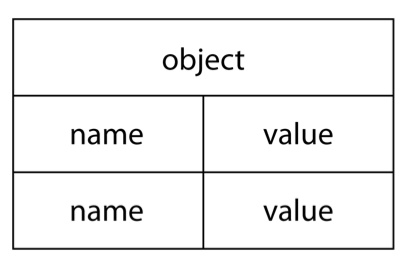
# **Objects**

## **Create object**

****

| **const myObj = {  key: value  *// ...* };** |
| --- |

| **var myObj = new Object(); myObj.key = value;** |
| --- |

The constructed form and the literal form result in exactly the same sort of object. The only difference really is that you can add one or more key/value pairs to the literal declaration, whereas with constructed-form objects, you must add the properties one-by-one.

**Note:** It's extremely uncommon to use the "constructed form" for creating objects as just shown. You would pretty much always want to use the literal syntax form. The same will be true of most of the built-in objects (see below).

In objects, property names are always strings. If you use any other value besides a string (primitive) as the property, it will first be converted to a string. This even includes numbers, which are commonly used as array indexes, so be careful not to confuse the use of numbers between objects and arrays.

| *const myObject = { };  myObject[true] = "foo"; myObject[3] = "bar"; myObject[myObject] = "baz"; myObject.name = "baz name";  myObject["true"]; // "foo" myObject["3"]; // "bar" myObject["[object Object]"]; // "baz" myObject.name; // "baz name"* |
| --- |

## **Computed Property Names**

The myObject[..] property access syntax we just described is useful if you need to use a computed expression value *as* the key name, like myObject[prefix + name]. But that's not really helpful when declaring objects using the object-literal syntax.

ES6 adds *computed property names*, where you can specify an expression, surrounded by a [ ] pair, in the key-name position of an object-literal declaration:

| const prefix = "foo";  const myObject = {  [prefix + "bar"]: "hello",  [prefix + "baz"]: "world" };  myObject["foobar"]; *// hello* myObject["foobaz"]; *// world* |
| --- |

## JavaScript Nested Objects

An object can also contain another object. For example

| *// nested object* const student = {  name: 'John',  age: 20,  marks: {  science: 70,  math: 75  } }  *// accessing property of student object* console.log(student.marks); *// {science: 70, math: 75}*  *// accessing property of marks object* console.log(student.marks.science); *// 70* |
| --- |

## Methods and this

In JavaScript, objects can also contain functions and it called object methods.

| *// object containing method* const person = {  name: 'John',  greet: function() { console.log('hello'); } }; |
| --- |

You can access an object method using a dot notation. The syntax is:

| objectName.methodKey() |
| --- |

You can access property by calling an objectName and a key. You can access a method by calling an objectName and a key for that method along with (). For example.

| *// accessing method and property* const person = {  name: 'John',  greet: function() { console.log('hello'); } };  *// accessing property* person.name; *// John*  *// accessing method* person.greet(); *// hello* |
| --- |

You can also add a method in an object. For example:

| *// creating an object* let student = { };  *// adding a property* student.name = 'John';  *// adding a method* student.greet = function() {  console.log('hello'); }  *// accessing a method* student.greet(); *// hello* |
| --- |

To access a property of an object from within a method of the same object, you need to use the this keyword. Let's consider an example.

| const person = {  name: 'John',  age: 30,   *// accessing name property by using this.name*  greet: function() { console.log('The name is' + ' ' + this.name); } };  person.greet(); *// The name is John* |
| --- |

## Object constructor

In JavaScript, you can create multiple objects from a constructor function. For example:

| *// constructor function* function Person () {  this.name = 'John',  this.age = 23,   this.greet = function () {  console.log('hello');  } }  *// create objects* const person1 = new Person(); const person2 = new Person();  *// access properties* console.log(person1.name); *// John* console.log(person2.name); *// John* |
| --- |

You can also create a constructor function with parameters. For example:

| *// constructor function* function Person (person\_name, person\_age, person\_gender) {   *// assigning parameter values to the calling object*  this.name = person\_name,  this.age = person\_age,  this.gender = person\_gender,   this.greet = function () {  return ('Hi' + ' ' + this.name);  } }   *// creating objects* const person1 = new Person('John', 23, 'male'); const person2 = new Person('Sam', 25, 'female');  *// accessing properties* console.log(person1.name); *// "John"* console.log(person2.name); *// "Sam"* |
| --- |

## Object Prototype

In JavaScript, a prototype can be used to add properties and methods to a constructor function. And objects inherit properties and methods from a prototype.

For example:

| *// constructor function* function Person () {  this.name = 'John',  this.age = 23 }  *// creating objects* let person1 = new Person(); let person2 = new Person();  *// adding new property to constructor function* Person.prototype.gender = 'Male';  console.log(person1.gender); *// Male* console.log(person2.gender); *// Male* |
| --- |

| *// constructor function* function Person () {  this.name = 'John',  this.age = 23 }  *// creating objects* const person1 = new Person(); const person2 = new Person();  *// adding property to constructor function* Person.prototype.gender = 'male';  *// prototype value of Person* console.log(Person.prototype);  *// inheriting the property from prototype* console.log(person1.gender); console.log(person2.gender); |
| --- |

In JavaScript, every function and object has a property named prototype by default. For example:

| function Person () {  this.name = 'John',  this.age = 23 }  const person = new Person();  *// checking the prototype value* console.log(Person.prototype); *// { ... }* |
| --- |

In the above example, we are trying to access the prototype property of a Person constructor function.

Since the prototype property has no value at the moment, it shows an empty object { ... }

**More about prototypal inheritance:**

<https://medium.com/@raviroshan.talk/javascript-prototypal-inheritance-f3a58cfb1179>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain>

## Object Getters and Setters

In JavaScript, **getter** methods are used to access the properties of an object. For example:

| const student = {   *// data property*  firstName: 'Monica',    *// accessor property(getter)*  get getName() {  return this.firstName;  } };  *// accessing data property* console.log(student.firstName); *// Monica*  *// accessing getter methods* console.log(student.getName); *// Monica*  *// trying to access as a method* console.log(student.getName()); *// error* |
| --- |

In JavaScript, setter methods are used to change the values of an object. For example:

| const student = {  firstName: 'Monica',    *//accessor property(setter)*  set changeName(newName) {  this.firstName = newName;  } };  console.log(student.firstName); *// Monica*  *// change(set) object property using a setter* student.changeName = 'Sarah';  console.log(student.firstName); *// Sarah* |
| --- |

## JavaScript Object.defineProperty()

In JavaScript, you can also use the Object.defineProperty() method to add getters and setters. For example:

| const student = {  firstName: 'Monica' }  *// getting property* Object.defineProperty(student, "getName", {  get : function () {  return this.firstName;  } });  *// setting property* Object.defineProperty(student, "changeName", {  set : function (value) {  this.firstName = value;  } });  console.log(student.firstName); *// Monica*  *// changing the property value* student.changeName = 'Sarah';  console.log(student.firstName); *// Sarah* |
| --- |

## Object shallow cloning

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 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().

| const existing = { a: 1, b: 2, c: 3 }; const clone = {}; for (const prop in existing) { if (existing.hasOwnProperty(prop)) { clone[prop] = existing[prop]; } } |
| --- |

More information about cloning:

<https://www.samanthaming.com/tidbits/70-3-ways-to-clone-objects/>

## Object deep cloning

| const existing = { a: 1, b: { c: 2 } }; const 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, []); } |
| --- |

## Object properties iteration

| for (let property in object) { *// always check if an object has a property if (object.hasOwnProperty(property)) {* *// do stuff* } } |
| --- |

| const obj = { 0: 'a', 1: 'b', 2: 'c' }; Object.keys(obj).map(function(key) { console.log(key); }); *// outputs: 0, 1, 2* |
| --- |

## Object.freeze

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.

| const 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'; |
| --- |

## 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:

| const user = { firstName: "John" }; Object.assign(user, {lastName: "Doe", age:39}); console.log(user); *// Logs: {firstName: "John", lastName: "Doe", age: 39}* |
| --- |

| const obj1 = { a: 1 }; const obj2 = {b: 2 }; const obj3 = { c: 3 }; const 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

| const var\_1 = 'abc'; const var\_2 = true; const var\_3 = 10; const var\_4 = Symbol('foo'); const obj = Object.assign({}, var\_1, null, var\_2, undefined, var\_3, var\_4); console.log(obj); *// Logs: { "0": "a", "1": "b", "2": "c" }* |
| --- |

## Object rest/spread (...)

| Object spreading is just syntactic sugar for Object.assign({}, obj1, ..., 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 } };* |
| --- |