# Working with objects

* [« Previous](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Keyed_collections)
* [Next »](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes)

JavaScript is designed on a simple object-based paradigm. An object is a collection of [properties](https://developer.mozilla.org/en-US/docs/Glossary/property/JavaScript), and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a [method](https://developer.mozilla.org/en-US/docs/Glossary/Method).

Objects in JavaScript, just as in many other programming languages, can be compared to objects in real life. In JavaScript, an object is a standalone entity, with properties and type. Compare it with a cup, for example. A cup is an object, with properties. A cup has a color, a design, weight, a material it is made of, etc. The same way, JavaScript objects can have properties, which define their characteristics.

In addition to objects that are predefined in the browser, you can define your own objects. This chapter describes how to use objects, properties, and methods, and how to create your own objects.

## [Creating new objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#creating_new_objects)

You can create an object using an [object initializer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer). Alternatively, you can first create a constructor function and then instantiate an object by invoking that function with the new operator.

### [Using object initializers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_object_initializers)

Object initializers are also called object literals. "Object initializer" is consistent with the terminology used by C++。

The syntax for an object using an object initializer is:

const obj = {

property1: value1, // property name may be an identifier

2: value2, // or a number

"property n": value3, // or a string

};

Copy to Clipboard

Each property name before colons is an identifier (either a name, a number, or a string literal), and each valueN is an expression whose value is assigned to the property name. The property name can also be an expression; computed keys need to be wrapped in square brackets. The [object initializer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer) reference contains a more detailed explanation of the syntax.

In this example, the newly created object is assigned to a variable obj — this is optional. If you do not need to refer to this object elsewhere, you do not need to assign it to a variable. (Note that you may need to wrap the object literal in parentheses if the object appears where a statement is expected, so as not to have the literal be confused with a block statement.)

Object initializers are expressions, and each object initializer results in a new object being created whenever the statement in which it appears is executed. Identical object initializers create distinct objects that do not compare to each other as equal.

The following statement creates an object and assigns it to the variable x if and only if the expression cond is true:

let x;

if (cond) {

x = { greeting: "hi there" };

}

Copy to Clipboard

The following example creates myHonda with three properties. Note that the engine property is also an object with its own properties.

const myHonda = {

color: "red",

wheels: 4,

engine: { cylinders: 4, size: 2.2 },

};

Copy to Clipboard

Objects created with initializers are called plain objects, because they are instances of [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object), but not any other object type. Some object types have special initializer syntaxes — for example, [array initializers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#array_literals) and [regex literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#creating_a_regular_expression).

### [Using a constructor function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_a_constructor_function)

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
2. Create an instance of the object with [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new).

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called Car, and you want it to have properties for make, model, and year. To do this, you would write the following function:

function Car(make, model, year) {

this.make = make;

this.model = model;

this.year = year;

}

Copy to Clipboard

Notice the use of this to assign values to the object's properties based on the values passed to the function.

Now you can create an object called myCar as follows:

const myCar = new Car("Eagle", "Talon TSi", 1993);

Copy to Clipboard

This statement creates myCar and assigns it the specified values for its properties. Then the value of myCar.make is the string "Eagle", myCar.model is the string "Talon TSi", myCar.year is the integer 1993, and so on. The order of arguments and parameters should be the same.

You can create any number of Car objects by calls to new. For example,

const kenscar = new Car("Nissan", "300ZX", 1992);

const vpgscar = new Car("Mazda", "Miata", 1990);

Copy to Clipboard

An object can have a property that is itself another object. For example, suppose you define an object called Person as follows:

function Person(name, age, sex) {

this.name = name;

this.age = age;

this.sex = sex;

}

Copy to Clipboard

and then instantiate two new Person objects as follows:

const rand = new Person("Rand McKinnon", 33, "M");

const ken = new Person("Ken Jones", 39, "M");

Copy to Clipboard

Then, you can rewrite the definition of Car to include an owner property that takes a Person object, as follows:

function Car(make, model, year, owner) {

this.make = make;

this.model = model;

this.year = year;

this.owner = owner;

}

Copy to Clipboard

To instantiate the new objects, you then use the following:

const car1 = new Car("Eagle", "Talon TSi", 1993, rand);

const car2 = new Car("Nissan", "300ZX", 1992, ken);

Copy to Clipboard

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects rand and ken as the arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

car2.owner.name;

Copy to Clipboard

You can always add a property to a previously defined object. For example, the statement

car1.color = "black";

Copy to Clipboard

adds a property color to car1, and assigns it a value of "black". However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the Car object type.

You can also use the [class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) syntax instead of the function syntax to define a constructor function. For more information, see the [class guide](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes).

### [Using the Object.create() method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_the_object.create_method)

Objects can also be created using the [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) method. This method can be very useful, because it allows you to choose the [prototype](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain) object for the object you want to create, without having to define a constructor function.

// Animal properties and method encapsulation

const Animal = {

type: "Invertebrates", // Default value of properties

displayType() {

// Method which will display type of Animal

console.log(this.type);

},

};

// Create new animal type called animal1

const animal1 = Object.create(Animal);

animal1.displayType(); // Logs: Invertebrates

// Create new animal type called fish

const fish = Object.create(Animal);

fish.type = "Fishes";

fish.displayType(); // Logs: Fishes

Copy to Clipboard

## [Objects and properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#objects_and_properties)

A JavaScript object has properties associated with it. Object properties are basically the same as variables, except that they are associated with objects, not [scopes](https://developer.mozilla.org/en-US/docs/Glossary/Scope). The properties of an object define the characteristics of the object.

For example, this example creates an object named myCar, with properties named make, model, and year, with their values set to "Ford", "Mustang", and 1969:

const myCar = {

make: "Ford",

model: "Mustang",

year: 1969,

};

Copy to Clipboard

Like JavaScript variables, property names are case sensitive. Property names can only be strings or Symbols — all keys are [converted to strings](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#string_coercion) unless they are Symbols. [Array indices](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#array_indices) are, in fact, properties with string keys that contain integers.

### [Accessing properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#accessing_properties)

You can access a property of an object by its property name. [Property accessors](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors) come in two syntaxes: dot notation and bracket notation. For example, you could access the properties of the myCar object as follows:

// Dot notation

myCar.make = "Ford";

myCar.model = "Mustang";

myCar.year = 1969;

// Bracket notation

myCar["make"] = "Ford";

myCar["model"] = "Mustang";

myCar["year"] = 1969;

Copy to Clipboard

An object property name can be any JavaScript string or [symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol), including an empty string. However, you cannot use dot notation to access a property whose name is not a valid JavaScript identifier. For example, a property name that has a space or a hyphen, that starts with a number, or that is held inside a variable can only be accessed using the bracket notation. This notation is also very useful when property names are to be dynamically determined, i.e. not determinable until runtime. Examples are as follows:

const myObj = {};

const str = "myString";

const rand = Math.random();

const anotherObj = {};

// Create additional properties on myObj

myObj.type = "Dot syntax for a key named type";

myObj["date created"] = "This key has a space";

myObj[str] = "This key is in variable str";

myObj[rand] = "A random number is the key here";

myObj[anotherObj] = "This key is object anotherObj";

myObj[""] = "This key is an empty string";

console.log(myObj);

// {

// type: 'Dot syntax for a key named type',

// 'date created': 'This key has a space',

// myString: 'This key is in variable str',

// '0.6398914448618778': 'A random number is the key here',

// '[object Object]': 'This key is object anotherObj',

// '': 'This key is an empty string'

// }

console.log(myObj.myString); // 'This key is in variable str'

Copy to Clipboard

In the above code, the key anotherObj is an object, which is neither a string nor a symbol. When it is added to the myObj, JavaScript calls the [toString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/toString) method of anotherObj, and use the resulting string as the new key.

You can also access properties with a string value stored in a variable. The variable must be passed in bracket notation. In the example above, the variable str held "myString" and it is "myString" that is the property name. Therefore, myObj.str will return as undefined.

str = "myString";

myObj[str] = "This key is in variable str";

console.log(myObj.str); // undefined

console.log(myObj[str]); // 'This key is in variable str'

console.log(myObj.myString); // 'This key is in variable str'

Copy to Clipboard

This allows accessing any property as determined at runtime:

let propertyName = "make";

myCar[propertyName] = "Ford";

// access different properties by changing the contents of the variable

propertyName = "model";

myCar[propertyName] = "Mustang";

console.log(myCar); // { make: 'Ford', model: 'Mustang' }

Copy to Clipboard

However, beware of using square brackets to access properties whose names are given by external input. This may make your code susceptible to [object injection attacks](https://github.com/nodesecurity/eslint-plugin-security/blob/main/docs/the-dangers-of-square-bracket-notation.md).

Nonexistent properties of an object have value [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) (and not [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null)).

myCar.nonexistentProperty; // undefined

Copy to Clipboard

### [Enumerating properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#enumerating_properties)

There are three native ways to list/traverse object properties:

* [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loops. This method traverses all of the enumerable string properties of an object as well as its prototype chain.
* [Object.keys(myObj)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys). This method returns an array with only the enumerable own string property names ("keys") in the object myObj, but not those in the prototype chain.
* [Object.getOwnPropertyNames(myObj)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames). This method returns an array containing all the own string property names in the object myObj, regardless of if they are enumerable or not.

You can use the bracket notation with [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) to iterate over all the enumerable properties of an object. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

function showProps(obj, objName) {

let result = "";

for (const i in obj) {

// Object.hasOwn() is used to exclude properties from the object's

// prototype chain and only show "own properties"

if (Object.hasOwn(obj, i)) {

result += `${objName}.${i} = ${obj[i]}\n`;

}

}

console.log(result);

}

Copy to Clipboard

The term "own property" refers to the properties of the object, but excluding those of the prototype chain. So, the function call showProps(myCar, 'myCar') would print the following:

myCar.make = Ford

myCar.model = Mustang

myCar.year = 1969

The above is equivalent to:

function showProps(obj, objName) {

let result = "";

Object.keys(obj).forEach((i) => {

result += `${objName}.${i} = ${obj[i]}\n`;

});

console.log(result);

}

Copy to Clipboard

There is no native way to list inherited non-enumerable properties. However, this can be achieved with the following function:

function listAllProperties(myObj) {

let objectToInspect = myObj;

let result = [];

while (objectToInspect !== null) {

result = result.concat(Object.getOwnPropertyNames(objectToInspect));

objectToInspect = Object.getPrototypeOf(objectToInspect);

}

return result;

}

Copy to Clipboard

For more information, see [Enumerability and ownership of properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties).

### [Deleting properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#deleting_properties)

You can remove a non-inherited property using the [delete](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) operator. The following code shows how to remove a property.

// Creates a new object, myobj, with two properties, a and b.

const myobj = new Object();

myobj.a = 5;

myobj.b = 12;

// Removes the a property, leaving myobj with only the b property.

delete myobj.a;

console.log("a" in myobj); // false

Copy to Clipboard

## [Inheritance](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#inheritance)

All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor. See [Inheritance and the prototype chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain) for more information.

### [Defining properties for all objects of one type](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_properties_for_all_objects_of_one_type)

You can add a property to all objects created through a certain [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_a_constructor_function) using the [prototype](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/prototype) property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a color property to all objects of type Car, and then reads the property's value from an instance car1.

Car.prototype.color = "red";

console.log(car1.color); // "red"

Copy to Clipboard

## [Defining methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_methods)

A method is a function associated with an object, or, put differently, a method is a property of an object that is a function. Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object. See also [method definitions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions) for more details. An example is:

objectName.methodName = functionName;

const myObj = {

myMethod: function (params) {

// do something

},

// this works too!

myOtherMethod(params) {

// do something else

},

};

Copy to Clipboard

where objectName is an existing object, methodName is the name you are assigning to the method, and functionName is the name of the function.

You can then call the method in the context of the object as follows:

object.methodName(params);

Copy to Clipboard

Methods are typically defined on the prototype object of the constructor, so that all objects of the same type share the same method. For example, you can define a function that formats and displays the properties of the previously-defined Car objects.

Car.prototype.displayCar = function () {

const result = `A Beautiful ${this.year} ${this.make} ${this.model}`;

console.log(result);

};

Copy to Clipboard

Notice the use of this to refer to the object to which the method belongs. Then you can call the displayCar method for each of the objects as follows:

car1.displayCar();

car2.displayCar();

Copy to Clipboard

### [Using this for object references](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_this_for_object_references)

JavaScript has a special keyword, [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), that you can use within a method to refer to the current object. For example, suppose you have 2 objects, Manager and Intern. Each object has its own name, age and job. In the function sayHi(), notice the use of this.name. When added to the 2 objects, the same function will print the message with the name of the respective object it's attached to.

const Manager = {

name: "Karina",

age: 27,

job: "Software Engineer",

};

const Intern = {

name: "Tyrone",

age: 21,

job: "Software Engineer Intern",

};

function sayHi() {

console.log(`Hello, my name is ${this.name}`);

}

// add sayHi function to both objects

Manager.sayHi = sayHi;

Intern.sayHi = sayHi;

Manager.sayHi(); // Hello, my name is Karina

Intern.sayHi(); // Hello, my name is Tyrone

Copy to Clipboard

this is a "hidden parameter" of a function call that's passed in by specifying the object before the function that was called. For example, in Manager.sayHi(), this is the Manager object, because Manager comes before the function sayHi(). If you access the same function from another object, this will change as well. If you use other methods to call the function, like [Function.prototype.call()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call) or [Reflect.apply()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect/apply), you can explicitly pass the value of this as an argument.

## [Defining getters and setters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_getters_and_setters)

A [getter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get) is a function associated with a property that gets the value of a specific property. A [setter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set) is a function associated with a property that sets the value of a specific property. Together, they can indirectly represent the value of a property.

Getters and setters can be either

* defined within [object initializers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_object_initializers), or
* added later to any existing object.

Within [object initializers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_object_initializers), getters and setters are defined like regular [methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions), but prefixed with the keywords get or set. The getter method must not expect a parameter, while the setter method expects exactly one parameter (the new value to set). For instance:

const myObj = {

a: 7,

get b() {

return this.a + 1;

},

set c(x) {

this.a = x / 2;

},

};

console.log(myObj.a); // 7

console.log(myObj.b); // 8, returned from the get b() method

myObj.c = 50; // Calls the set c(x) method

console.log(myObj.a); // 25

Copy to Clipboard

The myObj object's properties are:

* myObj.a — a number
* myObj.b — a getter that returns myObj.a plus 1
* myObj.c — a setter that sets the value of myObj.a to half of the value myObj.c is being set to

Getters and setters can also be added to an object at any time after creation using the [Object.defineProperties()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties) method. This method's first parameter is the object on which you want to define the getter or setter. The second parameter is an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions. Here's an example that defines the same getter and setter used in the previous example:

const myObj = { a: 0 };

Object.defineProperties(myObj, {

b: {

get() {

return this.a + 1;

},

},

c: {

set(x) {

this.a = x / 2;

},

},

});

myObj.c = 10; // Runs the setter, which assigns 10 / 2 (5) to the 'a' property

console.log(myObj.b); // Runs the getter, which yields a + 1 or 6

Copy to Clipboard

Which of the two forms to choose depends on your programming style and task at hand. If you can change the definition of the original object, you will probably define getters and setters through the original initializer. This form is more compact and natural. However, if you need to add getters and setters later — maybe because you did not write the particular object — then the second form is the only possible form. The second form better represents the dynamic nature of JavaScript, but it can make the code hard to read and understand.

## [Comparing objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#comparing_objects)

In JavaScript, objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

// Two variables, two distinct objects with the same properties

const fruit = { name: "apple" };

const fruitbear = { name: "apple" };

fruit == fruitbear; // return false

fruit === fruitbear; // return false

Copy to Clipboard

// Two variables, a single object

const fruit = { name: "apple" };

const fruitbear = fruit; // Assign fruit object reference to fruitbear

// Here fruit and fruitbear are pointing to same object

fruit == fruitbear; // return true

fruit === fruitbear; // return true

fruit.name = "grape";

console.log(fruitbear); // { name: "grape" }; not { name: "apple" }

Copy to Clipboard

For more information about comparison operators, see [equality operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#equality_operators).

**Object.create() & Object.keys() & Object.values() a lot more**

**Object.create()**

The **Object.create()** method creates a new object, using an existing object as the prototype of the newly created object.

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#syntax)

Object.create(proto)

Object.create(proto, propertiesObject)

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#parameters)

proto

The object which should be the prototype of the newly-created object.

propertiesObject Optional

If specified and not [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined), an object whose [enumerable own properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties) specify property descriptors to be added to the newly-created object, with the corresponding property names. These properties correspond to the second argument of [Object.defineProperties()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties).

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#return_value)

A new object with the specified prototype object and properties.

[**Exceptions**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#exceptions)

[TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError)

Thrown if proto is neither [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null) nor an [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object).

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#examples)

[**Classical inheritance with Object.create()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#classical_inheritance_with_object.create)

Below is an example of how to use Object.create() to achieve classical inheritance. This is for a single inheritance, which is all that JavaScript supports.

// Shape - superclass

function Shape() {

this.x = 0;

this.y = 0;

}

// superclass method

Shape.prototype.move = function (x, y) {

this.x += x;

this.y += y;

console.info("Shape moved.");

};

// Rectangle - subclass

function Rectangle() {

Shape.call(this); // call super constructor.

}

// subclass extends superclass

Rectangle.prototype = Object.create(Shape.prototype, {

// If you don't set Rectangle.prototype.constructor to Rectangle,

// it will take the prototype.constructor of Shape (parent).

// To avoid that, we set the prototype.constructor to Rectangle (child).

constructor: {

value: Rectangle,

enumerable: false,

writable: true,

configurable: true,

},

});

const rect = new Rectangle();

console.log("Is rect an instance of Rectangle?", rect instanceof Rectangle); // true

console.log("Is rect an instance of Shape?", rect instanceof Shape); // true

rect.move(1, 1); // Logs 'Shape moved.'

Copy to Clipboard

Note that there are caveats to watch out for using create(), such as re-adding the [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor) property to ensure proper semantics. Although Object.create() is believed to have better performance than mutating the prototype with [Object.setPrototypeOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf), the difference is in fact negligible if no instances have been created and property accesses haven't been optimized yet. In modern code, the [class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) syntax should be preferred in any case.

[**Using propertiesObject argument with Object.create()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#using_propertiesobject_argument_with_object.create)

Object.create() allows fine-tuned control over the object creation process. The [object initializer syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer) is, in fact, a syntax sugar of Object.create(). With Object.create(), we can create objects with a designated prototype and also some properties. Note that the second parameter maps keys to *property descriptors* — this means you can control each property's enumerability, configurability, etc. as well, which you can't do in object initializers.

o = {};

// Is equivalent to:

o = Object.create(Object.prototype);

o = Object.create(Object.prototype, {

// foo is a regular data property

foo: {

writable: true,

configurable: true,

value: "hello",

},

// bar is an accessor property

bar: {

configurable: false,

get() {

return 10;

},

set(value) {

console.log("Setting `o.bar` to", value);

},

},

});

// Create a new object whose prototype is a new, empty

// object and add a single property 'p', with value 42.

o = Object.create({}, { p: { value: 42 } });

Copy to Clipboard

With Object.create(), we can create an object [with null as prototype](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#null-prototype_objects). The equivalent syntax in object initializers would be the [\_\_proto\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#prototype_setter) key.

o = Object.create(null);

// Is equivalent to:

o = { \_\_proto\_\_: null };

Copy to Clipboard

By default properties are *not* writable, enumerable or configurable.

o.p = 24; // throws in strict mode

o.p; // 42

o.q = 12;

for (const prop in o) {

console.log(prop);

}

// 'q'

delete o.p;

// false; throws in strict mode

Copy to Clipboard

To specify a property with the same attributes as in an initializer, explicitly specify writable, enumerable and configurable.

o2 = Object.create(

{},

{

p: {

value: 42,

writable: true,

enumerable: true,

configurable: true,

},

},

);

// This is not equivalent to:

// o2 = Object.create({ p: 42 })

// which will create an object with prototype { p: 42 }

Copy to Clipboard

You can use Object.create() to mimic the behavior of the [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) operator.

function Constructor() {}

o = new Constructor();

// Is equivalent to:

o = Object.create(Constructor.prototype);

Copy to Clipboard

Of course, if there is actual initialization code in the Constructor function, the Object.create() method cannot reflect it.

**Object.keys()**

The **Object.keys()** method returns an array of a given object's own enumerable string-keyed property names.

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#syntax)

Object.keys(obj)

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#parameters)

obj

An object.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#return_value)

An array of strings representing the given object's own enumerable string-keyed property keys.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#description)

Object.keys() returns an array whose elements are strings corresponding to the enumerable string-keyed property names found directly upon object. This is the same as iterating with a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop, except that a for...in loop enumerates properties in the prototype chain as well. The order of the array returned by Object.keys() is the same as that provided by a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop.

If you need the property values, use [Object.values()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values) instead. If you need both the property keys and values, use [Object.entries()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries) instead.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#examples)

[**Using Object.keys()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#using_object.keys)

// Simple array

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

console.log(Object.keys(arr)); // ['0', '1', '2']

// Array-like object

const obj = { 0: "a", 1: "b", 2: "c" };

console.log(Object.keys(obj)); // ['0', '1', '2']

// Array-like object with random key ordering

const anObj = { 100: "a", 2: "b", 7: "c" };

console.log(Object.keys(anObj)); // ['2', '7', '100']

// getFoo is a non-enumerable property

const myObj = Object.create(

{},

{

getFoo: {

value() {

return this.foo;

},

},

},

);

myObj.foo = 1;

console.log(Object.keys(myObj)); // ['foo']

Copy to Clipboard

If you want *all* string-keyed own properties, including non-enumerable ones, see [Object.getOwnPropertyNames()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames).

[**Using Object.keys() on primitives**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys#using_object.keys_on_primitives)

Non-object arguments are [coerced to objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#object_coercion). Only strings may have own enumerable properties, while all other primitives return an empty array.

// Strings have indices as enumerable own properties

console.log(Object.keys("foo")); // ['0', '1', '2']

// Other primitives have no own properties

console.log(Object.keys(100)); // []

Copy to Clipboard

**Object.values()**

The **Object.values()** method returns an array of a given object's own enumerable string-keyed property values.

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#syntax)

Object.values(obj)

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#parameters)

obj

An object.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#return_value)

An array containing the given object's own enumerable string-keyed property values.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#description)

Object.values() returns an array whose elements are strings corresponding to the enumerable string-keyed property values found directly upon object. This is the same as iterating with a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop, except that a for...in loop enumerates properties in the prototype chain as well. The order of the array returned by Object.values() is the same as that provided by a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop.

If you need the property keys, use [Object.keys()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys) instead. If you need both the property keys and values, use [Object.entries()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries) instead.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#examples)

[**Using Object.values()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#using_object.values)

const obj = { foo: "bar", baz: 42 };

console.log(Object.values(obj)); // ['bar', 42]

// Array-like object

const arrayLikeObj1 = { 0: "a", 1: "b", 2: "c" };

console.log(Object.values(arrayLikeObj1)); // ['a', 'b', 'c']

// Array-like object with random key ordering

// When using numeric keys, the values are returned in the keys' numerical order

const arrayLikeObj2 = { 100: "a", 2: "b", 7: "c" };

console.log(Object.values(arrayLikeObj2)); // ['b', 'c', 'a']

// getFoo is a non-enumerable property

const myObj = Object.create(

{},

{

getFoo: {

value() {

return this.foo;

},

},

},

);

myObj.foo = "bar";

console.log(Object.values(myObj)); // ['bar']

Copy to Clipboard

[**Using Object.values() on primitives**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values#using_object.values_on_primitives)

Non-object arguments are [coerced to objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#object_coercion). Only strings may have own enumerable properties, while all other primitives return an empty array.

// Strings have indices as enumerable own properties

console.log(Object.values("foo")); // ['f', 'o', 'o']

// Other primitives have no own properties

console.log(Object.values(100)); // []

**Object.entries()**

The **Object.entries()** method returns an array of a given object's own enumerable string-keyed property key-value pairs.

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#syntax)

Object.entries(obj)

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#parameters)

obj

An object.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#return_value)

An array of the given object's own enumerable string-keyed property key-value pairs. Each key-value pair is an array with two elements: the first element is the property key (which is always a string), and the second element is the property value.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#description)

Object.entries() returns an array whose elements are arrays corresponding to the enumerable string-keyed property key-value pairs found directly upon object. This is the same as iterating with a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop, except that a for...in loop enumerates properties in the prototype chain as well. The order of the array returned by Object.entries() is the same as that provided by a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop.

If you only need the property keys, use [Object.keys()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys) instead. If you only need the property values, use [Object.values()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values) instead.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#examples)

[**Using Object.entries()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#using_object.entries)

const obj = { foo: "bar", baz: 42 };

console.log(Object.entries(obj)); // [ ['foo', 'bar'], ['baz', 42] ]

// Array-like object

const obj = { 0: "a", 1: "b", 2: "c" };

console.log(Object.entries(obj)); // [ ['0', 'a'], ['1', 'b'], ['2', 'c'] ]

// Array-like object with random key ordering

const anObj = { 100: "a", 2: "b", 7: "c" };

console.log(Object.entries(anObj)); // [ ['2', 'b'], ['7', 'c'], ['100', 'a'] ]

// getFoo is a non-enumerable property

const myObj = Object.create(

{},

{

getFoo: {

value() {

return this.foo;

},

},

},

);

myObj.foo = "bar";

console.log(Object.entries(myObj)); // [ ['foo', 'bar'] ]

Copy to Clipboard

[**Using Object.entries() on primitives**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#using_object.entries_on_primitives)

Non-object arguments are [coerced to objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#object_coercion). Only strings may have own enumerable properties, while all other primitives return an empty array.

// Strings have indices as enumerable own properties

console.log(Object.entries("foo")); // [ ['0', 'f'], ['1', 'o'], ['2', 'o'] ]

// Other primitives have no own properties

console.log(Object.entries(100)); // []

Copy to Clipboard

[**Converting an Object to a Map**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#converting_an_object_to_a_map)

The [Map()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/Map) constructor accepts an iterable of entries. With Object.entries, you can easily convert from [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object) to [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map):

const obj = { foo: "bar", baz: 42 };

const map = new Map(Object.entries(obj));

console.log(map); // Map(2) {"foo" => "bar", "baz" => 42}

Copy to Clipboard

[**Iterating through an Object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries#iterating_through_an_object)

Using [array destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#array_destructuring), you can iterate through objects easily.

// Using for...of loop

const obj = { a: 5, b: 7, c: 9 };

for (const [key, value] of Object.entries(obj)) {

console.log(`${key} ${value}`); // "a 5", "b 7", "c 9"

}

// Using array methods

Object.entries(obj).forEach(([key, value]) => {

console.log(`${key} ${value}`); // "a 5", "b 7", "c 9"

});

Copy to Clipboard

**Object.hasOwn()**

The **Object.hasOwn()** static method returns true if the specified object has the indicated property as its *own* property. If the property is inherited, or does not exist, the method returns false.

**Note:** Object.hasOwn() is intended as a replacement for [Object.prototype.hasOwnProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty).

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#syntax)

hasOwn(instance, prop)

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#parameters)

instance

The JavaScript object instance to test.

prop

The [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) name or [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol) of the property to test.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#return_value)

true if the specified object has directly defined the specified property. Otherwise false

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#description)

The **Object.hasOwn()** method returns true if the specified property is a direct property of the object — even if the property value is null or undefined. The method returns false if the property is inherited, or has not been declared at all. Unlike the [in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) operator, this method does not check for the specified property in the object's prototype chain.

It is recommended over [Object.prototype.hasOwnProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty) because it works for objects created using Object.create(null) and with objects that have overridden the inherited hasOwnProperty() method. While it is possible to workaround these problems by calling Object.prototype.hasOwnProperty() on an external object, Object.hasOwn() is more intuitive.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#examples)

[**Using hasOwn to test for a property's existence**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#using_hasown_to_test_for_a_propertys_existence)

The following code shows how to determine whether the example object contains a property named prop.

const example = {};

Object.hasOwn(example, 'prop'); // false - 'prop' has not been defined

example.prop = 'exists';

Object.hasOwn(example, 'prop'); // true - 'prop' has been defined

example.prop = null;

Object.hasOwn(example, 'prop'); // true - own property exists with value of null

example.prop = undefined;

Object.hasOwn(example, 'prop'); // true - own property exists with value of undefined

Copy to Clipboard

[**Direct vs. inherited properties**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#direct_vs._inherited_properties)

The following example differentiates between direct properties and properties inherited through the prototype chain:

const example = {};

example.prop = 'exists';

// `hasOwn` will only return true for direct properties:

Object.hasOwn(example, 'prop'); // returns true

Object.hasOwn(example, 'toString'); // returns false

Object.hasOwn(example, 'hasOwnProperty'); // returns false

// The `in` operator will return true for direct or inherited properties:

'prop' in example; // returns true

'toString' in example; // returns true

'hasOwnProperty' in example; // returns true

Copy to Clipboard

[**Iterating over the properties of an object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#iterating_over_the_properties_of_an_object)

To iterate over the enumerable properties of an object, you *should* use:

const example = { foo: true, bar: true };

for (const name of Object.keys(example)) {

// …

}

Copy to Clipboard

But if you need to use for...in, you can use Object.hasOwn() to skip the inherited properties:

const example = { foo: true, bar: true };

for (const name in example) {

if (Object.hasOwn(example, name)) {

// …

}

}

Copy to Clipboard

[**Checking if an Array index exists**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#checking_if_an_array_index_exists)

The elements of an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) are defined as direct properties, so you can use hasOwn() method to check whether a particular index exists:

const fruits = ['Apple', 'Banana','Watermelon', 'Orange'];

Object.hasOwn(fruits, 3); // true ('Orange')

Object.hasOwn(fruits, 4); // false - not defined

Copy to Clipboard

[**Problematic cases for hasOwnProperty**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn#problematic_cases_for_hasownproperty)

This section demonstrates that hasOwn() is immune to the problems that affect hasOwnProperty. Firstly, it can be used with objects that have reimplemented hasOwnProperty():

const foo = {

hasOwnProperty() {

return false;

},

bar: 'The dragons be out of office',

};

if (Object.hasOwn(foo, 'bar')) {

console.log(foo.bar); //true - reimplementation of hasOwnProperty() does not affect Object

}

Copy to Clipboard

It can also be used to test objects created using [Object.create(null)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create). These do not inherit from Object.prototype, and so hasOwnProperty() is inaccessible.

const foo = Object.create(null);

foo.prop = 'exists';

if (Object.hasOwn(foo, 'prop')) {

console.log(foo.prop); //true - works irrespective of how the object is created.

}

**Object.prototype.hasOwnProperty()**

The **hasOwnProperty()** method returns a boolean indicating whether the object has the specified property as its own property (as opposed to inheriting it).

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#try_it)

**Note:** [Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) is recommended over hasOwnProperty(), in browsers where it is supported.

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#syntax)

hasOwnProperty(prop)

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#parameters)

prop

The [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) name or [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol) of the property to test.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#return_value)

Returns true if the object has the specified property as own property; false otherwise.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#description)

The **hasOwnProperty()** method returns true if the specified property is a direct property of the object — even if the value is null or undefined. The method returns false if the property is inherited, or has not been declared at all. Unlike the [in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) operator, this method does not check for the specified property in the object's prototype chain.

The method can be called on *most* JavaScript objects, because most objects descend from [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object), and hence inherit its methods. For example [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) is an [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object), so you can use hasOwnProperty() method to check whether an index exists:

const fruits = ['Apple', 'Banana','Watermelon', 'Orange'];

fruits.hasOwnProperty(3); // true ('Orange')

fruits.hasOwnProperty(4); // false - not defined

Copy to Clipboard

The method will not be available in objects where it is reimplemented, or on objects created using Object.create(null) (as these don't inherit from Object.prototype). Examples for these cases are given below.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#examples)

[**Using hasOwnProperty to test for an own property's existence**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#using_hasownproperty_to_test_for_an_own_propertys_existence)

The following code shows how to determine whether the example object contains a property named prop.

const example = {};

example.hasOwnProperty('prop'); // false

example.prop = 'exists';

example.hasOwnProperty('prop'); // true - 'prop' has been defined

example.prop = null;

example.hasOwnProperty('prop'); // true - own property exists with value of null

example.prop = undefined;

example.hasOwnProperty('prop'); // true - own property exists with value of undefined

Copy to Clipboard

[**Direct vs. inherited properties**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#direct_vs._inherited_properties)

The following example differentiates between direct properties and properties inherited through the prototype chain:

const example = {};

example.prop = 'exists';

// `hasOwnProperty` will only return true for direct properties:

example.hasOwnProperty('prop'); // returns true

example.hasOwnProperty('toString'); // returns false

example.hasOwnProperty('hasOwnProperty'); // returns false

// The `in` operator will return true for direct or inherited properties:

'prop' in example; // returns true

'toString' in example; // returns true

'hasOwnProperty' in example; // returns true

Copy to Clipboard

[**Iterating over the properties of an object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#iterating_over_the_properties_of_an_object)

The following example shows how to iterate over the enumerable properties of an object without executing on inherited properties.

const buz = {

fog: 'stack',

};

for (const name in buz) {

if (buz.hasOwnProperty(name)) {

console.log(`this is fog (${name}) for sure. Value: ${buz[name]}`);

} else {

console.log(name); // toString or something else

}

}

Copy to Clipboard

Note that the [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop only iterates enumerable items: the absence of non-enumerable properties emitted from the loop does not imply that hasOwnProperty itself is confined strictly to enumerable items (as with [Object.getOwnPropertyNames()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames)).

[**Using hasOwnProperty as a property name**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#using_hasownproperty_as_a_property_name)

JavaScript does not protect the property name hasOwnProperty; an object that has a property with this name may return incorrect results:

const foo = {

hasOwnProperty() {

return false;

},

bar: 'Here be dragons',

};

foo.hasOwnProperty('bar'); // reimplementation always returns false

Copy to Clipboard

The recommended way to overcome this problem is to instead use [Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) (in browsers that support it). Other alternatives include using an *external* hasOwnProperty:

const foo = { bar: 'Here be dragons' };

// Use Object.hasOwn() method - recommended

Object.hasOwn(foo, "bar"); // true

// Use the hasOwnProperty property from the Object prototype

Object.prototype.hasOwnProperty.call(foo, 'bar'); // true

// Use another Object's hasOwnProperty

// and call it with 'this' set to foo

({}).hasOwnProperty.call(foo, 'bar'); // true

Copy to Clipboard

Note that in the first two cases there are no newly created objects.

[**Objects created with Object.create(null)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty#objects_created_with_object.createnull)

Objects created using [Object.create(null)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) do not inherit from Object.prototype, making hasOwnProperty() inaccessible.

const foo = Object.create(null);

foo.prop = 'exists';

foo.hasOwnProperty("prop"); // Uncaught TypeError: foo.hasOwnProperty is not a function

Copy to Clipboard

The solutions in this case are the same as for the previous section: use [Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) by preference, otherwise use an external object's hasOwnProperty().

The **Object.assign()** method copies all [enumerable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/propertyIsEnumerable) [own properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) from one or more *source objects* to a *target object*. It returns the modified target object.

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#syntax)

Object.assign(target, ...sources)

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#parameters)

target

The target object — what to apply the sources' properties to, which is returned after it is modified.

sources

The source object(s) — objects containing the properties you want to apply.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#return_value)

The target object.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#description)

Properties in the target object are overwritten by properties in the sources if they have the same [key](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys). Later sources' properties overwrite earlier ones.

The Object.assign() method only copies *enumerable* and *own* properties from a source object to a target object. It uses [[Get]] on the source and [[Set]] on the target, so it will invoke [getters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get) and [setters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set). Therefore it *assigns* properties, versus copying or defining new properties. This may make it unsuitable for merging new properties into a prototype if the merge sources contain getters.

For copying property definitions (including their enumerability) into prototypes, use [Object.getOwnPropertyDescriptor()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor) and [Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) instead.

Both [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) and [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol) properties are copied.

In case of an error, for example if a property is non-writable, a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError) is raised, and the target object is changed if any properties are added before the error is raised.

**Note:** Object.assign() does not throw on [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null) or [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) sources.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#examples)

[**Cloning an object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#cloning_an_object)

const obj = { a: 1 };

const copy = Object.assign({}, obj);

console.log(copy); // { a: 1 }

Copy to Clipboard

[**Warning for Deep Clone**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#warning_for_deep_clone)

For [deep cloning](https://developer.mozilla.org/en-US/docs/Glossary/Deep_copy), we need to use alternatives, because Object.assign() copies property values.

If the source value is a reference to an object, it only copies the reference value.

const obj1 = { a: 0, b: { c: 0 } };

const obj2 = Object.assign({}, obj1);

console.log(obj2); // { a: 0, b: { c: 0 } }

obj1.a = 1;

console.log(obj1); // { a: 1, b: { c: 0 } }

console.log(obj2); // { a: 0, b: { c: 0 } }

obj2.a = 2;

console.log(obj1); // { a: 1, b: { c: 0 } }

console.log(obj2); // { a: 2, b: { c: 0 } }

obj2.b.c = 3;

console.log(obj1); // { a: 1, b: { c: 3 } }

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

// Deep Clone

const obj3 = { a: 0, b: { c: 0 } };

const obj4 = JSON.parse(JSON.stringify(obj3));

obj3.a = 4;

obj3.b.c = 4;

console.log(obj4); // { a: 0, b: { c: 0 } }

Copy to Clipboard

[**Merging objects**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#merging_objects)

const o1 = { a: 1 };

const o2 = { b: 2 };

const o3 = { c: 3 };

const obj = Object.assign(o1, o2, o3);

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

console.log(o1); // { a: 1, b: 2, c: 3 }, target object itself is changed.

Copy to Clipboard

[**Merging objects with same properties**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#merging_objects_with_same_properties)

const o1 = { a: 1, b: 1, c: 1 };

const o2 = { b: 2, c: 2 };

const o3 = { c: 3 };

const obj = Object.assign({}, o1, o2, o3);

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

Copy to Clipboard

The properties are overwritten by other objects that have the same properties later in the parameters order.

[**Copying symbol-typed properties**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#copying_symbol-typed_properties)

const o1 = { a: 1 };

const o2 = { [Symbol('foo')]: 2 };

const obj = Object.assign({}, o1, o2);

console.log(obj); // { a : 1, [Symbol("foo")]: 2 } (cf. bug 1207182 on Firefox)

Object.getOwnPropertySymbols(obj); // [Symbol(foo)]

Copy to Clipboard

[**Properties on the prototype chain and non-enumerable properties cannot be copied**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#properties_on_the_prototype_chain_and_non-enumerable_properties_cannot_be_copied)

const obj = Object.create({ foo: 1 }, { // foo is on obj's prototype chain.

bar: {

value: 2 // bar is a non-enumerable property.

},

baz: {

value: 3,

enumerable: true // baz is an own enumerable property.

}

});

const copy = Object.assign({}, obj);

console.log(copy); // { baz: 3 }

Copy to Clipboard

[**Primitives will be wrapped to objects**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#primitives_will_be_wrapped_to_objects)

const v1 = 'abc';

const v2 = true;

const v3 = 10;

const v4 = Symbol('foo');

const obj = Object.assign({}, v1, null, v2, undefined, v3, v4);

// Primitives will be wrapped, null and undefined will be ignored.

// Note, only string wrappers can have own enumerable properties.

console.log(obj); // { "0": "a", "1": "b", "2": "c" }

Copy to Clipboard

[**Exceptions will interrupt the ongoing copying task**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#exceptions_will_interrupt_the_ongoing_copying_task)

const target = Object.defineProperty({}, 'foo', {

value: 1,

writable: false

}); // target.foo is a read-only property

Object.assign(target, { bar: 2 }, { foo2: 3, foo: 3, foo3: 3 }, { baz: 4 });

// TypeError: "foo" is read-only

// The Exception is thrown when assigning target.foo

console.log(target.bar); // 2, the first source was copied successfully.

console.log(target.foo2); // 3, the first property of the second source was copied successfully.

console.log(target.foo); // 1, exception is thrown here.

console.log(target.foo3); // undefined, assign method has finished, foo3 will not be copied.

console.log(target.baz); // undefined, the third source will not be copied either.

Copy to Clipboard

[**Copying accessors**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#copying_accessors)

const obj = {

foo: 1,

get bar() {

return 2;

}

};

let copy = Object.assign({}, obj);

console.log(copy);

// { foo: 1, bar: 2 }

// The value of copy.bar is obj.bar's getter's return value.

// This is an assign function that copies full descriptors

function completeAssign(target, ...sources) {

sources.forEach((source) => {

const descriptors = Object.keys(source).reduce((descriptors, key) => {

descriptors[key] = Object.getOwnPropertyDescriptor(source, key);

return descriptors;

}, {});

// By default, Object.assign copies enumerable Symbols, too

Object.getOwnPropertySymbols(source).forEach((sym) => {

const descriptor = Object.getOwnPropertyDescriptor(source, sym);

if (descriptor.enumerable) {

descriptors[sym] = descriptor;

}

});

Object.defineProperties(target, descriptors);

});

return target;

}

copy = completeAssign({}, obj);

console.log(copy);

// { foo:1, get bar() { return 2 } }

Copy to Clipboard

Thread

To master Javascript you have to understand objects as everything in JS is object.

Below topics will help you understand basics of Objects.

1 Introduction to objects

2 Creating objects

3 Accessing data in objects

4 Modification of objects

5 Nesting array and objects

That's all for this thread, this thread included the basic concepts of functions in javascript . I will be coming up with more advance javascript , till then do follow me for more content and if you have any queries drop them below