','c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}
```

result

```
0 "a"
1 "b"
2 "c"
```

Note: This method is not supported in Internet Explorer.

Portions of this content from `Array.prototype.entries` by Mozilla Contributors licensed under CC-by-SA 2.5

Section 12.25: Remove value from array

When you need to remove a specific value from an array, you can use the following one-liner to create a copy array without the given value:

```
array.filter(function(val) { return val !== to_remove; });
```

Or if you want to change the array itself without creating a copy (for example if you write a function that get an array as a function and manipulates it) you can use this snippet:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

And if you need to remove just the first value found, remove the while loop:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index , 1); }
```

Section 12.26: Flattening Arrays

2 Dimensional arrays

Version ≥ 6

In ES6, we can flatten the array by the spread operator `...`:

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

Version ≥ 5

In ES5, we can achieve that by .apply():

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Higher Dimension Arrays

Given a deeply nested array like so

```
var deeplyNested = [4,[5,6,[7,8],9]];
```

It can be flattened with this magic

```
console.log(String(deeplyNested).split(',').map(Number);
#=> [4,5,6,7,8,9]
```

Or

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
#=> [4,5,6,7,8,9]
```

Both of the above methods only work when the array is made up exclusively of numbers. A multi-dimensional array of objects cannot be flattened by this method.

Section 12.27: Append / Prepend items to Array

Unshift

Use `.unshift` to add one or more items in the beginning of an array.

For example:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

array results in:

```
[1, 2, 3, 4, 5, 6]
```

Push

Further `.push` is used to add items after the last currently existent item.

For example:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

array results in:

```
[1, 2, 3, 4, 5, 6]
```

Both methods return the new array length.

Section 12.28: Object keys and values to array

```
var object = {
    key1: 10,
    key2: 3,
    key3: 40,
    key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

Now array is

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

Section 12.29: Logical connective of values

Version ≥ 5.1

`.some` and `.every` allow a logical connective of Array values.

While `.some` combines the return values with `OR`, `.every` combines them with `AND`.

Examples for `.some`

```
[false, false].some(function(value) {
  return value;
});
// Result: false

[false, true].some(function(value) {
  return value;
});
// Result: true

[true, true].some(function(value) {
  return value;
});
// Result: true
```

And examples for `.every`

```
[false, false].every(function(value) {
  return value;
});
// Result: false

[false, true].every(function(value) {
  return value;
});
// Result: false

[true, true].every(function(value) {
  return value;
});
// Result: true
```

Section 12.30: Checking if an object is an Array

`Array.isArray(obj)` returns **`true`** if the object is an `Array`, otherwise **`false`**.

```
Array.isArray([])           // true
Array.isArray([1, 2, 3])    // true
Array.isArray({})           // false
Array.isArray(1)            // false
```

In most cases you can **`instanceof`** to check if an object is an `Array`.

```
[] instanceof Array; // true
{} instanceof Array; // false
```

`Array.isArray` has the an advantage over using a **`instanceof`** check in that it will still return **`true`** even if the prototype of the array has been changed and will return **`false`** if a non-arrays prototype was changed to the `Array`

prototype.

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr);   // true
arr instanceof Array; // false
```

Section 12.31: Insert an item into an array at a specific index

Simple item insertion can be done with Array.**prototype**.splice method:

```
arr.splice(index, 0, item);
```

More advanced variant with multiple arguments and chaining support:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  this.splice.apply(this, [index, 0].concat(
    Array.prototype.slice.call(arguments, 1)));
  return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6);  // ["b", "X", "Y", "Z", "c"]
```

And with array-type arguments merging and chaining support:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  index = Math.min(index, this.length);
  arguments.length > 1
    && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
    && this.insert.apply(this, arguments);
  return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-");  // "a-b-V-W-X-Y-Z-c-d"
```

Section 12.32: Sorting multidimensional array

Given the following array

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

You can sort it sort it by number(second index)

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

Version ≥ 6

```
array.sort((a,b) => a[1] - b[1]);
```

This will output

```
[
  ["key2", 3],
  ["key1", 10],
  ["key4", 20],
  ["key3", 40]
]
```

Be aware that the sort method operates on the array *in place*. It changes the array. Most other array methods return a new array, leaving the original one intact. This is especially important to note if you use a functional programming style and expect functions to not have side-effects.

Section 12.33: Test all array items for equality

The `.every` method tests if all array elements pass a provided predicate test.

To test all objects for equality, you can use the following code snippets.

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true
```

Version ≥ 6

```
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

The following code snippets test for property equality

```
let data = [
  { name: "alice", id: 111 },
  { name: "alice", id: 222 }
];

data.every(function(item, i, list) { return item === list[0]; }); // false
data.every(function(item, i, list) { return item.name === list[0].name; }); // true
```

Version ≥ 6

```
data.every((item, i, list) => item.name === list[0].name); // true
```

Section 12.34: Copy part of an Array

The slice() method returns a copy of a portion of an array.

It takes two parameters, `arr.slice([begin[, end]])`:

begin

Zero-based index which is the beginning of extraction.

end

Zero-based index which is the end of extraction, slicing up to this index but it's not included.

If the end is a negative number,`end = arr.length + end`.

Example 1

```
// Let's say we have this Array of Alphabets
var arr = ["a", "b", "c", "d"...];

// I want an Array of the first two Alphabets
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

Example 2

```
// Let's say we have this Array of Numbers
// and I don't know it's end
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];

// I want to slice this Array starting from
// number 5 to its end
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```