JavaScript Notes
(part 1)

Julian Shen

07/31/2021 – 08/03/2021

(compilated elements from
codewithmosh.com JavaScript (part 1)
MDN Web Docs Mozilla)

For-In Loop

- iterates over all enumerable properties of an object that are keyed by strings
- Note: for…in should not be used to iterate over an array where the index order is important

Syntax

```
for (variable in object) {
     statement
}
```

Example

```
const object1 = { a: 1, b: 2, c: 3 };
for (const whatever in object) {
     console.log(`${whatever}: ${object[whatever]}`);
}
// expected output:
// a: 1
// b: 2
// c: 3
```

For-Of Loop

- iterates over iterable objects including: built-in String, Array, array-like objects, TypedArray, Map, Set, and user-defined iterables.

Syntax

```
for (variable of iterable) {
     statement
}
```

- Note: `variable` may be declared with `const`, `let`, or `var`.
- Use `let` instead of `const` if you reassign the variable inside the block.

Example (Iterate over an Array)

```
const array1 = [‘a’, ‘b’, ‘c’];
for (const whatever of array1) {
     console.log(whatever);
}
// expected output:
// a
// b
// c
```

Example (Iterate over a String)

```
const string1 = ‘ubi’;
for (const whatever of string1) {
     console.log(whatever);
}
// expected output:
// u
// b
// i
```

Difference between for…of and for…in

- The for…in statement iterates over the enumerable properties of an object, in arbitrary order.
- The for…of statement iterates over values that the iterable object defines to be iterated over.

Example (differences when used with an Array)

```
const iter = [3, 5, 6];
for (const i in iter)
     console.log(i); // 0, 1, 2
for (const i of iter)
     console.log(i); // 3, 5, 6
```

Break and Continue

- The break statement terminates the current loop, switch, or label statement and transfers program control to the statement following the terminated statement.

- The continue statement terminates execution of the statements in the current iteration of the current or labeled loop, and continues execution of the loop with the next iteration.

Factory Functions

- Factory functions are similar to constructor/class functions, but they do not require the use of the this keyword for inner values or the use of the new keyword when instantiating new objects. Instead of using new to create an object, factory functions simply creates an object and returns it.
- **Why is it useful?** If we have complex logic and we have to create multiple objects again and again that have the same logic, we can write the logic once in a function and use that function as a factory to create our objects, like in a real-world factory producing products.

Example (function creating new object w/o 'new' keyword)

```
function createRobot(name) {
      return {
            name,
            communicate() {
                  console.log('My name is ' + name + ', the robot.');
            }
      };
}
const robo1 = createRobot('R2D2');
robo1.communicate(); // My name is R2D2, the robot.
```

Example

```
let Person = function(name, age) {
      return {
            name,
            age,
            greet() {
                  return `Hello I'm ${name}, I'm ${age} years old`;
            }
      }
};
let person1 = Person('Julian', 34);
console.log(person1.greet()); // Hello I'm Julian, I'm 34 years old
```

Constructor Functions

Example

```
function Person(first, last, age) {
      this.firstName = first;
      this.lastName = last;
      this.age = age;
      this.default = ‘default’;
      this.fullName = function() {
            return this.firstName + ‘ ‘ + this.lastName;
      };
}
// Adding a Property to a Constructor
Person.nationality = ‘Canadian’;
// Note: Adding methods to an object constructor must be done inside
the constructor function
```

Value Types vs Reference Types

Value Types: Number, String, Boolean, Symbol, undefined, null
Reference Types: Object, Function, Array

Example (Value types)

```
let x = 10;
let y = x; // only the value of x is assigned to y
x = 20;
console.log(`%{x} and ${y}`); // 20 and 10
```

Example (Reference types)

```
let a = { property: ‘yesh’ }; // a is an object
let b = a; // address pointer to a is assigned to b
a.property = ‘yeppers’;
console.log(`{$a.property} and ${b.property}`); // yeppers and yeppers
```

Example (Primitives are copied by their value)

```
let number = 10;
function increase(num) {
      num++;
}
increase(number);
console.log(number); // 10
```

Example (Objects are copied by their reference)

```
let object = { value: 10 };
function increment(obj) {
      obj.value++;
}
increment(object);
console.log(object.value); // 11
```

Enumerating Properties of an Object

- `for…of` loop is only used for iterables like arrays and maps.
- Objects are not iterable, hence using `Object.keys(circle)` returns all keys of 'circle' as an array of keys (as strings) which is iterable

Example Object

```
const circle = {
      radius: 1,
      draw() {
            console.log('draw');
      }
};
```

Example (using Object.keys(circle))

```
for (let key of Object.keys(circle))
      console.log(key); // radius, draw
```

Example (using Object.entries(circle))

```
for (let entry of Object.entries(circle))
      console.log(entry); // [ 'radius', 1 ], [ 'draw', [Function: draw] ]
```

Example (using Object.values(circle))

```
for (let value of Object.values(circle))
      console.log(value); // 1, [Function: draw]
```

Example (if given property exists in an object)

```
('radius' in circle) ? yes : no; // yes
('color' in circle) ? yes : no; // no
```

Cloning an Object

- Spread Operator – copy properties of one object to another (…obj)

Example

```
const circle = {
      radius: 1,
      draw() {
            console.log('draw');
      }
};
const aDifferentCircle = { transparent: true, …circle, color: 'red' };
console.log(aDifferentCircle); // { transparent: true, radius: 1,
draw: [Function: draw], color: 'red' }
```

Strings

- JavaScript automatically wraps the String primitive with a String object allowing for use of String functions

Example (String Primitive)

```
const message1 = 'This is a message.';
console.log(typeof(message1)); // string
```

Example (String Object)

```
const message2 = 'This is another message.';
console.log(typeof(message2)); // object
```

Example (String functions)

```
console.log(message1.length); // 18
console.log(message1[5]); // i
console.log(message1.includes('is')); // true
console.log(message1.startsWith('This')); // true
console.log(message1.indexOf('is')); // 2
console.log(message1.replace('a', 'an')); // This is an message.
console.log(message1.toUpperCase()); // THIS IS A MESSAGE.
console.log(message1.split(' ')); // ['This', 'is', 'a', 'messages']
message1.trim(); // removes whitespace on left and right
message1.trimRight(); // removes whitespace on right
message1.trimLeft(); // removes whitespace on left
// changes to message1 are not permanent
// to make permanent, you have to reassign
message1 = message1.toLowerCase();
console.log(message1); // this is a message.
```

Adding/Removing Elements to/from Arrays

- The push() method appends one or more values/elements to the end of an array and returns the new length of the array.
- Note: Although strings are native, Array-like objects, they are not suitable for the push() method; strings are immutable.
- The pop() method removes the last element from an array and returns that removed element (returns undefined if empty). This changes the length of the array.
- The unshift() method adds one or more values/elements to the beginning of an array and returns the new length of the array.
- The shift() method removes the first element from an array and returns that removed element (returns undefined if empty). This changes the length of the array.
- The splice() method changes the content of an array by removing or replacing existing elements and/or adding new elements in place. Splice() returns an array containing the deleted elements.

Syntax (splice())

splice(startIndex, deleteCount, item1, item2, itemN)

- Note: deleteCount and item are optional. If only startIndex in parameter list, splice() will delete all elements that index onwards from the array.

Example

```
const numbers = [3, 4];

// Add to end of array (.push())
console.log(numbers.push(5, 6, 7)); // 5
console.log(numbers); // [3, 4, 5, 6, 7]

// Remove from end of array (.pop())
console.log(numbers.pop()); // 7
console.log(numbers); // [3, 4, 5, 6]

// Add to beginning of array (.unshift())
console.log(numbers.unshift(0, 1, 2)); // 7
console.log(numbers); // [0, 1, 2, 3, 4, 5, 6]

// Remove from beginning of array (.shift())
console.log(numbers.shift()); // 0
console.log(numbers); // [1, 2, 3, 4, 5, 6]

// Add to middle of array (.splice())
console.log(numbers.splice(2, 0, 'a', 'b')); // []
console.log(numbers); // [1, 2, 'a', 'b', 3, 4, 5, 6]
```

```
// Remove the 5th element of the array (.splice())
console.log(numbers.splice(4, 1)); // [3]
console.log(numbers); // [1, 2, ‘a’, ‘b’, 4, 5, 6]

// Remove the last half of the array (.splice())
console.log(numbers.splice(3)); // [‘b’, 4, 5, 6]
console.log(numbers); // [1, 2, ‘a’]

// Remove 1 element starting at index 1 (.splice())
console.log(numbers.splice(1, 1)); // [2]
console.log(numbers); // [1, ‘a’]
```

Emptying an Array

Example

```
const array = [1, 2, 3, 4, 5];
const arrayCopy = array;
const arrayCopy2 = arrayCopy;

// Solution 1
console.log(arrayCopy); // [1, 2, 3, 4, 5]
console.log(arrayCopy2); // [1, 2, 3, 4, 5]
array.length = 0;
console.log(array); // []
console.log(arrayCopy) // []
console.log(arrayCopy2) // []
```

Finding Elements of Primitive Types

- The indexOf() method returns the first index at which a given element can be found in the array, or -1 if it is not present.
- The lastIndexOf() method returns the last index at which a given element can be found in the array, or -1 if not present. The array is searched backwards starting at fromIndex.
- The includes() method determines whether an array includes a certain value among its entries, returning true or false.

Syntax (.indexOf())

```
indexOf(searchElement)
indexOf(searchElement, fromIndex)
```

- Note: If fromIndex >= array's length, -1 is returned.
- Note: If fromIndex is negative, it is taken as the offset from the end of the array.
- Note: If fromIndex = 0 (default), then the whole array is searched.

Syntax (.lastIndexOf())

```
lastIndexOf(searchElement)
lastIndexOf(searchElement, fromIndex)
```

- Note: If fromIndex >= array's length, whole array is searched.
- Note: If fromIndex is negative, it is taken as the offset from the end of the array.

Syntax (.includes())

```
includes(searchElement)
includes(searchElement, fromIndex)
```

- Note: When comparing strings and characters, is case-sensitive.
- Note: If fromIndex >= array's length, false is returned and array will not be searched.
- Note: If fromIndex is negative, it is taken as the offset from the end of the array.

Example

```
const array1 = [1, 2, 3, 4, 'a', 'b', 'c', 'b', 2, 5, 6, 7];

console.log(array1.indexOf(2)); // 1
console.log(array1.indexOf(2, 2)); // 8

console.log(array1.indexOf('b')); // 7
console.log(array1.indexOf('b', 6)); // 5

console.log(array1.indexOf('b')); // true
console.log(array1.indexOf('b', 8)); // false
```

Finding Elements of Reference Types

- The `find()` method returns the `value` of the first element in the provided array (or object) that satisfies the provided testing function. If no value satisfies the testing function, `undefined` is returned.
- The `findIndex()` method returns the `index` of the first element in the array that satisfies the provided testing function. Otherwise, it returns `-1` indicating that no element passed the test.

Syntax (.find()) or (.findIndex) (identical)

```
// Arrow function
find((element) => {…} )
find((element, index) => {…} )
find((element, index, array) => {…} )
// Callback function
find(callbackfn)
find(callbackfn, thisArg)
// Inline callback function
find(function callbackFn(element) {…})
find(function callbackFn(element, index) {…})
find(function callbackFn(element, index, array) {…})
find(function callbackFn(element, index, array) {…}, thisArg)
```

Example

```
const courses = [
     { id: 1, name: 'a' },
     { id: 2, name: 'b' },
     { id: 3, name: 'c' },
     { id: 4, name: 'd' }
];

// find()
const course = courses.find(courseObj => courseObj.name === 'd');
console.log(course); // { id: 4, name: 'd' }

// findIndex()
const courseIndex = courses.findIndex(function(courseObj) {
     return courseObj.id === 3;
});
console.log(courseIndex); // 2
```

Arrow Function Expressions

- An arrow function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

Differences and Limitations

- Does not have its own bindings to `this` or `super`, and should not be used as methods.
- Does not have `arguments` or `new.target` keywords.
- Not suitable for `call`, `apply`, and `bind` methods, which generally rely on establishing a scope.
- Can not be used as constructors.
- Can not use `yield`, within its body.

Example

```
// Traditional Single Argument Function
function(a) {
	return a * 2;
}
// Arrow Function
a => a * 2;
// Traditional Multiple Argument Function
function(a, b) {
	return a + b + 100;
}
// Arrow Function
(a, b) => a + b + 100;
// Traditional Function w/Additional Lines
function(a, b) {
	let stuff = 22;
	return a + b + stuff;
}
// Arrow Function
(a, b) => {
	let stuff = 22;
	return a + b + stuff;
}
// Traditional Named Function
function blurb(a) {
	return a % 2;
}
// Arrow Function
let blurb = a => a % 2;
```

Combining and Slicing Arrays

- The concat() method is used to merge two or more arrays.
- Note: this method does not change the existing arrays, but instead returns a new array.
- The slice() method returns a shallow copy of a portion of an array into a new array object selected from start to end(end not included) where start and end represent the index of items in that array.
- Note: The original array will not be modified.

Syntax (.concat())

```
concat()
concat(value0)
concat(value0, value1, …, valueN)
```

- Note: if parameters are omitted, concat() returns a shallow copy of the existing array on which it is called.

Syntax (.slice())

```
slice()
slice(start)
slice(start, end)
```

- Note: if start or end are negative numbers, it is considered an offset from the end of the sequence.
- Note: if start is greater than the index range of the sequence, an empty array is returned.
- Note: if end is greater than the length of the sequence, it extracts through to the end of the sequence (arr.length).

Example

```
const ar = [1, 2, 3, 4, 5, 6];

let ar1 = ar.concat();
console.log(ar1); // [1, 2, 3, 4, 5, 6]
ar1 = ar.concat(7, 8, 9)
console.log(ar1); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
console.log(ar); // [1, 2, 3, 4, 5, 6]

let ar2 = ar.slice();
console.log(ar2); // [1, 2, 3, 4, 5, 6]
ar2 = ar.slice(3);
console.log(ar2); // [4, 5, 6]
ar2 = ar.slice(2, 5);
console.log(ar2); // [3, 4, 5]
```

Spread Operator for Arrays

- The spread syntax (…) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

Example

```
const first = [1, 2, 3];
const second = [4, 5, 6];

const combined = […first, ‘a’, …second, ‘b’];
console.log(combined); // [1, 2, 3, ‘a’, 4, 5, 6, ‘b’];

const copy = […combined];
console.log(copy); // [1, 2, 3, ‘a’, 4, 5, 6, ‘b’];
```

Iterating an Array (forEach())

- The forEach() method calls a provided callbackFn function once for each element in an array in ascending index order. Returns undefined.
- Note: forEach() does not mutate the array on which it called, the callbackFn may do so.

Syntax

```
// Arrow function
forEach((element) => {…} )
forEach((element, index) => {…} )
forEach((element, index, array) => {…} )
// Callback function
forEach(callbackFn)
forEach(callbackFn, thisArg)
// Inline callback function
forEach(function callbackFn(element) {…})
forEach(function callbackFn(element, index) {…})
forEach(function callbackFn(element, index, array) {…})
forEach(function callbackFn(element, index, array) {…}, thisArg)
```

- callbackFn is the function to execute on each element.
- element is the current element being processed in the array.
- index (optional) is the index of the element of the array.
- array (optional) is the array the forEach() was called upon.
- thisArg (optional) is the value to use as this when executing callbackFn.

Example

```
const numbers = [1, 2, 3];
// Inline callback function
numbers.forEach(function(number) {
      console.log(number); // 1, 2, 3
});
// Arrow function
numbers.forEach((number, index) => console.log(index, number)); // 0
1, 1 2, 2 3
```

Joining and Splitting Arrays

- The join() method creates and returns a new string by concatenating all of the elements in an array (or array-like object), separated by commas (default) or a specified separator string. If the array has only one item, that that item will be returned without using the separator. If the array is empty, an empty string is returned.
- The split() method divides a string into an ordered list of substrings, puts these substrings into an array, and returns the array.

Syntax

```
join()
join(separator)

split()
split(separator)
split(separator, limit)
```

- separator (optional) is the pattern describing where each split should occur.
- Note: if the separator contains multiple characters, that entire character sequence must be found in order to split.
- Note: if the separator appears at the beginning or end of the string, it still has the effect of splitting.
- limit (optional) is a non-negative integer specifying a limit on the number of substrings to be included in the array. Any leftover text is not included in the array at all.
- Note: if limit is 0, [] is returned.

Example

```
let s = 'I am learning to code javascript.';

console.log(s.split()); // ['I am learning to code javascript.']
console.log(s.split(' ')); // ['I', 'am', 'learning', 'to', 'code',
'javascript.']
console.log(s.split(' ', 5)); // ['I', 'am', 'learning', 'to', 'code']
console.log(s.split(' ', -5)); // ['I', 'am', 'learning', 'to',
'code', 'javascript.']

let arr = [ 1, 2, 3, 4, 5];
let join = arr.join();
console.log(join); // 1,2,3,4,5
console.log(arr.join(' ')); // 1 2 3 4 5
console.log(arr.join(' and ')); // 1 and 2 and 3 and 4 and 5
```

Sorting and Reversing Arrays

- The sort() method sorts the elements of an array in place and returns the sorted array. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units into values.
- The reverse() method reverses an array in place.

Syntax

```
// Functionless
sort()
// Arrow function
sort((elem1, elem2) => {…})
// Compare function
sort(compareFn)
// Inline compare function
sort(function compareFn(elem1, elem2) {…})
```

- compareFn (optional) specifies a function that defines the sort order.
- elem1, elem2 are the first and second elements for comparison.

```
reverse()
```

Example

```
const numbers = [3, 6, 1, 7, 4];
numbers.reverse();
console.log(numbers); // [4, 7, 1, 6, 3]
numbers.sort();
console.log(numbers); // [1, 3, 4, 6, 7]

const courses = [
      { id: 1, name: 'Python' },
      { id: 2, name: 'C++' },
      { id: 3, name: 'JavaScript' }
];
courses.sort(function(a, b) {
      // a < b => -1
      // a > b => 1
      // a === b => 0
      const nameA = a.name.toUpperCase();
      const nameB = b.name.toUpperCase();
      if (nameA < nameB) return -1; // don't switch a and b
      if (nameA > nameB) return 1; // switch a and b
      return 0; // jump out of function
});
console.log(courses); // [ {id: 2, name: 'C++}, {id: 3, name:
'JavaScript'}, {id: 1, name: 'Python'}]
```

Testing Array Elements with every() and some()

- The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.
- The some() method tests whether at least element in the array passes the test implemented by the provided function. It returns true if it finds an element for which the provided function returns true; otherwise it returns false. It doesn't modify the array.

Syntax (same for some()

```
// Arrow function
every((element) => { ... } )
every((element, index) => { ... } )
every((element, index, array) => { ... } )
// Callback function
every(callbackFn)
every(callbackFn, thisArg)
// Inline callback function
every(function callbackFn(element) { ... })
every(function callbackFn(element, index) { ... })
every(function callbackFn(element, index, array){ ... })
every(function callbackFn(element, index, array) { ... }, thisArg)
```

- callbackFn is a function to test for each element.
- element is the current element being processed in the array.
- index (optional) is the index of the current element being processed in the array.
- array (optional) is the array every() or some() was called upon.
- thisArg (optional) is the value to use as this when executing callbackFn.
- Note: Returns true if the callbackFn function returns a truthy value; otherwise, false.

Example

```
const numbers = [1, 2, 3];
const isAllPositive = numbers.every(function(value) {
      return value >= 0;
});
console.log(allpositive); // true

const numbers2 = [-1, 1, -2, -3];
const atLeastOnePositive = numbers2.some(function(value) {
      return value >= 0;
});
console.log(atLeastOnePositive); // true
```

Filtering an Array

- The `filter()` method creates and returns a new array with all elements that pass the test implemented by the provided function.

Syntax

```
// Arrow function
filter((element) => { ... } )
filter((element, index) => { ... } )
filter((element, index, array) => { ... } )
// Callback function
filter(callbackFn)
filter(callbackFn, thisArg)
// Inline callback function
filter(function callbackFn(element) { ... })
filter(function callbackFn(element, index) { ... })
filter(function callbackFn(element, index, array){ ... })
filter(function callbackFn(element, index, array) { ... }, thisArg)
```

- `callbackFn` is a function to test each element of the array. Returns a value that coerces to `true` to keep the element or `false` otherwise.
- `element` is the current element being processed in the array.
- `index` (`optional`) is the index of the current element being processed in the array.
- `array` (`optional`) the array `filter()` was called upon.
- `thisArg` (`optional`) is the value to use as `this` when executing `callbackFn`.

Example

```
const numbers = [-1, 1, -2, 3];

const filtered = numbers.filter(n => n >= 0);
console.log(filtered); // [1, 3]
```

Mapping and Joining an Array

- The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.
- Note: callbackFn is not called for missing elements of the array such as indexes that have never been set or deleted indexes.
- Note: You shouldn't be using map() if you're not using the array it returns or if you're not returning a value from the callback. Use forEach() or for…of instead.
- The join() method creates and returns a new string by concatenating all of the elements in an array (or array-like object), separated by commas (default) or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

Syntax

```
// Arrow function
map((element) => { ... } )
map((element, index) => { ... } )
map((element, index, array) => { ... } )
// Callback function
map(callbackFn)
map(callbackFn, thisArg)
// Inline callback function
map(function callbackFn(element) { ... })
map(function callbackFn(element, index) { ... })
map(function callbackFn(element, index, array){ ... })
map(function callbackFn(element, index, array) { ... }, thisArg)
```

- callbackFn is a function that is called for every element of array. Each time callbackFn executes, the returned value is added to newArray.
- element is the current element being processed in the array.
- index (optional) is the index of the current element being processed in the array.
- array (optional) the array map() was called upon.
- thisArg (optional) is the value to use as this when executing callbackFn.

```
join()
join(separator)
```

- separator (optional) specifies a string (or converted to a string if necessary) to separate each pair of adjacent elements of the array.
- Note: If separator is an empty string, all elements are joined without any characters in between them.

Example

```
const groceryList = ['apples', 'oranges', 'bananas'];

const htmlMapped = groceryList.map(n => '<li>' + n + '</li>');
console.log(htmlMapped); // ['<li>apples</li>', '<li>oranges</li>',
'<li>bananas</li>']

const htmlJoined = '<ul>' + htmlMapped.join('') + '</ul>';
console.log(htmlJoined); //
<ul><li>apples</li><li>oranges</li><li>bananas</li></ul>

// Cleaner code of above by chaining
// let html = groceryList
//    .map(n => '<li>' + n + '</li>')
//    .join('');
// html = '<ul>' + html + '</ul>';
// console.log(html); //
<ul><li>apples</li><li>oranges</li><li>bananas</li></ul>

// Mapping into Objects
const objGroceries = groceryList.map(n=> ({ item: n }));
console.log(objGroceries); // [ { item: 'apples' }, { item:
'oranges' }, { item: 'bananas' } ]
```

Reducing an Array

- The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

Syntax

```
// Arrow function
reduce((accumulator, currentValue) => { ... } )
reduce((accumulator, currentValue, index) => { ... } )
reduce((accumulator, currentValue, index, array) => { ... } )
reduce((accumulator, currentValue, index, array) => { ... },
initialValue)
// Callback function
reduce(callbackFn)
reduce(callbackFn, initialValue)
// Inline callback function
reduce(function callbackFn(accumulator, currentValue) { ... })
reduce(function callbackFn(accumulator, currentValue, index) { ... })
reduce(function callbackFn(accumulator, currentValue, index,
array){ ... })
reduce(function callbackFn(accumulator, currentValue, index, array)
{ ... }, initialValue)
```

- callbackFn is a function to execute on each element in the array (except for the first, if no initialValue is supplied).
- accumulator is the accumulated value previously returned in the last invocation of the callback (or initialValue if it was supplied). It accumulates the callbackFn's return values.
- currentValue is the current element being processed in the array.
- index (optional) is the index of the current element being processed in the array. Starts from index 0 if an initialValue is provided, otherwise, it starts from index 1.
- array (optional) is the array reduce() was called upon.
- initialValue (optional) is a value to use as the first argument to the first call of the callbackFn. If no initialValue is supplied, the first element in the array will be used as the initial accumulator value and skipped as the currentValue.
- Note: Calling reduce() on an empty array without an initialValue will throw a TypeError.

Examples

```
// sum
const numbers = [1, -1, 2, 3];
const sum = numbers.reduce((accumulator, currentValue) => accumulator
+ currentValue);
console.log(sum); // 5

// countOccurrences
function countOccurrences(array, searchElement) {
      return array.reduce((accumulator, currentValue) => {
            if (currentValue === searchElement)
                  return accumulator + 1;
            return accumulator;
      }, 0);
}

const numbers = [1, 2, -1, 3, 4, 1, -1, 1, 1, 1];
console.log(countOccurrences(numbers, 1)); // 5
console.log(countOccurrences(numbers, -1)); // 2

// getMax
function getMax(array) {
      if (array.length === 0) return undefined;
      return array.reduce((a, b) => (a > b)? a : b);
}

const numbers = [44, 125, 67, 223, 1, 226, 156, 203];
console.log(getMax(numbers)); // 226
```

Function Declarations vs Expressions

```
// Function Declaration
function walk() {
      console.log('walk');
} // no semicolon

// Anonymous Function Expression
// Functions are objects, so run is being set as an object
let run = function() {
      console.log('run');
}; // semicolon

// Named Function Expression
let crawl = function crawl() {
      console.log('crawl');
};

let move = run; // Both move and run reference the same anonymous
function
```

Hoisting Functions

- Hoisting: function declarations are moved to the top to be processed first when program is run
- Function Declarations can be called before they are defined
- Function Expressions cannot be called before they are defined

Example

```
walk(); // walk
// Function Declaration
function walk() {
      console.log('walk');
} // no semicolon

run(); // error: 'Uncaught ReferenceError: run is not defined'
// Anonymous Function Expression
// Functions are objects, so run is being set as an object
let run = function() {
      console.log('run');
}; // semicolon
```

The Arguments Object

- `arguments` is an array-like object accessible inside functions that contains the values of the arguments passed to that function.
- `Note:` "Array-like" means that `arguments` have a `length` property and properties indexed from zero, but it doesn't have array's built-in methods like `forEach()` or `map()`.

Example

```
function func1(a, b) {
      console.log(arguments);
      console.log(arguments[0]);
      console.log(arguments[1]);
}

func1(3, 'pineapple');
// [Arguments] { '0': 3, '1': 'pineapple' }
// 3
// pineapple
```

The Rest Parameter

- The `rest parameter` syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent variadic functions in JavaScript.
- A function definition's last parameter can be prefixed with '…' which will cause all remaining (user supplied) parameters to be placed within a 'standard' JavaScript array. Only the last parameter in a function definition can be a rest parameter.

Example

```
function func1(a, b, ...restArgs) {
      console.log("a: ", a);
      console.log("b: ", b);
      console.log("rest arguments: ", restArgs);
}

func1(3, 'pineapple', true, 'bicycle', 1.24);
// a:  3
// b:  pineapple
// rest arguments:  [ true, 'bicycle', 1.24 ]
```

Default Parameters

- Default function parameters allow named parameters to be initialized with default values if not value or undefined is passed.
- Note: In a list of parameters, all defaulted parameters need to hug to the right.

Example

```
function interest(principal, rate = 3.5, years = 5) {
     return principal * rate / 100 * years;
}
console.log(interest(10000)); // 1750
```

Getters and Setters

- The get syntax binds an object property to a function that will be called when that property is looked up.
- The set syntax binds an object property to a function to be called when there is an attempt to set that property.

Syntax

```
// getter
{get prop() {…} }
{get [expression]() {…} }
```

- prop is the name of the property to bind to the given function.
- expression can be used for a computed property name to bind to the given function.

```
// setter
{set prop(value) {…} }
{set [expression](value) {…} }
```

- prop is the name of the property to bind to the given function.
- value is the variable that holds the value attempted to be assigned to prop.
- expression can be used for a computed property name to bind to the given function.

Example

```
const person = {
      firstName: 'Julian',
      lastName: 'Shen',
      get fullName() {
            return `${person.firstName} ${person.lastName}`;
      },
      set fullName(value) {
            const parts = value.split(' ');
            this.firstName = parts[0];
            this.lastName = parts[1];
      }
};

console.log(`${person.firstName} ${person.lastName}`); // Julian Shen
//console.log(person.fullName());

// getters => access properties
// setters => change (mutate) them

// using setter
person.fullName = 'Mosh Hamedani';
console.log(person); // { firstName: 'Mosh', lastName: 'Hamedani',
fullName: [Getter/Setter] }

// using getter
console.log(person.fullName); // Mosh Hamedani
```

Try...Catch

- The try…catch statement marks a block of statements to try and specifies a response should an exception be thrown.

Syntax

```
try {
     try_statements
}
catch (exception_var) {
     catch_statements
}
finally {
     finally_statements
}
```

- try_statements are the statements to be executed.
- catch_statements is the statement that is executed if an exception is thrown in the try-block.
- exception_var is an optional identifier to hold an exception object for the associated catch-block.
- finally_statements are statements that are executed after the try statement completes. These statements execute regardless of whether an exception was thrown or caught.

Example

```
const person = {
	firstName: 'Julian',
	lastName: 'Shen',
	get fullName() {
		return `${person.firstName} ${person.lastName}`;
	},
	set fullName(value) {
		// defensive programming
		if (typeof value !== 'string')
			// Error is in pascal case, hence a constructor (new
keyword)
			throw new Error('Value is not a string.');

		const parts = value.split(' ');
		if (parts.length !== 2)
			throw new Error('Enter a first and last name');

		this.firstName = parts[0];
		this.lastName = parts[1];
	}
};

try {
	person.fullName = ''; // Error: Enter a first and last name
}
catch (e) {
	console.log(e);
	alert(e);
}
```

Let vs Var

- The var statement declares a function-scoped or globally-scoped variable, optionally initializing it to a value.
- The let statement declares a block-scoped local variable, optionally initializing it to a value.

Example

```
function start() {
	for (var i = 0; i < 5; i++) {
		console.log(i);
	}
	console.log(i); // 5 (i is accessible outside of its scope)
}

start(); // 0, 1, 2, 3, 4, 5

// global window object
var color = 'red'; // window.color = 'red'
let age = 30; // window.age = undefined
// Likewise, functions are connected to the window object as well
// i.e. - window.start() will run the function start()
```

This

- The this keyword references the object that is executing the current function.
- A method is a function that is defined in an object. If the this keyword is in a method, then it references the object itself.
- If the this keyword is in a regular function (not part of an object), this references the global object which is window (in browsers) and global (in node).

Example

```
// video object
const video = {
	title: 'a',
	play() {
		console.log(this);
	}
};
// add stop function to video object
// ‘this’ below references the ‘video’ object
video.stop = function() {
	console.log(this);
};
```

```
// ‘this’ below references the global object
function playVideo() {
      console.log(this);
}

video.play(); // { title: 'a', play: [Function: play], stop:
[Function] }
video.stop(); // { title: 'a', play: [Function: play], stop:
[Function] }
//playVideo(); // displays global object



// Video constructor function
function Video(title) {
      this.title = title;
      console.log(this);
}

// the ‘new’ operator creates a new empty object {}, and sets 'this'
to point to the empty object
const v = new Video('apples'); // Video { title: 'apples' }



const dvd = {
      title: 'a',
      tags: ['a', 'b', 'c'],
      showTags() {
            this.tags.forEach(function(tag) {
                  console.log(this.title, tag);
            }, this); // ‘this’ is an arg param that makes the 'this'
(this.title) above it reference to dvd object rather than global
object
      }
};
// the function (function(tag)…{}) is just a regular function within
the showTags() function so it references the global object (not the
dvd object), hence the need to specify the secondary ‘this’ in the
‘thisArg’ variable of the forEach() method

dvd.showTags(); // a a, a b, a c
```

Changing the Value of “this”

```
// Solution 1 (not preferred approach)
const dvd = {
	title: 'a',
	tags: ['a', 'b', 'c'],
	showTags() {
		const self = this; // ‘this’ refers to the dvd object
		this.tags.forEach(function(tag) {
			console.log(self.title, tag);
		}); // this is an arg param that makes the 'this' above
reference to dvd object rather than global object
	}
};

dvd.showTags(); // aa, ab, ac

// Solution 2
// .bind functioning instead of const self = this;
const movie = {
	title: 'a',
	tags: ['a', 'b', 'c'],
	showTags() {
		this.tags.forEach(function(tag) {
			console.log(this.title, tag);
		}.bind(this)); // .bind(this) binds the global function to
the movie object
	}
};
movie.showTags(); // a a, a b, a c

// Solution 3 (preferred solution)
// arrow functions (ecmascript 6) inherit the ‘this’ value
const bluRay = {
	title: 'a',
	tags: ['a', 'b', 'c'],
	showTags() {
		this.tags.forEach(tag => {
			console.log(this.title, tag);
		}); // ‘this’ is automatically bound to bluRay
	}
};
bluRay.showTags(); // a a, a b, a c
```

.call(), .apply(), .bind()

- The call() method calls a function with a given this value and arguments provided individually.
- The apply() method calls a function with a given this value and arguments provided as an array (or an array-like object).
- The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

Syntax

```
// call()
call()
call(thisArg)
call(thisArg, arg1, ... , argN)
// apply()
apply(thisArg)
apply(thisArg, argsArray)
// bind()
bind(thisArg)
bind(thisArg, arg1, ... , argN)
```

Example

```
function playVideo(a, b) {
      console.log(this, a, b);
}

// .call, .apply, .bind all take a ‘thisArg’ param that assigns ‘this’ in playVideo() to the ‘thisArg’ object

// .call (similar to .apply but params ‘a’ and ‘b’ from playVideo() follow the ‘thisArg’ object as a list)
playVideo.call({ name: 'Mosh' }, 1, 2); // { name: ‘Mosh’ } 1 2
// playVideo(); // references global object if no parameter input

// .apply (similar to .call but params ‘a’ and ‘b’ from playVideo() follow the ‘thisArg’ object in an array form)
playVideo.apply({ name: 'Julian' }, [3, 4]); // { name: ‘Julian’ } 3 4

// .bind (sets ‘this’ to reference the ‘thisArg’ object PERMANENTLY and returns the new function to be stored in a new variable)
const fn = playVideo.bind({ name: 'Alice' }, 5, 6);
fn(); // { name: ‘Alice’ } 5 6
// same as above
playVideo.bind({ name: 'Susan' }, 7, 8)(); // { name: ‘Susan’ } 7 8
```

Table of Contents