Chapter 13: Objects

Property	Description
`value`	The value to assign to the property.
`writable`	Whether the value of the property can be changed or not.
`enumerable`	Whether the property will be enumerated in `for in` loops or not.
`configurable`	Whether it will be possible to redefine the property descriptor or not.
`get`	A function to be called that will return the value of the property.
`set`	A function to be called when the property is assigned a value.

Section 13.1: Shallow cloning

Version ≥ 6

ES6's `Object.assign()` function can be used to copy all of the **enumerable** properties from an existing `Object` instance to a new one.

```
const existing = { a: 1, b: 2, c: 3 };

const clone = Object.assign({}, existing);
```

This includes `Symbol` properties in addition to `String` ones.

Object rest/spread destructuring which is currently a stage 3 proposal provides an even simpler way to create shallow clones of Object instances:

```
const existing = { a: 1, b: 2, c: 3 };

const { ...clone } = existing;
```

If you need to support older versions of JavaScript, the most-compatible way to clone an Object is by manually iterating over its properties and filtering out inherited ones using `.hasOwnProperty()`.

```
var existing = { a: 1, b: 2, c: 3 };

var clone = {};
for (var prop in existing) {
  if (existing.hasOwnProperty(prop)) {
    clone[prop] = existing[prop];
  }
}
```

Section 13.2: Object.freeze

Version ≥ 5

`Object.freeze` makes an object immutable by preventing the addition of new properties, the removal of existing properties, and the modification of the enumerability, configurability, and writability of existing properties. It also prevents the value of existing properties from being changed. However, it does not work recursively which means that child objects are not automatically frozen and are subject to change.

The operations following the freeze will fail silently unless the code is running in strict mode. If the code is in strict

mode, a `TypeError` will be thrown.

```
var obj = {
  foo: 'foo',
  bar: [1, 2, 3],
  baz: {
    foo: 'nested-foo'
  }
};

Object.freeze(obj);

// Cannot add new properties
obj.newProperty = true;

// Cannot modify existing values or their descriptors
obj.foo = 'not foo';
Object.defineProperty(obj, 'foo', {
    writable: true
});

// Cannot delete existing properties
delete obj.foo;

// Nested objects are not frozen
obj.bar.push(4);
obj.baz.foo = 'new foo';
```

Section 13.3: Object cloning

When you want a complete copy of an object (i.e. the object properties and the values inside those properties, etc...), that is called **deep cloning**.

`Version ≥ 5.1`

If an object can be serialized to JSON, then you can create a deep clone of it with a combination of `JSON.parse` and `JSON.stringify`:

```
var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c will not change
```

Note that `JSON.stringify` will convert `Date` objects to ISO-format string representations, but `JSON.parse` will not convert the string back into a `Date`.

There is no built-in function in JavaScript for creating deep clones, and it is not possible in general to create deep clones for every object for many reasons. For example,

- objects can have non-enumerable and hidden properties which cannot be detected.
- object getters and setters cannot be copied.
- objects can have a cyclic structure.
- function properties can depend on state in a hidden scope.

Assuming that you have a "nice" object whose properties only contain primitive values, dates, arrays, or other "nice" objects, then the following function can be used for making deep clones. It is a recursive function that can detect objects with a cyclic structure and will throw an error in such cases.

```
function deepClone(obj) {
    function clone(obj, traversedObjects) {
        var copy;
        // primitive types
        if(obj === null || typeof obj !== "object") {
            return obj;
        }

        // detect cycles
        for(var i = 0; i < traversedObjects.length; i++) {
            if(traversedObjects[i] === obj) {
                throw new Error("Cannot clone circular object.");
            }
        }

        // dates
        if(obj instanceof Date) {
            copy = new Date();
            copy.setTime(obj.getTime());
            return copy;
        }
        // arrays
        if(obj instanceof Array) {
            copy = [];
            for(var i = 0; i < obj.length; i++) {
                copy.push(clone(obj[i], traversedObjects.concat(obj)));
            }
            return copy;
        }
        // simple objects
        if(obj instanceof Object) {
            copy = {};
            for(var key in obj) {
                if(obj.hasOwnProperty(key)) {
                    copy[key] = clone(obj[key], traversedObjects.concat(obj));
                }
            }
            return copy;
        }
        throw new Error("Not a cloneable object.");
    }

    return clone(obj, []);
}
```

Section 13.4: Object properties iteration

You can access each property that belongs to an object with this loop

```
for (var property in object) {
    // always check if an object has a property
    if (object.hasOwnProperty(property)) {
        // do stuff
    }
}
```

You should include the additional check for `hasOwnProperty` because an object may have properties that are inherited from the object's base class. Not performing this check can raise errors.

Version ≥ 5

You can also use `Object.keys` function which return an Array containing all properties of an object and then you can loop through this array with `Array.map` or `Array.forEach` function.

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
    console.log(key);
});
// outputs: 0, 1, 2
```

Section 13.5: Object.assign

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Use it to assign values to an existing object:

```
var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Or to create a shallow copy of an object:

```
var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Or merge many properties from multiple objects to one:

```
var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // Logs: { a: 1, b: 2, c: 3 }
console.log(obj1); // Logs: { a: 1, b: 2, c: 3 }, target object itself is changed
```

Primitives will be wrapped, null and undefined will be ignored:

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');

var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // Logs: { "0": "a", "1": "b", "2": "c" }
```

> Note, only string wrappers can have own enumerable properties

Use it as reducer: (merges an array to an object)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user})
```

Section 13.6: Object rest/spread (...)

Version > 7

Object spreading is just syntactic sugar for `Object.assign({}, obj1, ..., objn);`

It is done with the `...` operator:

```
let obj = { a: 1 };

let obj2 = { ...obj, b: 2, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 };
```

As `Object.assign` it does **shallow** merging, not deep merging.

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

NOTE: This specification is currently in stage 3

Section 13.7: Object.defineProperty

Version ≥ 5

It allows us to define a property in an existing object using a property descriptor.

```
var obj = { };

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

Console output

> foo

`Object.defineProperty` can be called with the following options:

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // if false, the property is read-only
  configurable : true, // true means the property can be changed later
  enumerable : true // true means property can be enumerated such as in a for..in loop
});
```

`Object.defineProperties` allows you to define multiple properties at a time.

```
var obj = {};
```

```
Object.defineProperties(obj, {
  property1: {
    value: true,
    writable: true
  },
  property2: {
    value: 'Hello',
    writable: false
  }
});
```

Section 13.8: Accesor properties (get and set)

Version ≥ 5

Treat a property as a combination of two functions, one to get the value from it, and another one to set the value in it.

The `get` property of the property descriptor is a function that will be called to retrieve the value from the property.

The `set` property is also a function, it will be called when the property has been assigned a value, and the new value will be passed as an argument.

You cannot assign a `value` or `writable` to a descriptor that has `get` or `set`

```
var person = { name: "John", surname: "Doe"};
Object.defineProperty(person, 'fullName', {
    get: function () {
        return this.name + " " + this.surname;
    },
    set: function (value) {
        [this.name, this.surname] = value.split(" ");
    }
});

console.log(person.fullName); // -> "John Doe"

person.surname = "Hill";
console.log(person.fullName); // -> "John Hill"

person.fullName = "Mary Jones";
console.log(person.name) // -> "Mary"
```

Section 13.9: Dynamic / variable property names

Sometimes the property name needs to be stored into a variable. In this example, we ask the user what word needs to be looked up, and then provide the result from an object I've named `dictionary`.

```
var dictionary = {
    lettuce: 'a veggie',
    banana: 'a fruit',
    tomato: 'it depends on who you ask',
    apple: 'a fruit',
    Apple: 'Steve Jobs rocks!' // properties are case-sensitive
}

var word = prompt('What word would you like to look up today?')
var definition = dictionary[word]
```

```
alert(word + '\n\n' + definition)
```

Note how we are using [ ] bracket notation to look at the variable named `word`; if we were to use the traditional `.` notation, then it would take the value literally, hence:

```
console.log(dictionary.word)  // doesn't work because word is taken literally and dictionary has no field named `word`
console.log(dictionary.apple) // it works! because apple is taken literally

console.log(dictionary[word]) // it works! because word is a variable, and the user perfectly typed in one of the words from our dictionary when prompted
console.log(dictionary[apple]) // error! apple is not defined (as a variable)
```

You could also write literal values with [ ] notation by replacing the variable `word` with a string `'apple'`. See [Properties with special characters or reserved words] example.

You can also set dynamic properties with the bracket syntax:

```
var property="test";
var obj={
 [property]=1;
};

console.log(obj.test);//1
```

It does the same as:

```
var property="test";
var obj={};
obj[property]=1;
```

Section 13.10: Arrays are Objects

> **Disclaimer:** Creating array-like objects is not recommend. However, it is helpful to understand how they work, especially when working with DOM. This will explain why regular array operations don't work on DOM objects returned from many DOM `document` functions. (i.e. `querySelectorAll`, `form.elements`)

Supposing we created the following object which has some properties you would expect to see in an Array.

```
var anObject = {
    foo: 'bar',
    length: 'interesting',
    '0': 'zero!',
    '1': 'one!'
};
```

Then we'll create an array.

```
var anArray = ['zero.', 'one.'];
```

Now, notice how we can inspect both the object, and the array in the same way.

```
console.log(anArray[0], anObject[0]); // outputs: zero.  zero!
console.log(anArray[1], anObject[1]); // outputs: one.  one!
console.log(anArray.length, anObject.length); // outputs: 2 interesting
```

```
console.log(anArray.foo, anObject.foo); // outputs: undefined bar
```

Since `anArray` is actually an object, just like `anObject`, we can even add custom wordy properties to `anArray`

> **Disclaimer:** Arrays with custom properties are not usually recommended as they can be confusing, but it can be useful in advanced cases where you need the optimized functions of an Array. (i.e. jQuery objects)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

We can even make `anObject` to be an array-like object by adding a `length`.

```
anObject.length = 2;
```

Then you can use the C-style **`for`** loop to iterate over `anObject` just as if it were an Array. See Array Iteration

Note that `anObject` is only an **array-like** object. (also known as a List) It is not a true Array. This is important, because functions like `push` and `forEach` (or any convenience function found in `Array.`**`prototype`**) will not work by default on array-like objects.

Many of the DOM `document` functions will return a List (i.e. `querySelectorAll`, `form.elements`) which is similar to the array-like `anObject` we created above. See Converting Array-like Objects to Arrays

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // outputs: true  true
console.log(anArray instanceof Object, anObject instanceof Object); // outputs: true  true
console.log(anArray instanceof Array, anObject instanceof Array); // outputs: true  false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // outputs: true  false
```

Section 13.11: Object.seal

Version ≥ 5

`Object.seal` prevents the addition or removal of properties from an object. Once an object has been sealed its property descriptors can't be converted to another type. Unlike `Object.freeze` it does allow properties to be edited.

Attempts to do this operations on a sealed object will fail silently

```
var obj = { foo: 'foo', bar: function () { return 'bar'; } };

Object.seal(obj)

obj.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'

// Can't make foo an accessor property
Object.defineProperty(obj, 'foo', {
    get: function () { return 'newFoo'; }
}); // TypeError

// But you can make it read only
Object.defineProperty(obj, 'foo', {
```

```
    writable: false
}); // TypeError

obj.foo = 'newFoo';
obj.foo; // 'foo';
```

In strict mode these operations will throw a `TypeError`

```
(function () {
    'use strict';

    var obj = { foo: 'foo' };

    Object.seal(obj);

    obj.newFoo = 'newFoo'; // TypeError
}());
```

Section 13.12: Convert object's values to array

Given this object:

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!",
};
```

You can convert its values to an array by doing:

```
var array = Object.keys(obj)
    .map(function(key) {
        return obj[key];
    });

console.log(array); // ["hello", "this is", "javascript!"]
```

Section 13.13: Retrieving properties from an object

Characteristics of properties :

Properties that can be retrieved from an *object* could have the following characteristics,

- Enumerable
- Non - Enumerable
- own

While creating the properties using *Object.defineProperty(ies)*, we could set its characteristics except *"own"*. Properties which are available in the direct level not in the *prototype* level (`__proto__`) of an object are called as *own* properties.

And the properties that are added into an object without using `Object.defindProperty(ies)` will don't have its enumerable characteristic. That means it be considered as true.

Purpose of enumerability :

The main purpose of setting enumerable characteristics to a property is to make the particular property's

availability when retrieving it from its object, by using different programmatical methods. Those different methods will be discussed deeply below.

Methods of retrieving properties :

Properties from an object could be retrieved by the following methods,

1. `for..in` loop

   This loop is very useful in retrieving enumerable properties from an object. Additionally this loop will retrieve enumerable own properties as well as it will do the same retrieval by traversing through the prototype chain until it sees the prototype as null.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});

for(prop in x){
  props.push(prop);
}

console.log(props); //["a"]
```

2. `Object.keys()` function

   This function was unveiled as a part of ECMAScript 5. It is used to retrieve enumerable own properties from an object. Prior to its release people used to retrieve own properties from an object by combining `for..in` loop and `Object.prototype.hasOwnProperty()` function.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.keys(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.keys(x);
```

```
console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.keys(x);

console.log(props); //["a"]
```

3. `Object.getOwnProperties()` function

   This function will retrieve both enumerable and non enumerable, own properties from an object. It was also released as a part of ECMAScript 5.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]
```

Miscellaneous :

A technique for retrieving all (own, enumerable, non enumerable, all prototype level) properties from an object is given below,

```
function getAllProperties(obj, props = []){
  return obj == null ? props :
           getAllProperties(Object.getPrototypeOf(obj),
               props.concat(Object.getOwnPropertyNames(obj)));
}

var x = {a:10, __proto__ : { b : 5, c : 15 }};

//adding a non enumerable property to first level prototype
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]
```

And this will be supported by the browsers which supports ECMAScript 5.

Section 13.14: Read-Only property

Version ≥ 5

Using property descriptors we can make a property read only, and any attempt to change its value will fail silently, the value will not be changed and no error will be thrown.

The `writable` property in a property descriptor indicates whether that property can be changed or not.

```
var a  = { };

Object.defineProperty(a, 'foo', { value: 'original', writable: false });

a.foo = 'new';

console.log(a.foo);
```

Console output

> original

Section 13.15: Non enumerable property

Version ≥ 5

We can avoid a property from showing up in `for (... in ...)` loops

The `enumerable` property of the property descriptor tells whether that property will be enumerated while looping through the object's properties.

```
var obj = { };

Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
    console.log(obj[prop]);
}
```

Console output

> show

Section 13.16: Lock property description

Version ≥ 5

A property's descriptor can be locked so no changes can be made to it. It will still be possible to use the property normally, assigning and retrieving the value from it, but any attempt to redefine it will throw an exception.

The `configurable` property of the property descriptor is used to disallow any further changes on the descriptor.

```
var obj = {};

// Define 'foo' as read only and lock it
Object.defineProperty(obj, "foo", {
    value: "original value",
    writable: false,
    configurable: false
});

Object.defineProperty(obj, "foo", {writable: true});
```

This error will be thrown:

> TypeError: Cannot redefine property: foo

And the property will still be read only.

```
obj.foo = "new value";
console.log(foo);
```

Console output

> original value

Section 13.17: Object.getOwnPropertyDescriptor

Get the description of a specific property in an object.

```
var sampleObject = {
    hello: 'world'
};

Object.getOwnPropertyDescriptor(sampleObject, 'hello');
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

Section 13.18: Descriptors and Named Properties

Properties are members of an object. Each named property is a pair of (name, descriptor). The name is a string that allows access (using the dot notation `object.propertyName` or the square brackets notation `object['propertyName']`). The descriptor is a record of fields defining the bevahiour of the property when it is accessed (what happens to the property and what is the value returned from accessing it). By and large, a property associates a name to a behavior (we can think of the behavior as a black box).

There are two types of named properties:

1. *data property*: the property's name is associated with a value.
2. *accessor property*: the property's name is associated with one or two accessor functions.

Demonstration:

```
obj.propertyName1 = 5; //translates behind the scenes into
                       //either assigning 5 to the value field* if it is a data property
               //or calling the set function with the parameter 5 if accessor property
```

```
//*actually whether an assignment would take place in the case of a data property
//also depends on the presence and value of the writable field - on that later on
```

The property's type is determined by its descriptor's fields, and a property cannot be of both types.

Data descriptors -

- Required fields: `value` or `writable` or both
- Optional fields:`configurable`,`enumerable`

Sample:

```
{
   value: 10,
   writable: true;
}
```

Accessor descriptors -

- Required fields: **`get`** or **`set`** or both
- Optional fields: `configurable`, `enumerable`

Sample:

```
{
    get: function () {
        return 10;
    },
    enumerable: true
}
```

meaning of fields and their defaults

`configurable`,`enumerable` and `writable`:

- These keys all default to **`false`**.
- `configurable` is **`true`** if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.
- `enumerable` is **`true`** if and only if this property shows up during enumeration of the properties on the corresponding object.
- `writable` is **`true`** if and only if the value associated with the property may be changed with an assignment operator.

`get` and **`set`**:

- These keys default to **`undefined`**.
- **`get`** is a function which serves as a getter for the property, or **`undefined`** if there is no getter. The function return will be used as the value of the property.
- **`set`** is a function which serves as a setter for the property, or **`undefined`** if there is no setter. The function will receive as only argument the new value being assigned to the property.

`value`:

- This key defaults to **`undefined`**.
- The value associated with the property. Can be any valid JavaScript value (number, object, function, etc).

Example:

```
    var obj = {propertyName1: 1}; //the pair is actually ('propertyName1', {value:1,
                                                                    // writable:true,
                                                                    // enumerable:true,
                                                                    // configurable:true})
    Object.defineProperty(obj, 'propertyName2', {get: function() {
                                                    console.log('this will be logged ' +
                                    'every time propertyName2 is accessed to get its value');
                                                },
                                            set: function() {
                                                    console.log('and this will be logged ' +
                                    'every time propertyName2\'s value is tried to be set')
                        //will be treated like it has enumerable:false, configurable:false
                                                }});
//propertyName1 is the name of obj's data property
//and propertyName2 is the name of its accessor property




obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

obj.propertyName2 = 3; //and this will be logged every time propertyName2's value is tried to be set
console.log(obj.propertyName2); //this will be logged every time propertyName2 is accessed to get
its value
```

Section 13.19: Object.keys

Version ≥ 5

`Object.keys(obj)` returns an array of a given object's keys.

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]
```

Section 13.20: Properties with special characters or reserved words

While object property notation is usually written as `myObject.property`, this will only allow characters that are normally found in JavaScript variable names, which is mainly letters, numbers and underscore (_).

If you need special characters, such as space, ☺, or user-provided content, this is possible using [ ] bracket notation.

```
myObject['special property ☺'] = 'it works!'
console.log(myObject['special property ☺'])
```

All-digit properties:

In addition to special characters, property names that are all-digits will require bracket notation. However, in this case the property need not be written as a string.

```
myObject[123] = 'hi!' // number 123 is automatically converted to a string
console.log(myObject['123']) // notice how using string 123 produced the same result
console.log(myObject['12' + '3']) // string concatenation
console.log(myObject[120 + 3]) // arithmetic, still resulting in 123 and producing the same result
console.log(myObject[123.0]) // this works too because 123.0 evaluates to 123
console.log(myObject['123.0']) // this does NOT work, because '123' != '123.0'
```

However, leading zeros are not recommended as that is interpreted as Octal notation. (TODO, we should produce and link to an example describing octal, hexadecimal and exponent notation)

See also: [Arrays are Objects] example.

Section 13.21: Creating an Iterable object

Version ≥ 6

```
var myIterableObject = {};
// An Iterable object must define a method located at the Symbol.iterator key:
myIterableObject[Symbol.iterator] = function () {
  // The iterator should return an Iterator object
  return {
    // The Iterator object must implement a method, next()
    next: function () {
      // next must itself return an IteratorResult object
      if (!this.iterated) {
        this.iterated = true;
        // The IteratorResult object has two properties
        return {
          // whether the iteration is complete, and
          done: false,
          // the value of the current iteration
          value: 'One'
        };
      }
      return {
        // When iteration is complete, just the done property is needed
        done: true
      };
    },
    iterated: false
  };
};

for (var c of myIterableObject) {
  console.log(c);
}
```

Console output

> One

Section 13.22: Iterating over Object entries - Object.entries()

Version ≥ 8

The proposed `Object.entries()` method returns an array of key/value pairs for the given object. It does not return an iterator like `Array.prototype.entries()`, but the Array returned by `Object.entries()` can be iterated

regardless.

```
const obj = {
    one: 1,
    two: 2,
    three: 3
};

Object.entries(obj);
```

Results in:

```
[
    ["one", 1],
    ["two", 2],
    ["three", 3]
]
```

It is an useful way of iterating over the key/value pairs of an object:

```
for(const [key, value] of Object.entries(obj)) {
    console.log(key); // "one", "two" and "three"
    console.log(value); // 1, 2 and 3
}
```

Section 13.23: Object.values()

Version ≥ 8

The `Object.values()` method returns an array of a given object's own enumerable property values, in the same order as that provided by a for...in loop (the difference being that a for-in loop enumerates properties in the prototype chain as well).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Note:

For browser support, please refer to this link