Javascript Object Methods

**Summary**: in this tutorial, you’ll learn about the JavaScript object methods and how to define methods for an object.

Introduction to the JavaScript object methods

An object is a collection of key/value pairs or [properties](https://www.javascripttutorial.net/javascript-object-properties/). When the value is a function, the property becomes a method. Typically, you use methods to describe the object behaviors.

For example, the following adds the greet method to the person object:

let person = {

firstName: 'John',

lastName: 'Doe'

};

person.greet = function () {

console.log('Hello!');

}

person.greet();

Code language: JavaScript (javascript)

Output:

Hello!

In this example:

* First, use a function expression to define a function and assign it to the greet property of the person object.
* Then, call the method greet() method.

Besides using a function expression, you can define a function and assign it to an object like this:

let person = {

firstName: 'John',

lastName: 'Doe'

};

function greet() {

console.log('Hello, World!');

}

person.greet = greet;

person.greet();

Code language: JavaScript (javascript)

In this example:

* First, define the greet() function as a regular function.
* Second, assign the function name to the the greet property of the person object.
* Third, call the greet() method.

Object method shorthand

JavaScript allows you to define methods of an object using the object literal syntax as shown in the following example:

let person = {

firstName: 'John',

lastName: 'Doe',

greet: function () {

console.log('Hello, World!');

}

};

Code language: JavaScript (javascript)

ES6 provides you with the [concise method syntax](https://www.javascripttutorial.net/es6/object-literal-extensions/) that allows you to define a method for an object:

let person = {

firstName: 'John',

lastName: 'Doe',

greet() {

console.log('Hello, World!');

}

};

person.greet();

Code language: JavaScript (javascript)

This syntax looks much cleaner and less verbose.

The this value

Typically, methods need to access other properties of the object.

For example, you may want to define a method that returns the full name of the person object by concatenating the first name and last name.

Inside a method, the this value references the object that invokes the method. Therefore, you can access a property using the this value as follows:

this.propertyName

Code language: JavaScript (javascript)

The following example uses the this value in the getFullName() method:

let person = {

firstName: 'John',

lastName: 'Doe',

greet: function () {

console.log('Hello, World!');

},

getFullName: function () {

return this.firstName + ' ' + this.lastName;

}

};

console.log(person.getFullName());

Code language: JavaScript (javascript)

Output

'John Doe'

Code language: JavaScript (javascript)

Check out [this tutorial](https://www.javascripttutorial.net/javascript-this/) for more information on [this](https://www.javascripttutorial.net/javascript-this/) value.

Summary

* When a function is a property of an object, it becomes a method.

JavaScript Constructor Function

**Summary**: in this tutorial, you’ll learn about the JavaScript constructor function and how to use the new keyword to create an object.

Introduction to JavaScript constructor functions

In the [JavaScript objects tutorial](https://www.javascripttutorial.net/javascript-objects/), you learned how to use the object literal syntax to create a new object.

For example, the following creates a new person object with two properties firstName and lastName:

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

In practice, you often need to create many similar objects like the person object.

To do that, you can use a constructor function to define a custom type and the new operator to create multiple objects from this type.

Technically speaking, a constructor function is a regular [function](https://www.javascripttutorial.net/javascript-function/) with the following convention:

* The name of a constructor function starts with a capital letter like Person, Document, etc.
* A constructor function should be called only with the new operator.

Note that ES6 introduces the [class](https://www.javascripttutorial.net/es6/javascript-class/) keyword that allows you to define a custom type. And classes are just syntactic sugar over the constructor functions with some enhancements.

The following example defines a constructor function called Person:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

Code language: JavaScript (javascript)

In this example, the Person is the same as a regular function except that its name starts with the capital letter P.

To create a new instance of the Person, you use the new operator:

let person = new Person('John','Doe');

Code language: JavaScript (javascript)

Basically, the new operator does the following:

* Create a new empty object and assign it to the this variable.
* Assign the arguments 'John' and 'Doe' to the firstName and lastName properties of the object.
* Return the this value.

It’s functionally equivalent to the following:

function Person(firstName, lastName) {

*// this = {};*

*// add properties to this*

this.firstName = firstName;

this.lastName = lastName;

*// return this;*

}

Code language: JavaScript (javascript)

Therefore, the following statement:

let person = new Person('John','Doe');

Code language: JavaScript (javascript)

… returns the same result as the following statement:

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

However, the constructor function Person allows you to create multiple similar objects. For example:

let person1 = new Person('Jane','Doe')

let person2 = new Person('James','Smith')

Code language: JavaScript (javascript)

Adding methods to JavaScript constructor functions

An object may have methods that manipulate its data. To add a method to an object created via the constructor function, you can use the this keyword. For example:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

this.getFullName = function () {

return this.firstName + " " + this.lastName;

};

}

Code language: JavaScript (javascript)

Now, you can create a new Person object and invoke the getFullName() method:

let person = new Person("John", "Doe");

console.log(person.getFullName());

Code language: JavaScript (javascript)

Output:

John Doe

The problem with the constructor function is that when you create multiple instances of the Person, the this.getFullName() is duplicated in every instance, which is not memory efficient.

To resolve this, you can use the [prototype](https://www.javascripttutorial.net/javascript-prototype/) so that all instances of a custom type can share the same methods.

Returning from constructor functions

Typically, a constructor function implicitly returns this that set to the newly created object. But if it has a return statement, then here’s are the rules:

* If return is called with an object, the constructor function returns that object instead of this.
* If return is called with a value other than an object, it is ignored.

Calling a constructor function without the new keyword

Technically, you can call a constructor function like a regular function without using the new keyword like this:

let person = Person('John','Doe');

Code language: JavaScript (javascript)

In this case, the Person just executes like a regular function. Therefore, the this inside the Person function doesn’t bind to the person variable but the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/).

If you attempt to access the firstName or lastName property, you’ll get an error:

console.log(person.firstName);

Code language: CSS (css)

Error:

TypeError: Cannot read property 'firstName' of undefined

Code language: JavaScript (javascript)

Similarly, you cannot access the getFullName() method since it’s bound to the global object.

person.getFullName();

Code language: CSS (css)

Error:

TypeError: Cannot read property 'getFullName' of undefined

Code language: JavaScript (javascript)

To prevent a constructor function to be invoked without the new keyword, ES6 introduced the [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) property.

If a constructor function is called with the new keyword, the new.target returns a reference of the function. Otherwise, it returns undefined.

The following adds a statement inside the Person function to show the new.target to the console:

function Person(firstName, lastName) {

console.log(new.target);

this.firstName = firstName;

this.lastName = lastName;

this.getFullName = function () {

return this.firstName + " " + this.lastName;

};

}

Code language: JavaScript (javascript)

The following returns undefined because the Person constructor function is called like a regular function:

let person = Person("John", "Doe");

Code language: JavaScript (javascript)

Output:

undefined

Code language: JavaScript (javascript)

However, the following returns a reference to the Person function because it’s called with the new keyword:

let person = new Person("John", "Doe");

Code language: JavaScript (javascript)

Output:

[Function: Person]

Code language: JSON / JSON with Comments (json)

By using the new.target, you can force the callers of the constructor function to use the new keyword. Otherwise, you can throw an error like this:

function Person(firstName, lastName) {

if (!new.target) {

throw Error("Cannot be called without the new keyword");

}

this.firstName = firstName;

this.lastName = lastName;

}

Code language: JavaScript (javascript)

Alternatively, you can make the syntax more flexible by creating a new Person object if the users of the constructor function don’t use the new keyword:

function Person(firstName, lastName) {

if (!new.target) {

return new Person(firstName, lastName);

}

this.firstName = firstName;

this.lastName = lastName;

}

let person = Person("John", "Doe");

console.log(person.firstName);

Code language: JavaScript (javascript)

This pattern is often used in JavaScript libraries and frameworks to make the syntax more flexible.

Summary

* JavaScript constructor function is a regular function used to create multiple similar objects.

JavaScript Prototype

**Summary**: in this tutorial, you’ll learn about the JavaScript prototype and how it works under the hood.

Introduction to JavaScript prototype

In JavaScript, objects can inherit features from one another via **prototypes**. Every object has its own property called prototype.

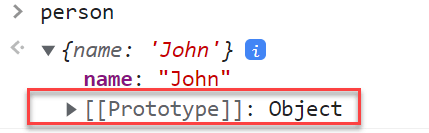
Because a prototype itself is also another object, the prototype has its own prototype. This creates a something called **prototype chain**. The prototype chain ends when a prototype has [null](https://www.javascripttutorial.net/object/javascript-null/) for its own prototype.

Suppose you have an object person with a property called name:

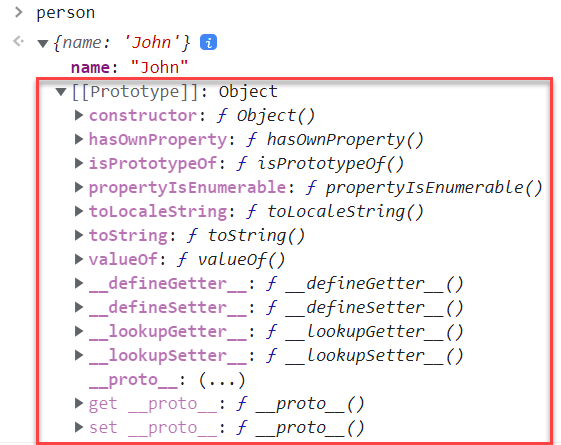
let person = {'name' : 'John'}

Code language: JavaScript (javascript)

When examining the person object in the console, you’ll find that the person object has a property called prototype denoted by the [[Prototype]]:

[](https://www.javascripttutorial.net/wp-content/uploads/2022/08/JavaScript-Prototype.png)

The prototype itself is an object with its own properties:

[](https://www.javascripttutorial.net/wp-content/uploads/2022/08/JavaScript-Prototype-object.png)

When you access a property of an object, if the object has that property, it’ll return the property value. The following example accesses the name property of the person object:

[](https://www.javascripttutorial.net/wp-content/uploads/2022/08/JavaScript-Prototype-access-property-that-exists.png)

It returns the value of the name property as expected.

However, if you access a property that doesn’t exist in an object, the JavaScript engine will search in the prototype of the object.

If the JavaScript engine cannot find the property in the object’s prototype, it’ll search in the prototype’s prototype until either it finds the property or reaches the end of the prototype chain.

For example, you can call the toString() method of the person object like this:

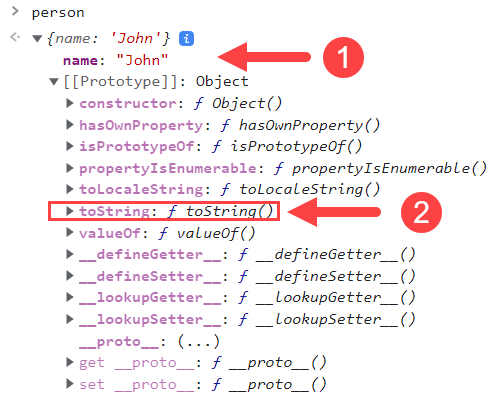
[](https://www.javascripttutorial.net/wp-content/uploads/2022/08/JavaScript-Prototype-toString-method.png)

The toString() method returns the string representation of the person object. By default, it’s [object Object] which is not obvious.

Note that when a [function](https://www.javascripttutorial.net/javascript-function/) is a value of an object’s property, it’s called a method. Therefore, a method is a property with value as a function.

In this example, when we call the toString() method on the person object, the JavaScript engine finds it in the person object. Because the person object doesn’t have the toString() method, it’ll search for the toString() method in the person’s prototype object.

Since the person’s prototype has the toString() method, JavaScript calls the toString() of the person’s prototype object.

[](https://www.javascripttutorial.net/wp-content/uploads/2022/08/JavaScript-Prototype-prototype-chain-search.png)

JavaScript prototype illustration

JavaScript has the built-in Object() function. The [typeof](https://www.javascripttutorial.net/javascript-typeof/) operator returns 'function' if you pass the Object function to it. For example:

typeof(Object)

Code language: JavaScript (javascript)

Output:

'function'

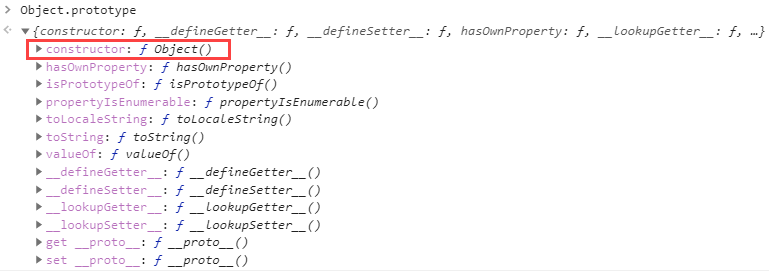
Code language: JavaScript (javascript)

Please note that Object() is a function, not an object. It’s confusing if this is the first time you’ve learned about the JavaScript prototype.

Also, JavaScript provides an anonymous [object](https://www.javascripttutorial.net/javascript-objects/) that can be referenced via the prototype property of the Object() function:

console.log(Object.prototype);

Code language: JavaScript (javascript)

[](https://www.javascripttutorial.net/wp-content/uploads/2021/01/JavaScript-Prototype-Object.prototype.png)

The Object.prototype object has some useful [properties](https://www.javascripttutorial.net/javascript-object-properties/) and [methods](https://www.javascripttutorial.net/javascript-object-methods/) such as toString() and valueOf().

The Object.prototype also has an important property called constructor that references the Object() function.

The following statement confirms that the Object.prototype.constructor property references the Object function:

console.log(Object.prototype.constructor === Object); *// true*

Code language: JavaScript (javascript)

Suppose a circle represents a function and a square represents an object. The following picture illustrates the relationship between the Object() function and the Object.prototype object:

First, define a [constructor function](https://www.javascripttutorial.net/javascript-constructor-function/) called Person as follows:

function Person(name) {

this.name = name;

}

Code language: JavaScript (javascript)

In this example, the Person() function accepts a name argument and assigns it to the name property of the this object.

Behind the scenes, JavaScript creates a new function Person() and an anonymous object:

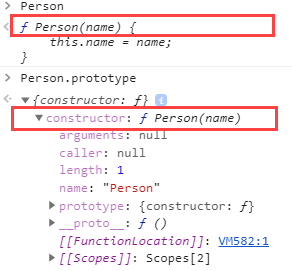
Like the Object() function, the Person() function has a property called prototype that references an anonymous object. And the anonymous object has the constructor property that references the Person() function.

The following shows the Person() function and the anonymous object referenced by the Person.prototype:

console.log(Person);

console.log(Person.prototype);

Code language: CSS (css)

[](https://www.javascripttutorial.net/wp-content/uploads/2021/01/JavaScript-Prototype-Person-function.png)

In addition, JavaScript links the Person.prototype object to the Object.prototype object via the [[Prototype]], which is known as a *prototype linkage*.

The prototype linkage is denoted by [[Prototype]] in the following figure:

Defining methods in the JavaScript prototype object

The following defines a new method called greet() in the Person.prototype object:

Person.prototype.greet = function() {

return "Hi, I'm " + this.name + "!";

}

Code language: JavaScript (javascript)

In this case, the JavaScript engine adds the greet() method to the Person.prototype object:

The following creates a new instance of the Person :

let p1 = new Person('John');

Code language: JavaScript (javascript)

Internally, the JavaScript engine creates a new object named p1 and links the p1 object to the Person.prototype object via the prototype linkage:

The link between p1, Person.prototype, and Object.protoype is called a *prototype chain*.

The following calls the greet() method on the p1 object:

let greeting = p1.greet();

console.log(greeting);

Code language: JavaScript (javascript)

Because p1 doesn’t have the greet() method, JavaScript follows the prototype linkage and finds it on the Person.prototype object.

Since JavaScript can find the greet() method on the Person.prototype object, it executes the greet() method and returns the result:

The following calls the toString() method on the p1 object:

let s = p1.toString();

console.log(s);

Code language: JavaScript (javascript)

In this case, the JavaScript engine follows the prototype chain to look up for the toString() method in the Person.prototype.

Because the Person.prototype doesn’t have the toString() method, the JavaScript engine goes up to the prototype chain and searches for the toString() method in the Object.prototype object.

Since JavaScript can find the toString() method in the Object.prototype, it executes the toString() method.

If you call a method that doesn’t exist on the Person.prototype and Object.prototype object, the JavaScript engine will follow the prototype chain and throw an error if it cannot find the method. For example:

p1.fly();

Code language: CSS (css)

Because the fly() method doesn’t exist on any object in the prototype chain, the JavaScript engine issues the following error:

TypeError: p1.fly is not a function

Code language: JavaScript (javascript)

The following creates another instance of the Person whose name property is 'Jane':

let p2 = new Person('Jane');

Code language: JavaScript (javascript)

The p2 object has the properties and methods as the p1 object.

In conclusion, when you define a method on the prototype object, this method is shared by all instances.

Defining methods in an individual object

The following defines the draw() method on the p2 object.

p2.draw = function () {

return "I can draw.";

};

Code language: JavaScript (javascript)

The JavaScript engine adds the draw() method to the p2 object, not the Person.prototype object:

It means that you can call the draw() method on the p2 object:

p2.draw();

Code language: CSS (css)

But you cannot call the draw() method on the p1 object:

p1.draw()

Code language: CSS (css)

Error:

TypeError: p1.draw is not a function

Code language: JavaScript (javascript)

When you define a method in an object, the method is only available to that object. It cannot be shared with other objects by default.

Getting prototype linkage

The \_\_proto\_\_ is pronounced as dunder proto. The \_\_proto\_\_ is an [accessor property](https://www.javascripttutorial.net/javascript-object-properties/) of the Object.prototype object. It exposes the internal prototype linkage ( [[Prototype]]) of an object through which it is accessed.

The \_\_proto\_\_ has been standardized in [ES6](https://www.javascripttutorial.net/es6/) to ensure compatibility for web browsers. However, it may be deprecated in favor of Object.getPrototypeOf() in the future. Therefore, you should never use the \_\_proto\_\_ in your production code.

The  p1.\_\_proto\_\_ exposes the [[Prototype]] that references the Person.prototype object.

Similarly, p2.\_\_proto\_\_ also references the same object as p1.\_\_proto\_\_:

console.log(p1.\_\_proto\_\_ === Person.prototype); *// true*

console.log(p1.\_\_proto\_\_ === p2.\_\_proto\_\_); *// true*

Code language: JavaScript (javascript)

As mentioned earlier, you should use the Object.getPrototypeOf() method instead of the \_\_proto\_\_. The Object.getPrototypeOf() method returns the prototype of a specified object.

console.log(p1.\_\_proto\_\_ === Object.getPrototypeOf(p1)); *// true*

Code language: JavaScript (javascript)

Another popular way to get the prototype linkage is when the Object.getPrototypeOf() method is not available is via the constructor property as follows:

p1.constructor.prototype

Code language: CSS (css)

The p1.constructor returns Person, therefore, p1.constructor.prototype returns the prototype object.

Shadowing

See the following method call:

console.log(p1.greet());

Code language: CSS (css)

The p1 object doesn’t have the greet() method defined, therefore JavaScript goes up to the prototype chain to find it. In this case, it can find the method in the Person.prototype object.

Let’s add a new method to the object p1 with the same name as the method in the Person.prototype object:

p1.greet = function() {

console.log('Hello');

}

Code language: JavaScript (javascript)

And call the greet() method:

console.log(p1.greet());

Code language: CSS (css)

Because the p1 object has the greet() method, JavaScript just executes it immediately without looking it up in the prototype chain.

This is an example of shadowing. The greet() method of the p1 object shadows the greet() method of the prototype object which the p1 object references.

Summary

* The Object() function has a property called prototype that references a Object.prototype object.
* The Object.prototype object has all properties and methods which are available in all objects such as toString() and valueOf().
* The Object.prototype object has the constructor property that references the Object function.
* Every function has a prototype object. This prototype object references the Object.prototype object via [[prototype]] linkage or \_\_proto\_\_ property.
* The prototype chain allows one object to use the methods and properties of its prototype objects via the [[prototype]] linkages.
* The Object.getPrototypeOf() method returns the prototype object of a given object. Do use the Object.getPrototypeOf() method instead of \_\_proto\_\_.

JavaScript Constructor/Prototype Pattern

**Summary**: in this tutorial, you’ll learn how to use the JavaScript constructor/Prototype pattern to define a custom type in ES5.

Introduction to the JavaScript Constructor / Prototype pattern

The combination of the [constructor](https://www.javascripttutorial.net/javascript-constructor-function/)and [prototype](https://www.javascripttutorial.net/javascript-prototype/) patterns is the most common way to define custom types in ES5. In this pattern:

* The constructor pattern defines the object properties.
* The prototype pattern defines the object methods.

By using this pattern, all objects of the custom type share the methods defined in the prototype. Also, each object has its own properties.

This constructor/prototype pattern takes the best parts of both constructor and prototype patterns.

JavaScript Constructor / Prototype example

Suppose that you want to define a custom type called Person that has:

* Two properties firstName and lastName
* One method getFullName()

First, use the [constructor function](https://www.javascripttutorial.net/javascript-constructor-function/) to initialize the properties:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

Code language: JavaScript (javascript)

Behind the scene, the JavaScript engine defines a Person function denoted by the circle and an anonymous object denoted by the square.

The Person function has the prototype property that references an anonymous object. The anonymous object has a constructor property that references the Person function:

Second, define the getFullName() method in the prototype object of the Person function:

Person.prototype.getFullName = function () {

return this.firstName + ' ' + this.lastName;

};

Code language: JavaScript (javascript)

JavaScript defines the getFullName() method on the Person.prototype object like this:

Third, create multiple instances of the Person type:

let p1 = new Person("John", "Doe");

let p2 = new Person("Jane", "Doe");

console.log(p1.getFullName());

console.log(p2.getFullName());

Code language: JavaScript (javascript)

Output:

'John Doe'

'Jane Doe'

Code language: JavaScript (javascript)

Javascript creates two objects p1 and p2. These objects link to the Person.prototype object via the [[Prototype]] linkage:

Each object has its own properties firstName and lastName. However, they share the same getFullName() method.

When you call the getFullName() method on the p1 or p2 object, the JavaScript engine searches for the method on these objects. Because the JavaScript engine doesn’t find the method there, it follows the prototype linkage and searches for the method in the Person.prototype object.

Because the Person.prototype object has the getFullName() method, JavaScript stops searching and executes the method.

Put it all together:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

Person.prototype.getFullName = function () {

return this.firstName + ' ' + this.lastName;

};

let p1 = new Person('John', 'Doe');

let p2 = new Person('Jane', 'Doe');

console.log(p1.getFullName());

console.log(p2.getFullName());

Code language: JavaScript (javascript)

Classes in ES6

ES6 introduces the [class](https://www.javascripttutorial.net/es6/javascript-class/) keyword that makes the constructor/prototype pattern easier to use. For example, the following uses the class keyword to define exactly the same Person type:

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

getFullName() {

return this.firstName + " " + this.lastName;

}

}

let p1 = new Person('John', 'Doe');

let p2 = new Person('Jane', 'Doe');

console.log(p1.getFullName());

console.log(p2.getFullName());

Code language: JavaScript (javascript)

In this syntax, the class moves the property initialization to the constructor method. It also packs the getFullName() method in the same place as the constructor function.

The class syntax looks cleaner and less verbose. However, it’s syntactic sugar over the constructor/prototype pattern with some enhancements.

For more information on the classes, check out the [JavaScript class tutorial](https://www.javascripttutorial.net/es6/javascript-class/).

Summary

* Use JavaScript constructor/prototype to define a custom type in ES5.
* Initialize the object properties in the constructor function and define methods and properties that can be shared by all instances in the prototype object.

JavaScript Prototypal Inheritance

**Summary**: in this tutorial, you’ll learn how the JavaScript prototypal inheritance works.

Introduction to JavaScript prototypal inheritance

If you’ve worked with other object-oriented programming languages such as Java or C++, you’ve been familiar with the inheritance concept.

In this programming paradigm, a class is a blueprint for creating objects. If you want a new class to reuse the functionality of an existing class, you can create a new class that extends the existing class. This is called **classical inheritance**.

JavaScript doesn’t use **classical inheritance**. Instead, it uses **prototypal inheritance**.

In prototypal inheritance, an object “inherits” [properties](https://www.javascripttutorial.net/javascript-object-properties/) from another object via the [prototype](https://www.javascripttutorial.net/javascript-prototype/) linkage.

JavaScript prototypal inheritance and \_\_proto\_\_

Let’s take an example to make the concept clear.

The following defines a person object:

let person = {

name: "John Doe",

greet: function () {

return "Hi, I'm " + this.name;

}

};

Code language: JavaScript (javascript)

In this example, the person object has a property and a method:

* name is a property that stores the person’s name.
* greet is a method that returns a greeting as a string.

By default, the JavaScript engine provides you with a built-in Object() function and an anonymous object that can be referenced by the Object.prototype:

Note that the circle represents a function while the square represents an object.

The person object has a link to the anonymous object referenced by the Object() function. The [[Prototype]] represents the linkage:

It means that the person object can call any methods defined in the anonymous object referenced by the Object.prototype likes this. For example, the following shows how to call the toString() method via the person object:

console.log(person.toString());

Code language: JavaScript (javascript)

Output:

[object Object]

Code language: JavaScript (javascript)

The [object Object] is the default string representation of an object.

When you call toString() method via person, the JavaScript engine cannot find it on the person object. Therefore, the JavaScript engine follows the prototype chain and searches for the method in the Object.prototype object.

Since the JavaScript engine can find the toString() method in the Object.prototype object, it executes the toString() method.

To access the prototype of the person object, you can use the \_\_proto\_\_ property as follows

console.log(person.\_\_proto\_\_);

Code language: JavaScript (javascript)

The following shows the person.\_\_proto\_\_ and Object.prototype references the same object:

console.log(person.\_\_proto\_\_ === Object.prototype); *// true*

Code language: JavaScript (javascript)

The following defines the teacher object that has the teach() method:

let teacher = {

teach: function (subject) {

return "I can teach " + subject;

}

};

Code language: JavaScript (javascript)

Like the person object, the teacher.\_\_proto\_\_ references the Object.prototype as illustrated in the following picture:

If you want the teacher object to access all methods and properties of the person object, you can set the prototype of teacher object to the person object like this:

teacher.\_\_proto\_\_ = person;

Code language: JavaScript (javascript)

Note that you should never use the \_\_proto\_\_ property in the production code. Please use it for demonstration purposes only.

Now, the teacher object can access the name property and greet() method from the person object via the prototype chain:

console.log(teacher.name);

console.log(teacher.greet());

Code language: JavaScript (javascript)

Output:

John Doe

Hi, I'm John Doe

Code language: JavaScript (javascript)

When you call the greet() method on the teacher object, the JavaScript engine finds it in the teacher object first.

Since the JavaScript engine cannot find the method in the teacher object, it follows the prototype chain and searches for the method in the person object. Because the JavaScript can engine can find the greet() method in the person object, it executes the method.

In JavaScript, we say that the teacher object inherits the methods and properties of the person object. And this kind of inheritance is called prototypal inheritance.

A standard way to implement prototypal inheritance in ES5

ES5 provided a standard way to work with prototypal inheritance by using the Object.create() method.

Note that now you should use the newer ES6 [class](https://www.javascripttutorial.net/es6/javascript-class/) and [extends](https://www.javascripttutorial.net/es6/javascript-inheritance/) keywords to implement [inheritance](https://www.javascripttutorial.net/es6/javascript-inheritance/). It’s much simpler.

The Object.create() method creates a new object and uses an existing object as a prototype of the new object:

Object.create(proto, [propertiesObject])

Code language: JavaScript (javascript)

The Object.create() method accepts two arguments:

* The first argument (proto) is an object used as the prototype for the new object.
* The second argument (propertiesObject), if provided, is an optional object that defines additional properties for the new object.

Suppose you have a person object:

let person = {

name: "John Doe",

greet: function () {

return "Hi, I'm " + this.name;

}

};

Code language: JavaScript (javascript)

The following creates an empty teacher object with the \_\_proto\_\_ of the person object:

let teacher = Object.create(person);

Code language: JavaScript (javascript)

After that, you can define properties for the teacher object:

teacher.name = 'Jane Doe';

teacher.teach = function (subject) {

return "I can teach " + subject;

}

Code language: JavaScript (javascript)

Or you can do all of these steps in one statement as follows:

let teacher = Object.create(person, {

name: { value: 'John Doe' } ,

teach: { value: function(subject) {

return "I can teach " + subject;

}}

});

Code language: JavaScript (javascript)

ES5 also introduced the Object.getPrototypeOf() method that returns the prototype of an object. For example:

console.log(Object.getPrototypeOf(teacher) === person);

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

Summary

* Inheritance allows an object to use the properties and methods of another object without duplicating the code.
* JavaScript uses the prototypal inheritance.

Demystifying the JavaScript this Keyword

**Summary**: in this tutorial, you will  learn about the JavaScript this value and understand it clearly in various contexts.

If you have been working with other programming languages such as Java, [C#](https://www.csharptutorial.net/csharp-tutorial/csharp-this/), or [PHP](https://www.phptutorial.net/php-oop/php-this/), you’re already familiar with the this keyword.

In these languages, the this keyword represents the current instance of the class. And it is only relevant within the class.

JavaScript also has this keyword. However, the this keyword in JavaScript behaves differently from other programming languages.

In JavaScript, you can use the this keyword in the [global and function contexts](https://www.javascripttutorial.net/javascript-execution-context/). Moreover, the behavior of the  this keyword changes between strict and non-strict modes.

What is this keyword

In general, the this references the object of which the function is a property. In other words, the this references the object that is currently calling the function.

Suppose you have an object called counter that has a method next(). When you call the next() method, you can access the this object.

let counter = {

count: 0,

next: function () {

return ++this.count;

},

};

counter.next();

Code language: JavaScript (javascript)

Inside the next() function, the this references the counter object. See the following method call:

counter.next();

Code language: CSS (css)

The next() is a function that is the property of the counter object. Therefore, inside the next() function, the this references the counter object.

Global context

In the global context, the this references the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/), which is the window object on the web browser or global object on Node.js.

This behavior is consistent in both strict and non-strict modes. Here’s the output on the web browser:

console.log(this === window); *// true*

Code language: JavaScript (javascript)

If you assign a property to this object in the global context, JavaScript will add the property to the global object as shown in the following example:

this.color= 'Red';

console.log(window.color); *// 'Red'*

Code language: JavaScript (javascript)

Function context

In JavaScript, you can call a [function](https://www.javascripttutorial.net/javascript-function/) in the following ways:

* Function invocation
* Method invocation
* Constructor invocation
* Indirect invocation

Each function invocation defines its own context. Therefore, the this behaves differently.

1) Simple function invocation

In the non-strict mode, the this references the global object when the function is called as follows:

function show() {

console.log(this === window); *// true*

}

show();

Code language: JavaScript (javascript)

When you call the show() function, the this references the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/), which is the window on the web browser and global on Node.js.

Calling the show() function is the same as:

window.show();

Code language: JavaScript (javascript)

In the strict mode, JavaScript sets the this inside a function to undefined. For example:

"use strict";

function show() {

console.log(this === undefined);

}

show();

Code language: JavaScript (javascript)

To enable the strict mode, you use the directive "use strict" at the beginning of the JavaScript file. If you want to apply the strict mode to a specific function only, you place it at the top of the function body.

Note that the strict mode has been available since ECMAScript 5.1. The strict mode applies to both function and nested functions. For example:

function show() {

"use strict";

console.log(this === undefined); *// true*

function display() {

console.log(this === undefined); *// true*

}

display();

}

show();

Code language: JavaScript (javascript)

Output:

true

true

Code language: JavaScript (javascript)

In the display() inner function, the this also set to undefined as shown in the console.

2) Method invocation

When you call a method of an object, JavaScript sets this to the object that owns the method. See the following car object:

let car = {

brand: 'Honda',

getBrand: function () {

return this.brand;

}

}

console.log(car.getBrand()); *// Honda*

Code language: JavaScript (javascript)

In this example, the this object in the getBrand() method references the car object.

Since a method is a property of an object which is a value, you can store it in a variable.

let brand = car.getBrand;

Code language: JavaScript (javascript)

And then call the method via the variable

console.log(brand()); *// undefined*

Code language: JavaScript (javascript)

You get undefined instead of "Honda" because when you call a method without specifying its object, JavaScript sets this to the global object in non-strict mode and undefined in the strict mode.

To fix this issue, you use the [bind()](https://www.javascripttutorial.net/javascript-bind/) method of the Function.prototype object. The bind() method creates a new function whose the this keyword is set to a specified value.

let brand = car.getBrand.bind(car);

console.log(brand()); *// Honda*

Code language: JavaScript (javascript)

In this example, when you call the brand() method, the this keyword is bound to the car object. For example:

let car = {

brand: 'Honda',

getBrand: function () {

return this.brand;

}

}

let bike = {

brand: 'Harley Davidson'

}

let brand = car.getBrand.bind(bike);

console.log(brand());

Code language: JavaScript (javascript)

Output:

Harley Davidson

In this example, the bind() method sets the this to the bike object, therefore, you see the value of the brand property of the bike object on the console.

3) Constructor invocation

When you use the new keyword to create an instance of a function object, you use the function as a constructor.

The following example declares a Car function, then invokes it as a constructor:

function Car(brand) {

this.brand = brand;

}

Car.prototype.getBrand = function () {

return this.brand;

}

let car = new Car('Honda');

console.log(car.getBrand());

Code language: JavaScript (javascript)

The expression new Car('Honda') is a constructor invocation of the Car function.

JavaScript creates a new object and sets this to the newly created object. This pattern works great with only one potential problem.

Now, you can invoke the Car() as a function or as a constructor. If you omit the new keyword as follows:

var bmw = Car('BMW');

console.log(bmw.brand);

*// => TypeError: Cannot read property 'brand' of undefined*

Code language: JavaScript (javascript)

Since the this value in the Car() sets to the global object, the bmw.brand returns undefined.

To make sure that the Car() function is always invoked using constructor invocation, you add a check at the beginning of the Car() function as follows:

function Car(brand) {

if (!(this instanceof Car)) {

throw Error('Must use the new operator to call the function');

}

this.brand = brand;

}

Code language: JavaScript (javascript)

ES6 introduced a meta-property named [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) that allows you to detect whether a function is invoked as a simple invocation or as a constructor.

You can modify the Car() function that uses the new.target metaproperty as follows:

function Car(brand) {

if (!new.target) {

throw Error('Must use the new operator to call the function');

}

this.brand = brand;

}

Code language: JavaScript (javascript)

4) Indirect Invocation

In JavaScript, [functions are first-class citizens](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/). In other words, functions are objects, which are instances of the [Function type](https://www.javascripttutorial.net/javascript-function-type/).

The Function type has two methods: [call()](https://www.javascripttutorial.net/javascript-call/) and [apply()](https://www.javascripttutorial.net/javascript-apply-method/) . These methods allow you to set the this value when calling a function. For example:

function getBrand(prefix) {

console.log(prefix + this.brand);

}

let honda = {

brand: 'Honda'

};

let audi = {

brand: 'Audi'

};

getBrand.call(honda, "It's a ");

getBrand.call(audi, "It's an ");

Code language: JavaScript (javascript)

Output:

It's a Honda

It's an Audi

Code language: PHP (php)

In this example, we called the getBrand() function indirectly using the call() method of the getBrand function. We passed honda and  audi object as the first argument of the call() method, therefore, we got the corresponding brand in each call.

The apply() method is similar to the call() method except that its second argument is an array of arguments.

getBrand.apply(honda, ["It's a "]); *// "It's a Honda"*

getBrand.apply(audi, ["It's an "]); *// "It's a Audi"*

Code language: JavaScript (javascript)

Arrow functions

[ES6](https://www.javascripttutorial.net/es6/)introduced a new concept called [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/). In arrow functions, JavaScript sets the this lexically.

It means the arrow function does not create its own [execution context](https://www.javascripttutorial.net/javascript-execution-context/) but inherits the this from the outer function where the arrow function is defined. See the following example:

let getThis = () => this;

console.log(getThis() === window); *// true*

Code language: JavaScript (javascript)

In this example, the this value is set to the global object i.e., window in the web browser.

Since an arrow function does not create its own execution context, defining a method using an arrow function will cause an issue. For example:

function Car() {

this.speed = 120;

}

Car.prototype.getSpeed = () => {

return this.speed;

};

var car = new Car();

console.log(car.getSpeed()); *// 👉 undefined*

Code language: JavaScript (javascript)

Inside the getSpeed() method, the this value reference the global object, not the Car object but the global object doesn’t have a property called speed. Therefore, the this.speed in the getSpeed() method returns undefined.

JavaScript globalThis

**Summary**: in this tutorial, you’ll learn how to about the JavaScript globalThis object.

Introduction to the JavaScript globalThis object

ES2020 introduced the globalThis object that provides a standard way to access the global object across environments.

Historically, JavaScript had a global object with different names in different environments.

In web browsers, the global object is [window](https://www.javascripttutorial.net/javascript-bom/javascript-window/) or frames.

However, the Web Workers API doesn’t have the window object because it has no browsing context. Hence, the Web Workers API uses self as a global object.

Node.js, on the other hand, uses the global keyword to reference the global object.

| **Environment** | **Global** |
| --- | --- |
| Web Browsers | this |
| Web Workers | self |
| Node.js | global |

If you write JavaScript code that works across environments and needs to access the global object, you have to use different syntaxes like window, frames, self, or global.

To standardize this, ES2020 introduced the globalThis that is available across environments.

For example, the following code checks if the current environment supports the Fetch API:

const canFetch = typeof globalThis.fetch === 'function';

console.log(canFetch);

Code language: JavaScript (javascript)

The code checks if the fetch() function is a property of the global object. In the web browsers, the globalThis is the window object. Therefore, if you run this code on the modern web browser, the canFetch will be true.

The following code returns true on the web browser:

globalThis === window

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

Summary

* Use the globalThis object to reference the global object to make the code works across environments.

JavaScript Object Properties

**Summary**: in this tutorial, you will learn about the JavaScript object’s properties and attributes such as configurable, enumerable, writable, get, set, and value.

Object Property types

JavaScript specifies the characteristics of properties of [objects](https://www.javascripttutorial.net/javascript-objects/) via internal attributes surrounded by the two pairs of square brackets, e.g., [[Enumerable]].

Objects have two types of properties: data and accessor properties.

1) Data properties

A data property contains a single location for a data value. A data property has four attributes:

* [[Configurarable]] – determines whether a property can be redefined or removed via delete operator.
* [[Enumerable]] – indicates if a property can be returned in the [for...in](https://www.javascripttutorial.net/javascript-for-in/) loop.
* [[Writable]] – specifies that the value of a property can be changed.
* [[Value]] – contains the actual value of a property.

By default, the [[Configurable]] , [[Enumerable]]And [[Writable]] attributes set to true for all properties defined directly on an object. The default value of the[[Value]] attribute is undefined.

The following example creates a person object with two properties firstName and lastName with the configurable, enumerable, and writable attributes set to true. And their values are set to 'John' and 'Doe' respectively:

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

To change any attribute of a property, you use the Object.defineProperty() method.

The Object.defineProperty() method accepts three arguments:

* An object.
* A property name of the object.
* A property descriptor object that has four properties: configurable, enumerable, writable, and value.

If you use the Object.defineProperty() method to define a property of the object, the default values of [[Configurable]], [[Enumerable]], and [[Writable]] are set to false unless otherwise specified.

The following example creates a person object with the age property:

let person = {};

person.age = 25;

Code language: JavaScript (javascript)

Since the default value of the [[Configurable]] attribute is set to true, you can remove it via the delete operator:

delete person.age;

console.log(person.age);

Code language: CSS (css)

Output:

undefined

Code language: JavaScript (javascript)

The following example creates a person object and adds the ssn property to it using the Object.defineProperty() method:

'use strict';

let person = {};

Object.defineProperty(person, 'ssn', {

configurable: false,

value: '012-38-9119'

});

delete person.ssn;

Code language: JavaScript (javascript)

Output:

TypeError: Cannot delete property 'ssn' of *#<Object>*

Code language: PHP (php)

In this example, the configurable attribute is set to false. herefore, deleting the ssn property causes an error.

Also, once you define a property as non-configurable, you cannot change it to configurable.

If you use the Object.defineProperty() method to change any attribute other than the writable, you’ll get an error. or example:

'use strict';

let person = {};

Object.defineProperty(person, 'ssn', {

configurable: false,

value: '012-38-9119'

});

Object.defineProperty(person, 'ssn', {

configurable: true

});

Code language: JavaScript (javascript)

Output:

TypeError: Cannot redefine property: ssn

Code language: JavaScript (javascript)

By default, the enumerable attribute of all the properties defined on an object is true. t means that you can iterate over all object properties using the [for...in](https://www.javascripttutorial.net/javascript-for-in/)loop like this:

let person = {};

person.age = 25;

person.ssn = '012-38-9119';

for (let property in person) {

console.log(property);

}

Code language: JavaScript (javascript)

Output:

age

ssn

The following makes the ssn property non-enumerable by setting the enumerable attribute to false.

let person = {};

person.age = 25;

person.ssn = '012-38-9119';

Object.defineProperty(person, 'ssn', {

enumerable: false

});

for (let prop in person) {

console.log(prop);

}

Code language: JavaScript (javascript)

Output

age

2) Accessor properties

Similar to data properties, accessor properties also have [[Configurable]] and [[Enumerable]] attributes.

But the accessor properties have the [[Get]] and [[Set]] attributes instead of [[Value]] and [[Writable]].

When you read data from an accessor property, the [[Get]] function is called automatically to return a value. The default return value of the [[Get]] function is undefined.

If you assign a value to an accessor property, the [[Set]] function is called automatically.

To define an accessor property, you must use the Object.defineProperty() method. or example:

let person = {

firstName: 'John',

lastName: 'Doe'

}

Object.defineProperty(person, 'fullName', {

get: function () {

return this.firstName + ' ' + this.lastName;

},

set: function (value) {

let parts = value.split(' ');

if (parts.length == 2) {

this.firstName = parts[0];

this.lastName = parts[1];

} else {

throw 'Invalid name format';

}

}

});

console.log(person.fullName);

Code language: JavaScript (javascript)

Output:

'John Doe'

Code language: JavaScript (javascript)

In this example:

* First, define the person object that contains two properties: firstName and lastName.
* Then, add the fullName property to the person object as an accessor property.

In the fullname accessor property:

* The [[Get]] returns the full name that is the result of [concatenating](https://www.javascripttutorial.net/javascript-string-concat/) of firstName, space, and lastName.
* The [[Set]] method [splits](https://www.javascripttutorial.net/javascript-string-split/) the argument by the space and assigns the firstName and lastName properties the corresponding parts of the name.
* If the full name is not in the correct format i.e., first name, space, and last name, it will [throw an error](https://www.javascripttutorial.net/es6/promise-error-handling/).

Define multiple properties: Object.defineProperties()

In ES5, you can define multiple properties in a single statement using the Object.defineProperties() method. or example:

var product = {};

Object.defineProperties(product, {

name: {

value: 'Smartphone'

},

price: {

value: 799

},

tax: {

value: 0.1

},

netPrice: {

get: function () {

return this.price \* (1 + this.tax);

}

}

});

console.log('The net price of a ' + product.name + ' is ' + product.netPrice.toFixed(2) + ' USD');

Code language: JavaScript (javascript)

Output:

The net price of a Smartphone is 878.90 USD

Code language: CSS (css)

In this example, we defined three data properties: name, price, and tax, and one accessor property netPrice for the product object.

JavaScript object property descriptor

The  Object.getOwnPropertyDescriptor() method allows you to get the descriptor object of a property. The Object.getOwnPropertyDescriptor() method takes two arguments:

1. An object
2. A property of the object

It returns a descriptor object that describes a property. The descriptor object has four properties: configurable, enumerable, writable, and value.

The following example gets the descriptor object of the name property of the product object in the prior example.

let person = {

firstName: 'John',

lastName: 'Doe'

};

let descriptor = Object.getOwnPropertyDescriptor(person, 'firstName');

console.log(descriptor);

Code language: JavaScript (javascript)

Output:

{ value: 'John',

writable: true,

enumerable: true,

configurable: true }

Code language: CSS (css)

Summary

* JavaScript objects have two types of properties: data properties and accessor properties.
* JavaScript uses internal attributes denoted by [[...]] to describe the characteristics of properties such as [[Configurable]], [[Enumerable]], [[Writable]], and  [[Value]], [[Get]], and [[Set]].
* The method Object.getOwnPropertyDescriptor() return a property descriptor of a property in an object.
* A property can be defined directly on an object or indirectly via the Object.defineProperty() or Object.defineProperties() methods. These methods can be used to change the attributes of a property.

JavaScript for…in Loop

**Summary**: in this tutorial, you will learn how to use the JavaScript for...in loop to iterate over the enumerable properties of an object.

Introduction to JavaScript for...in loop

The for...in loop over the [enumerable properties](https://www.javascripttutorial.net/javascript-enumerable-properties/) that are keyed by strings of an [object](https://www.javascripttutorial.net/javascript-objects/). Note that a property can be keyed by a string or a [symbol](https://www.javascripttutorial.net/es6/symbol/).

A property is enumerable when its internal enumerable flag is set to true.

The enumerable flag defaults to true when a property is created via a simple assignment or via a property initializer:

object.propertyName = value;

or

let obj = {

propertyName: value,

...

};

Code language: JavaScript (javascript)

The following shows the syntax of the for...in loop:

for(const propertyName in object) {

*// ...*

}

Code language: JavaScript (javascript)

The for...in  allows you to access each property and value of an object without knowing the specific name of the property. For example:

var person = {

firstName: 'John',

lastName: 'Doe',

ssn: '299-24-2351'

};

for(var prop in person) {

console.log(prop + ':' + person[prop]);

}

Code language: JavaScript (javascript)

Output:

firstName:John

lastName:Doe

ssn:299-24-2351

Code language: CSS (css)

In this example, we used the for...in loop to iterate over the properties of the person object. We accessed the value of each property using the following syntax:

object[property];

Code language: CSS (css)

The for...in loop & Inheritance

When you loop over the properties of an object that [inherits](https://www.javascripttutorial.net/javascript-prototypal-inheritance/) from another object, the for...in statement goes up in the [prototype](https://www.javascripttutorial.net/javascript-prototype/) chain and enumerates over inherited properties. Consider the following example:

var decoration = {

color: 'red'

};

var circle = Object.create(decoration);

circle.radius = 10;

for(const prop in circle) {

console.log(prop);

}

Code language: JavaScript (javascript)

Output:

radius

color

The circle object has its own prototype that references the decoration object. Therefore, the for...in loop displays the properties of the circle object and its prototype.

If you want to enumerate only the [own properties](https://www.javascripttutorial.net/javascript-own-properties/) of an object, you use the hasOwnProperty() method:

for(const prop in circle) {

if(circle.hasOwnProperty(prop)) {

console.log(prop);

}

}

Code language: JavaScript (javascript)

Output:

radius

The for...in loop and Array

It’s good practice to not use the for...in to iterate over an [array](https://www.javascripttutorial.net/javascript-array/), especially when the order of the array elements is important.

The following example works flawlessly:

const items = [10 , 20, 30];

let total = 0;

for(const item in items) {

total += items[item];

}

console.log(total);

Code language: JavaScript (javascript)

However, someone may set a property of the built-in [Array](https://www.javascripttutorial.net/javascript-array/) type in their libraries as follows:

Array.prototype.foo = 100;

Code language: JavaScript (javascript)

Hence, the for...in will not work correctly. For example:

*// somewhere else*

Array.prototype.foo = 100;

const items = [10, 20, 30];

let total = 0;

for (var prop in items) {

console.log({ prop, value: items[prop] });

total += items[prop];

}

console.log(total);

Code language: JavaScript (javascript)

Output:

{ prop: '0', value: 10 }

{ prop: '1', value: 20 }

{ prop: '2', value: 30 }

{ prop: 'foo', value: 100 }

160

Code language: CSS (css)

Or another example:

var arr = [];

*// set the third element to 3, other elements are `undefined`*

arr[2] = 3;

for (let i = 0; i < arr.length; i++) {

console.log(arr[i]);

}

Code language: JavaScript (javascript)

The output shows three elements of the array, which is correct:

undefined

undefined

3

Code language: JavaScript (javascript)

However, the for...in loop ignores the first two elements:

for (const key in arr) {

console.log(arr[key]);

}

Code language: JavaScript (javascript)

Output:

3

The output shows only the third element, not the first two elements.

Summary

* The for...in loop iterates over the [enumerable properties](https://www.javascripttutorial.net/javascript-enumerable-properties/) of an object. It also goes up to the [prototype](https://www.javascripttutorial.net/javascript-prototype/) chain and enumerates inherited properties.
* Avoid using for...in loop to iterate over elements of an array, especially when the index order is important.

A Basic Guide to Enumerable Properties of an Object in JavaScript

**Summary**: in this tutorial, you will learn about JavaScript enumerable properties of an object.

Introduction to JavaScript enumerable properties

Enumerable properties are iterated using the for...in loop or Objects.keys() method.

In JavaScript, an [object](https://www.javascripttutorial.net/javascript-objects/) is an unordered list of key-value pairs. The key is usually a [string](https://www.javascripttutorial.net/javascript-string/) or a [symbol](https://www.javascripttutorial.net/es6/symbol/). The value can be a value of any primitive type (string, boolean, number, undefined, or null), an object, or a [function](https://www.javascripttutorial.net/javascript-function/).

The following example creates a new object using the [object literal syntax](https://www.javascripttutorial.net/es6/object-literal-extensions/):

const person = {

firstName: 'John',

lastName: 'Doe

};

Code language: PHP (php)

The person object has two properties: firstName and lastName.

An object property has several internal attributes including value, writable, enumerable and configurable. See the [Object properties](https://www.javascripttutorial.net/javascript-object-properties/) for more details.

The enumerable attribute determines whether or not a property is accessible when the object’s properties are enumerated using the [for...in](https://www.javascripttutorial.net/javascript-for-in/) loop or Object.keys() method.

By default, all properties created via a simple assignment or via a property initializer are enumerable. For example:

const person = {

firstName: 'John',

lastName: 'Doe'

};

person.age = 25;

for (const key in person) {

console.log(key);

}

Code language: JavaScript (javascript)

Output:

firstName

lastName

age

In this example:

* The firstName and lastName are enumerable properties because they are created via a property initializer.
* The age property is also enumerable because it is created via a simple assignment.

To change the internal enumerable attribute of a property, you use the Object.defineProperty() method. For example:

const person = {

firstName: 'John',

lastName: 'Doe'

};

person.age = 25;

Object.defineProperty(person, 'ssn', {

enumerable: false,

value: '123-456-7890'

});

for (const key in person) {

console.log(key);

}

Code language: JavaScript (javascript)

Output:

firstName

lastName

age

In this example, the ssn property is created with the enumerable flag sets to false, therefore it does not show up in the for...in loop.

ES6 provides a method propertyIsEnumerable() that determines whether or not a property is enumerable. It returns true if the property is enumerable; otherwise false. For example:

const person = {

firstName: 'John',

lastName: 'Doe'

};

person.age = 25;

Object.defineProperty(person, 'ssn', {

enumerable: false,

value: '123-456-7890'

});

console.log(person.propertyIsEnumerable('firstName')); *// => true*

console.log(person.propertyIsEnumerable('lastName')); *// => true*

console.log(person.propertyIsEnumerable('age')); *// => true*

console.log(person.propertyIsEnumerable('ssn')); *// => false*

Code language: JavaScript (javascript)

Summary

* A property is enumerable if it has the enumerable attribute sets to true. The obj.propertyIsEnumerable() determines whether or not a property is enumerable.
* A property created via a simple assignment or a property initializer is enumerable.

Understanding Own Properties of an Object in JavaScript

**Summary**: in this tutorial, you will learn about the **own** properties of an object in JavaScript.

In JavaScript, an [object](https://www.javascripttutorial.net/javascript-objects/) is a collection of [properties](https://www.javascripttutorial.net/javascript-object-properties/), where each property a key-value pair.

This example creates a new object called person using an object initializer:

const person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

The person object has two properties: firstName and lastName.

JavaScript uses [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/). Therefore, a property of an object can be either **own** or **inherited**.

A property that is defined directly on an object is **own** while a property that the object receives from its prototype is **inherited**.

The following creates an object called employee that inherits from the person object:

const employee = Object.create(person, {

job: {

value: 'JS Developer',

enumerable: true

}

});

Code language: JavaScript (javascript)

The employee object has its own property job, and inherits firstName and lastName properties from its prototype person.

The hasOwnProperty() method returns true if a property is own. For example:

console.log(employee.hasOwnProperty('job')); *// => true*

console.log(employee.hasOwnProperty('firstName')); *// => false*

console.log(employee.hasOwnProperty('lastName')); *// => false*

console.log(employee.hasOwnProperty('ssn')); *// => false*

Code language: JavaScript (javascript)

Summary

* A property that is directly defined on an object is an own property.
* The obj.hasOwnProperty() method determines whether or not a property is own.

# JavaScript Object.values()

**Summary**: in this tutorial, you will learn how to use the JavaScript Object.values() method to access the own enumerable properties of an object.

Prior to ES2017, you use a [for...in](https://www.javascripttutorial.net/javascript-for-in/) loop and Object.hasOwnProperty()  method to access values of [own](https://www.javascripttutorial.net/javascript-own-properties/) [enumerable properties](https://www.javascripttutorial.net/javascript-enumerable-properties/) of an [object](https://www.javascripttutorial.net/javascript-objects/). For example:

const person = {

firstName: 'John',

lastName: 'Doe',

age: 25

};

for (const key in person) {

if (person.hasOwnProperty(key)) {

const value = person[key];

console.log(value);

}

}

Code language: JavaScript (javascript)

Output:

John

Doe

25

ES2017 introduces a new method called Object.values() that allows you to return an [array](https://www.javascripttutorial.net/javascript-array/) of own enumerable property’s values of an object.

The following shows the syntax of the Object.values():

Object.values(obj)

Code language: JavaScript (javascript)

The Object.values() accepts an object and returns its own enumerable property’s values as an array. See the following example:

const person = {

firstName: 'John',

lastName: 'Doe',

age: 25

};

const profile = Object.values(person);

console.log(profile);

Code language: JavaScript (javascript)

Output:

[ 'John', 'Doe', 25 ]

Code language: JSON / JSON with Comments (json)

## Object.values() vs. for...in

The Object.values() returns own enumerable properties while the for...in loop iterates properties in the prototype chain.

Technically, if you use the for...in loop with the Object.hasOwnProperty() method, you will get the same set of values as the Object.values().

JavaScript Object.entries()

**Summary**: in this tutorial, you will learn how to use the JavaScript Object.entries() method.

Introduction to JavaScript Object.entries() method

ES2017 introduces the Object.entries() method that accepts an object and returns its own enumerable string-keyed property [key, value] pairs of the object.

Here is the syntax of the Object.entries() method:

Object.entries()

Code language: JavaScript (javascript)

See the following example:

const ssn = Symbol('ssn');

const person = {

firstName: 'John',

lastName: 'Doe',

age: 25,

[ssn]: '123-345-789'

};

const kv = Object.entries(person);

console.log(kv);

Code language: JavaScript (javascript)

Output:

[

['firstName', 'John'],

['lastName', 'Doe'],

['age', 25]

]

Code language: JSON / JSON with Comments (json)

In this example:

* The firstName, lastName, and age are own enumerable string-keyed property of the person object, therefore, they are included in the result.
* The ssn is not a string-key property of the person object, so it is not included in the result.

Object.entries() vs. for...in

The main difference between the Object.entries() and the for...in loop is that the for...in loop also enumerates object [properties](https://www.javascripttutorial.net/javascript-object-properties/) in the [prototype chain](https://www.javascripttutorial.net/javascript-prototype/).

JavaScript Object.is()

**Summary**: in this tutorial, you will learn about the JavaScript Object.is() to check if two values are the same.

The Object.is() behaves like the === operator with two differences:

* -0 and +0
* NaN

Negative zero

The === operator treats -0 and +0 are the same value:

let amount = +0,

volume = -0;

console.log(volume === amount);

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

However, the Object.is() treats +0 and -0 as different values. For example:

let amount = +0,

volume = -0;

console.log(Object.is(amount, volume));

Code language: JavaScript (javascript)

Output

false

Code language: JavaScript (javascript)

NaN

The === operator considers NaN and NaN are different values. The NaN is the only number that does not equal itself. For example:

let quantity = NaN;

console.log(quantity === quantity);

Code language: JavaScript (javascript)

Output:

false

However, Object.is() treats NaN as the same value:

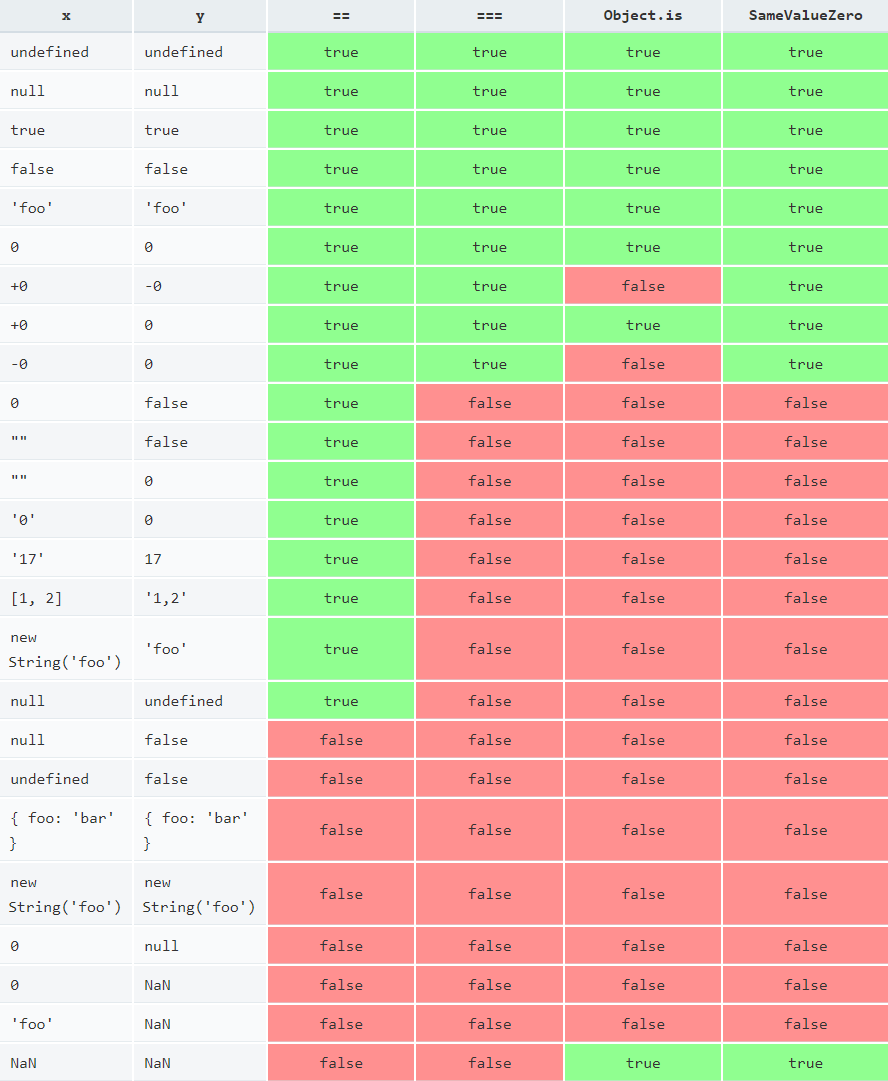
let quantity = NaN;

console.log(Object.is(quantity, quantity));

Output:

true

See the following sameness comparison table for reference:



JavaScript Factory Functions

**Summary**: in this tutorial, you will learn about the JavaScript factory functions which are functions that return objects.

Introduction to the factory functions in JavaScript

A factory function is a [function](https://www.javascripttutorial.net/javascript-function/) that returns a new [object](https://www.javascripttutorial.net/javascript-objects/). The following creates a person object named person1:

let person1 = {

firstName: 'John',

lastName: 'Doe',

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

console.log(person1.getFullName());

Code language: JavaScript (javascript)

Output:

John Doe

The person1 object has two properties: firstName and lastName, and one method getFullName() that returns the full name.

Suppose that you need to create another similar object called person2, you can duplicate the code as follows:

let person2 = {

firstName: 'Jane',

lastName: 'Doe',

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

console.log(person2.getFullName());

Code language: JavaScript (javascript)

Output:

Jane Doe

In this example, the person1 and person2 objects have the same properties and methods.

The problem is that the more objects you want to create, the more duplicate code you have.

To avoid copying the same code all over again, you can define a function that creates the person object:

function createPerson(firstName, lastName) {

return {

firstName: firstName,

lastName: lastName,

getFullName() {

return firstName + ' ' + lastName;

},

};

}

Code language: JavaScript (javascript)

When a function creates and returns a new object, it is called a factory function. The createPerson() is a factory function because it returns a new person object.

The following show how to use the createPerson() factory function to create two objects person1 and person2:

function createPerson(firstName, lastName) {

return {

firstName: firstName,

lastName: lastName,

getFullName() {

return firstName + ' ' + lastName;

},

};

}

let person1 = createPerson('John', 'Doe');

let person2 = createPerson('Jane', 'Doe');

console.log(person1.getFullName());

console.log(person2.getFullName());

Code language: JavaScript (javascript)

By using the factory function, you create any number of the person objects without duplicating code.

When you create an object, the JavaScript engine allocates memory to it. If you create many person objects, the JavaScript engine needs lots of memory spaces to store these objects.

However, each person object has a copy of the same getFullName() method. It’s not efficient memory management.

To avoid duplicating the same getFullName() function in every object, you can remove the getFullName() method from the person object:

function createPerson(firstName, lastName) {

return {

firstName: firstName,

lastName: lastName

}

}

Code language: JavaScript (javascript)

And move this method to another object:

var personActions = {

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

Code language: JavaScript (javascript)

And before calling the getFullName() method on the person object, you can assign the method of the personActions object to the person object as follows:

let person1 = createPerson('John', 'Doe');

let person2 = createPerson('Jane', 'Doe');

person1.getFullName = personActions.getFullName;

person2.getFullName = personActions.getFullName;

console.log(person1.getFullName());

console.log(person2.getFullName());

Code language: JavaScript (javascript)

This approach is not scalable if the object has many methods because you have to manually assign them individually. This is why the Object.create() method comes into play.

The Object.create() method

The Object.create() method creates a new object using an existing object as the [prototype](https://www.javascripttutorial.net/javascript-prototype/) of the new object:

Object.create(proto, [propertiesObject])

Code language: CSS (css)

So you can use the Object.create() as follows:

var personActions = {

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

function createPerson(firstName, lastName) {

let person = Object.create(personActions);

person.firstName = firstName;

person.lastName = lastName;

return person;

}

Code language: JavaScript (javascript)

Now, you can create person objects and call the methods of the personActions object:

let person1 = createPerson('John', 'Doe');

let person2 = createPerson('Jane', 'Doe');

console.log(person1.getFullName());

console.log(person2.getFullName());

Code language: JavaScript (javascript)

The code works perfectly fine. However, in practice, you will rarely use the factory functions. Instead, you use [classes](https://www.javascripttutorial.net/es6/javascript-class/) or [constructor/prototype](https://www.javascripttutorial.net/javascript-constructor-prototype/) patterns.

Summary

* A factory function is a function that returns a new object.
* Use Object.create() to create an object using an existing object as a prototype.

JavaScript Object Destructuring

**Summary**: in this tutorial, you’ll learn about JavaScript object destructuring which assigns properties of an object to individual variables.

If you want to learn how to destructure an [array](https://www.javascripttutorial.net/javascript-array/), you can check out the [array destructuring tutorial](https://www.javascripttutorial.net/es6/destructuring/).

Introduction to the JavaScript object destructuring assignment

Suppose you have a person object with two properties: firstName and lastName.

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

Prior to ES6, when you want to assign properties of the person object to variables, you typically do it like this:

let firstName = person.firstName;

let lastName = person.lastName;

Code language: JavaScript (javascript)

ES6 introduces the object destructuring syntax that provides an alternative way to assign [properties](https://www.javascripttutorial.net/javascript-object-properties/) of an [object](https://www.javascripttutorial.net/javascript-objects/) to variables:

let { firstName: fname, lastName: lname } = person;

Code language: JavaScript (javascript)

In this example, the firstName and lastName properties are assigned to the fName and lName variables respectively.

In this syntax:

let { property1: variable1, property2: variable2 } = object;

Code language: JavaScript (javascript)

The identifier before the colon (:) is the property of the object and the identifier after the colon is the variable.

Notice that the property name is always on the left whether it’s an object literal or object destructuring syntax.

If the variables have the same names as the properties of the object, you can make the code more concise as follows:

let { firstName, lastName } = person;

console.log(firstName); *// 'John'*

console.log(lastName); *// 'Doe'*

Code language: JavaScript (javascript)

In this example, we declared two variables firstName and lastName, and assigned the properties of the person object to the variables in the same statement.

It’s possible to separate the declaration and assignment. However, you must surround the variables in parentheses:

({firstName, lastName} = person);

If you don’t use the parentheses, the JavaScript engine will interpret the left-hand side as a block and throw a syntax error.

When you assign a property that does not exist to a variable using the object destructuring, the variable is set to undefined. For example:

let { firstName, lastName, middleName } = person;

console.log(middleName); *// undefined*

Code language: JavaScript (javascript)

In this example, the middleName property doesn’t exist in the person object, therefore, the middleName variable is undefined.

Setting default values

You can assign a default value to the variable when the property of an object doesn’t exist. For example:

let person = {

firstName: 'John',

lastName: 'Doe',

currentAge: 28

};

let { firstName, lastName, middleName = '', currentAge: age = 18 } = person;

console.log(middleName); *// ''*

console.log(age); *// 28*

Code language: JavaScript (javascript)

In this example, we assign an empty string to the middleName variable when the person object doesn’t have the middleName property.

Also, we assign the currentAge property to the age variable with the default value of 18.

However, when the person object does have the middleName property, the assignment works as usual:

let person = {

firstName: 'John',

lastName: 'Doe',

middleName: 'C.',

currentAge: 28

};

let { firstName, lastName, middleName = '', currentAge: age = 18 } = person;

console.log(middleName); *// 'C.'*

console.log(age); *// 28*

Code language: JavaScript (javascript)

Destructuring a null object

A function may return an object or null in some situations. For example:

function getPerson() {

return null;

}

Code language: JavaScript (javascript)

And you use the object destructuring assignment:

let { firstName, lastName } = getPerson();

console.log(firstName, lastName);

Code language: JavaScript (javascript)

The code will throw a TypeError:

TypeError: Cannot destructure property 'firstName' of 'getPerson(...)' as it is null.

Code language: JavaScript (javascript)

To avoid this, you can use the OR operator (||) to fallback the null object to an empty object:

let { firstName, lastName } = getPerson() || {};

Code language: JavaScript (javascript)

Now, no error will occur. And the firstName and lastName will be undefined.

Nested object destructuring

Assuming that you have an employee object which has a name object as the property:

let employee = {

id: 1001,

name: {

firstName: 'John',

lastName: 'Doe'

}

};

Code language: JavaScript (javascript)

The following statement destructures the properties of the nested name object into individual variables:

let {

name: {

firstName,

lastName

}

} = employee;

console.log(firstName); *// John*

console.log(lastName); *// Doe*

Code language: JavaScript (javascript)

It’s possible to do multiple assignement of a property to multiple variables:

let employee = {

id: 1001,

name: {

firstName: 'John',

lastName: 'Doe'

}

};

let {

name: {

firstName,

lastName

},

name

} = employee;

console.log(firstName); *// John*

console.log(lastName); *// Doe*

console.log(name); *// { firstName: 'John', lastName: 'Doe' }*

Code language: JavaScript (javascript)

Destructuring function arguments

Suppose you have a function that displays the person object:

let display = (person) => console.log(`${person.firstName} ${person.lastName}`);

let person = {

firstName: 'John',

lastName: 'Doe'

};

display(person);

Code language: JavaScript (javascript)

It’s possible to destructure the object argument passed into the function like this:

let display = ({firstName, lastName}) => console.log(`${firstName} ${lastName}`);

let person = {

firstName: 'John',

lastName: 'Doe'

};

display(person);

Code language: JavaScript (javascript)

It looks less verbose especially when you use many properties of the argument object. This technique is often used in React.

Summary

* Object destructuring assigns the properties of an object to variables with the same names by default.

JavaScript Optional Chaining Operator

**Summary**: in this tutorial, you’ll learn about the optional chaining operator (?.) that simplifies the way to access values through connected objects.

Introduction to the JavaScript optional chaining operator

The optional chaining operator (?.) allows you to access the value of a [property](https://www.javascripttutorial.net/javascript-object-properties/) located deep within a chain of [objects](https://www.javascripttutorial.net/javascript-objects/) without explicitly checking if each reference in the chain is null or undefined.

If one of the references in the chain is null or undefined, the optional chaining operator (?.) will short circuit and return undefined.

Suppose that you have a function that returns a user object:

function getUser(id) {

if(id <= 0) {

return null;

}

*// get the user from database*

*// and return null if id does not exist*

*// ...*

*// if user was found, return the user*

return {

id: id,

username: 'admin',

profile: {

avatar: '/avatar.png',

language: 'English'

}

}

}

Code language: JavaScript (javascript)

The following uses the getUser() function to access the user profile:

let user = getUser(1);

let profile = user.profile;

Code language: JavaScript (javascript)

However, if you pass the id that is less than or equal to zero or the id doesn’t exist in the database, the getUser() function will return null.

Therefore, before accessing the avatar property, you need to check if the user is not null using the [logical operator](https://www.javascripttutorial.net/javascript-logical-operators/) AND:

let user = getUser(2);

let profile = user && user.profile;

Code language: JavaScript (javascript)

In this example, we confirm that the user is not null or undefined before accessing the value of user.profile property. It prevents the error that would occur if you simply access the user.profile directly without checking the user first.

ES2020 introduced the optional chaining operator denoted by the question mark followed by a dot:

?.

Code language: JavaScript (javascript)

To access a property of an object using the optional chaining operator, you use one of the following:

objectName ?. propertyName

objectName ?. [expression]

Code language: JavaScript (javascript)

The optional chaining operator implicitly checks if the user is not null or undefined before attempting to access the user.profile:

let user = getUser(2);

let profile = user ?. profile;

Code language: JavaScript (javascript)

In this example, if the user is null or undefined, the optional chaining operator (?.) immediately returns undefined.

Technically, it is equivalent to the following:

let user = getUser(2);

let profile = (user !== null || user !== undefined)

? user.profile

: undefined;

Code language: JavaScript (javascript)

Stacking the optional chaining operator

In case the user object returned by the getUser() does not have the profile property. Trying to access the avatar without checking the user.profile first will result in an error.

To avoid the error, you can use the optional chaining operator multiple times like this:

let user = getUser(-1);

let avatar = user ?. profile ?. avatar;

Code language: JavaScript (javascript)

In this case, the avatar is undefined.

Combining with the nullish coalescing operator

If you want to assign a default profile to the user, you can combine the optional chaining operator (?.) with the nullish coalescing operator (??) as follows:

let defaultProfile = { default: '/default.png', language: 'English'};

let user = getUser(2);

let profile = user ?. profile ?? defaultProfile;

Code language: JavaScript (javascript)

In this example, if the user.profile is null or undefined, the profile will take the defaultProfile due to the nullish coalescing operator:

Using optional chaining operator with function calls

Suppose that you have a file API as follows:

let file = {

read() {

return 'file content';

},

write(content) {

console.log(`Writing ${content} to file...`);

return true;

}

};

Code language: JavaScript (javascript)

This example calls the read() method of the file object:

let data = file.read();

console.log(data);

Code language: JavaScript (javascript)

If you call a method that doesn’t exist in the file object, you’ll get a TypeError:

let compressedData = file.compress();

Code language: JavaScript (javascript)

Error:

Uncaught TypeError: file.compress is not a function

Code language: JavaScript (javascript)

However, if you use the optional chaining operator with the method call, the expression will return undefined instead of throwing an error:

let compressedData = file.compress?.();

Code language: JavaScript (javascript)

The compressedData is now undefined.

This is useful when you use an API in which a method might be not available for some reason e.g., a specific browser or device.

The following illustrates the syntax for using the optional chaining operator with a function or method call:

functionName ?. (args)

Code language: JavaScript (javascript)

The optional chaining operator (?.) is also helpful if you have a function with an optional [callback](https://www.javascripttutorial.net/javascript-callback/):

function getUser(id, callback) {

*// get user*

*// ...*

let user = {

id: id,

username: 'admin'

};

*// test if the callback exists*

if ( callback ) {

callback(user);

}

return user;

}

Code language: JavaScript (javascript)

By using the optional chaining operator, you can skip the test if the callback exists:

function getUser(id, callback) {

*// get user*

*// ...*

let user = {

id: id,

username: 'admin'

};

*// test if the callback exists*

callback ?. (user);

return user;

}

Code language: JavaScript (javascript)

Summary

* The optional chaining operator (?.) returns undefined instead of throwing an error if you attempt to access a property of an null or undefined object: obj ?. property.
* Combine the optional chaining operator (?.) with the nullish coalescing operator (??) to assign a default value.
* Use functionName ?. (args) to avoid explicitly checking if the functionName is not undefined or null before invoking it.

# Object Literal Syntax Extensions in ES6

**Summary**: in this tutorial, you will learn about the syntax extensions of the object literal in ES6 that make your code cleaner and more flexible.

The [object](https://www.javascripttutorial.net/javascript-objects/) literal is one of the most popular [patterns for creating objects in JavaScript](https://www.javascripttutorial.net/create-objects-in-javascript/) because of its simplicity. ES6 makes the object literal more succinct and powerful by extending the syntax in some ways.

## Object property initializer shorthand

Prior to ES6, an object literal is a collection of name-value pairs. For example:

function createMachine(name, status) {

return {

name: name,

status: status

};

}

Code language: JavaScript (javascript)

The createMachine() [function](https://www.javascripttutorial.net/javascript-function/) takes two arguments name and status and returns a new object literal with two properties: name and status.

The name and status properties take the values of the name and status parameters. This syntax looks redundant because name and status mentioned twice in both the name and value of properties.

ES6 allows you to eliminate the duplication when a property of an object is the same as the local variable name by including the name without a colon and value.

For example, you can rewrite the createMachine() function in ES6 as follows:

function createMachine(name, status) {

return {

name,

status

};

}

Code language: JavaScript (javascript)

Internally, when a property of an object literal only has a name, the JavaScript engine searches for a variable with the same name in the surrounding scope. If the JavaScript engine can find one, it assigns the property the value of the variable.

In this example, the JavaScript engine assigns the name and status property values of the name and status arguments.

Similarly, you can construct an object literal from local variables as shown in the following example:

let name = 'Computer',

status = 'On';

let machine = {

name,

status

};

Code language: JavaScript (javascript)

## Computed property name

Prior to ES6, you could use the square brackets( [])  to enable the **computed property names** for the properties on objects.

The square brackets allow you to use the string literals and variables as the property names.

See the following example:

let name = 'machine name';

let machine = {

[name]: 'server',

'machine hours': 10000

};

console.log(machine[name]); *// server*

console.log(machine['machine hours']); *// 10000*

Code language: JavaScript (javascript)

The name variable was initialized to a value of 'machine name'. Since both properties of the machine object contains a space, you can only reference them using the square brackets.

In ES6, the computed property name is a part of the object literal syntax, and it uses the square bracket notation.

When a property name is placed inside the square brackets, the JavaScript engine evaluates it as a string. It means that you can use an expression as a property name. For example:

let prefix = 'machine';

let machine = {

[prefix + ' name']: 'server',

[prefix + ' hours']: 10000

};

console.log(machine['machine name']); *// server*

console.log(machine['machine hours']); *// 10000*

Code language: JavaScript (javascript)

The machine object’s properties evaluate to 'machine name' and 'machine hours', therefore you can reference them as the properties of the machine object.

## Concise method syntax

Prior to ES6, when defining a method for an object literal, you need to specify the name and full function definition as shown in the following example:

let server = {

name: "Server",

restart: function () {

console.log("The" + this.name + " is restarting...");

}

};

Code language: JavaScript (javascript)

ES6 makes the syntax for making a method of the object literal more succinct by removing the colon (:) and the function keyword.

The following example rewrites the server object above using the ES6 syntax.

let server = {

name: 'Server',

restart() {

console.log("The" + this.name + " is restarting...");

}

};

Code language: JavaScript (javascript)

This shorthand syntax is also known as the **concise method syntax**. It’s valid to have spaces in the property name. For example:

let server = {

name: 'Server',

restart() {

console.log("The " + this.name + " is restarting...");

},

'starting up'() {

console.log("The " + this.name + " is starting up!");

}

};

server['starting up']();

Code language: JavaScript (javascript)

In this example, the method 'starting up' has spaces in its name. To call the method, you use the following syntax:

object\_name['property name']();

Code language: CSS (css)

In this tutorial, you have learned how to use some new object literal syntax extensions in ES6 including property initializer shorthand, computed properties, and concise method syntax.