Object

The Object type represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities. Objects can be created using the Object() constructor or the object initializer / literal syntax.

Nearly all objects in JavaScript are instances of Object; a typical object inherits properties (including methods) from Object.prototype, although these properties may be shadowed (a.k.a. overridden). The only objects that don't inherit from Object.prototype are those with null prototype, or descended from other null prototype objects.

The Object type represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities. Objects can be created using the Object() constructor or the object initializer / literal syntax.

**Syntax**

const person = {

name:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

**How to Add Properties in an Object**

There are two major approaches to adding properties to an object in JavaScript. The first is to create a brand-new object.

let person = {

name:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

person = {

name:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

},

isAdmin:true

}

The second approach is to add properties using dot notation.

const person = {

name:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

person.isAdmin = true;

### Why let Keyword Over dot Notation?

The choice between using the let keyword and dot notation depends on whether you need to create a new object reference or modify an existing one.

You would use the let keyword to create a new object when you need to create a new reference to an object, separate from any existing references to similar objects.

On the other hand, you would use dot notation to add or modify properties on an existing object reference. This is useful when you want to make changes to an object without creating a new reference.

### How to Modify Properties in an Object

You can also use dot notation to modify properties in an object. The below code shows that the property name with the pair value kamal will be changed to lawal when modified using dot notation.

const person = {

name:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

person.isAdmin = true;

person.name = 'lawal';

console.log(person);

### How to Delete Properties in an Object

It is very simple to delete a property in an object. JavaScript has a special keyword called "**delete**" that allows you to discard any properties you wish.

const person = {

name:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

person.isAdmin = true;

delete person.friends;

console.log(person);

The property friends will be deleted from the object with the above code.

### How to Use Special Keys in Objects

You can use anything as a key name that you can use as a variable name. But not every key name can serve as a variable name because keys are more flexible than variables.

let person = {

name:'kamal',

age:37,

}

Suppose you want to use "last name" as a key instead of "name". JavaScript syntax does not allow two separate words in this naming convention.

But you can overcome this by using a special key in an object. JavaScript automatically converts any key you enter into a string, even the key "age". As a result, objects in JavaScript are a dictionary of string keys and values of any type.

let person = {

last name:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

To use "last name" as a key in the above code, you need to inform JavaScript that it is a key by enclosing it in quotes, either single or double quotes will work. Using a name that can also be used as a variable is recommended instead of this exception method of setting a key.

let person = {

'last name':'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

Using single quotes to surround "last name" and using it as a key name is valid JavaScript code and will work properly.

### How to Access Properties with Square Brackets

To add and modify properties in an object, you can use the object notation method. However, there is another method in JavaScript for accessing object properties, known as square bracket notation, which enables you to access a property created by a **special key** in

JavaScript.

let person = {

'last name':'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

}

// console.log(person.last name); is not valid in JavaScript

You cannot access the key name 'last name' using the dot notation method, but you can use the square bracket notation, which is available for any object.

console.log(person['last name']);

The above code will display the value of the last name key which is kamal. However, it is crucial to enclose your key within single or double quotes.

### How to Dynamically Set Properties

You can enable another dynamic feature with square brackets and objects in JavaScript. This works when you need to define a new property, especially when you don't know the property name.

For example, you won't know specific user input ahead of time. But, you'll need to add the property with that name to the object.

const userName ='level';

let person = {

'first name':'kamal',

age:30,

[userName]: 'see',

}

console.log(person);

By not wrapping the **userName**in a square bracket, the property with the name userName will be added instead of the value stored inside it. Adding a square bracket to the **userName** will search and take the value stored in the variable as a key name and add it to a person object.

### Method Shorthand Syntax

There is an alternative syntax you can use to define a method in a more efficient way. Traditionally, to create a method in an object, you need a key and a value, where you create the value as a method using the function keyword.

let person = {

name:'jamaldeen',

age:30,

hobbies:[

'reading','playing','sleeping'

],

speek:function(){

return this.hobbies

}

}

console.log(person.speak());

In the above code, the method is created with a function keyword after a colon. Alternatively, you could do this:

let person = {

name:'jamaldeen',

age:30,

hobbies:[

'reading','playing','sleeping'

],

speek(){

return this.hobbies

}

}

console.log(person.speak());

In JavaScript, creating a method with object short methods is a shorthand notation. Omitting the "function" keyword and the colon(:) before the function body is allowed using the short methods.

This is because with the short method syntax, the property is automatically defined as method, which renders the "function" keyword useless.

The code remains functional because the JavaScript engine recognizes the shorthand syntax and interprets it as a regular function definition.

### Advantages of Using Object Short Methods over Regular Methods.

Some of the advantages of using object short methods over regular methods are as follows

1. Conciseness: Unlike regular methods, object short methods allow you to write more compact and readable code.
2. Improved performance: Although, the performance of both object short methods and regular are similar, but the shorter syntax makes it easier to write and maintain your code.
3. Reusability: You can easily reuse object short methods in other objects.
4. Better organization: With object short methods, you can easily group related methods within an object and keep your code organized.

While using object short methods is good, using regular methods may still be more appropriate. It's important to choose the right approach based on your particular requirements.

### Object Spread Operator

The object spread operator is a popular and powerful syntax in JavaScript. The spread operator takes all the key-value pairs of an object and copies the key name and value into a new object.

An object is a reference value, and if you want a copy of the object without pointing to the same property in memory, the spread operator is the answer.

let person = {

name:'kamal',

age:30,

hobbies:[

'reading','playing','sleeping'

]

}

console.log(person);

const person2 ={...person};

console.log(person2.age);

The syntax for object spread operator goes between the opening and closing brackets. Then there should be three dots and the object you want to spread into this object.

### Object Destructuring

Object destructuring is an important feature in JavaScript that allows you to pull out values from an object and assign them to individual variables.

To perform object destructuring, you use a destructuring pattern on the left-hand side of an assignment statement, and the object that you want to extract values from on the right-hand side. For example:

const person = { name: 'lawal', age: 39 };

const { person, age } = person;

console.log(name); // 'lawal'

console.log(age); // 39

The const statement uses object destructuring to extract the name and age properties from the person object and assign them to two separate variables. This is a concise and efficient way to extract values from an object, especially when dealing with complex objects.

Object destructuring also enables you to provide default values, in case the property you want to extract does not exist in the object. You can also rename the variables being extracted using an alias, giving you greater control over the structure and naming of the extracted values.

## **How to Use the this Keyword in JavaScript**

What is the this keyword? this is a specific keyword in JavaScript which is most important when used inside of a function in an object. But you can use it anywhere in your code aside from the function body of an object.

this is a powerful keyword used in referencing the current object in which it's used.

let person = {

name:'kamal',

age:30,

greet:function(){

return `My name is ${this.name}, and my age is ${this.age} years old`;

},

}

console.log(person.greet());

// My name is kamal, and my age is 30 years old.

The code above demonstrates that the "this" keyword refers to the object containing the function, in this case the "person" object, and the result displays the output of the "greet" function.

Regardless of its location within an object, the this keyword always refers to the entity that executed the function in the code. Using this in different contexts within the code can produce distinct results. For instance:

let person = {

name:'kamal',

age:30,

greet:function(){

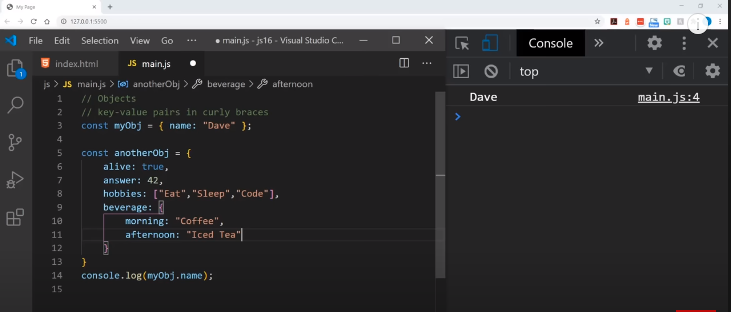
return `My name is ${this.name}, and my age is ${this.age} years old`;

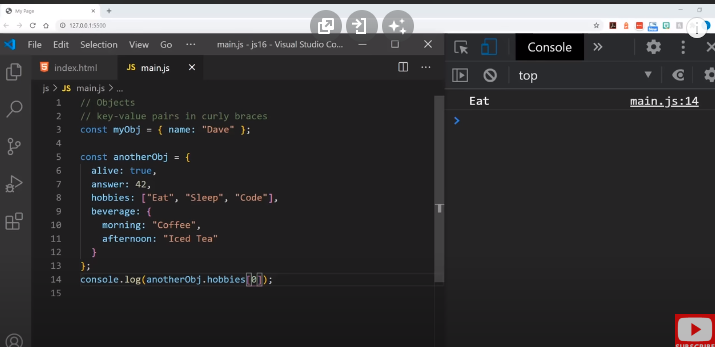
},

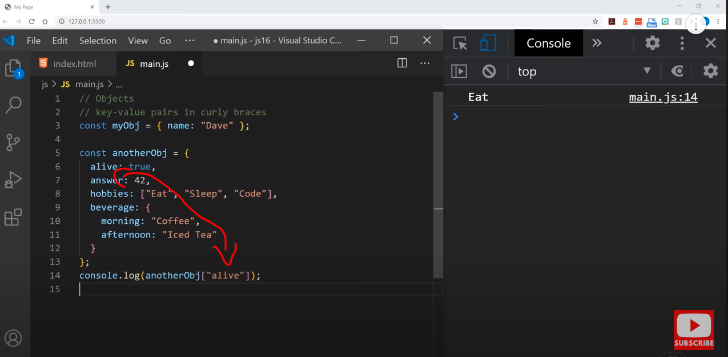
}

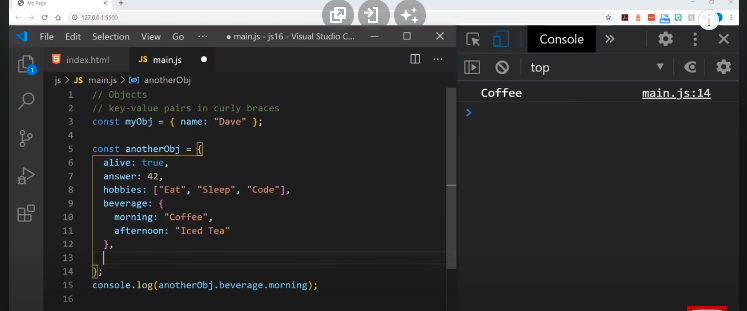
console.log(this);

The output of the code shows that the this keyword when console.logged will print a window object.

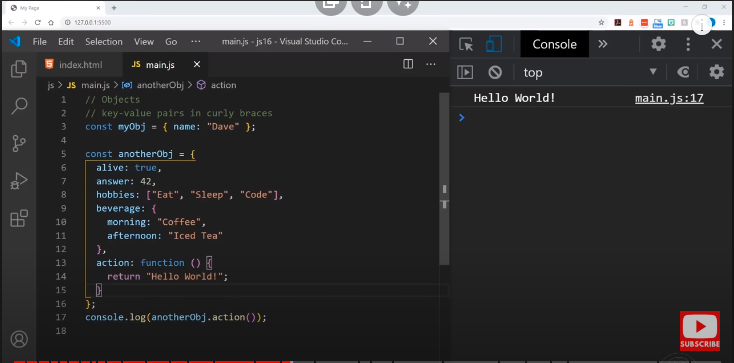


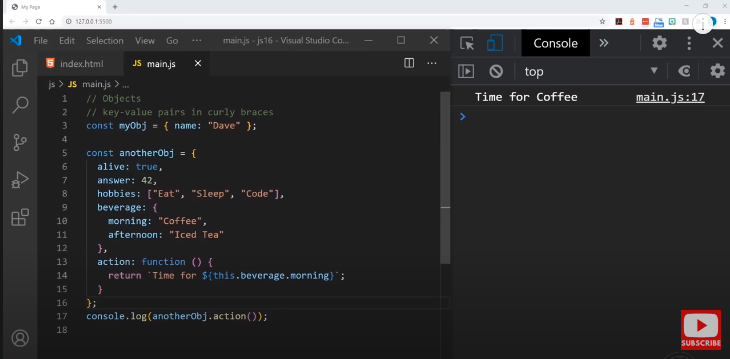






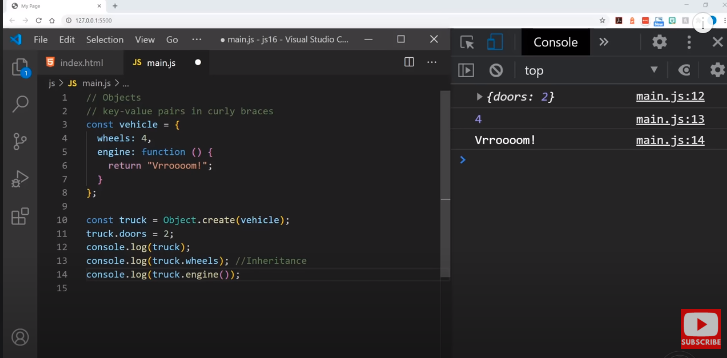
**Object function**

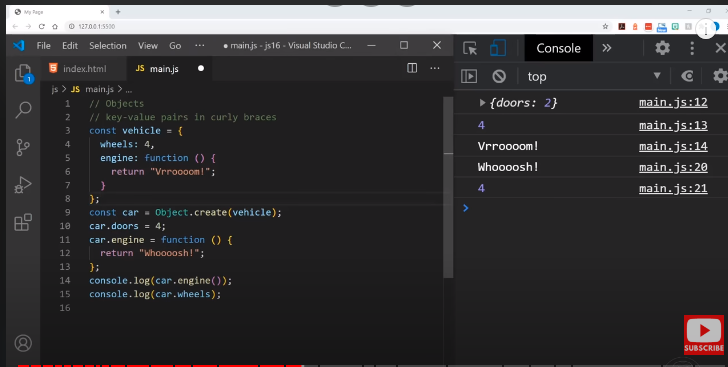


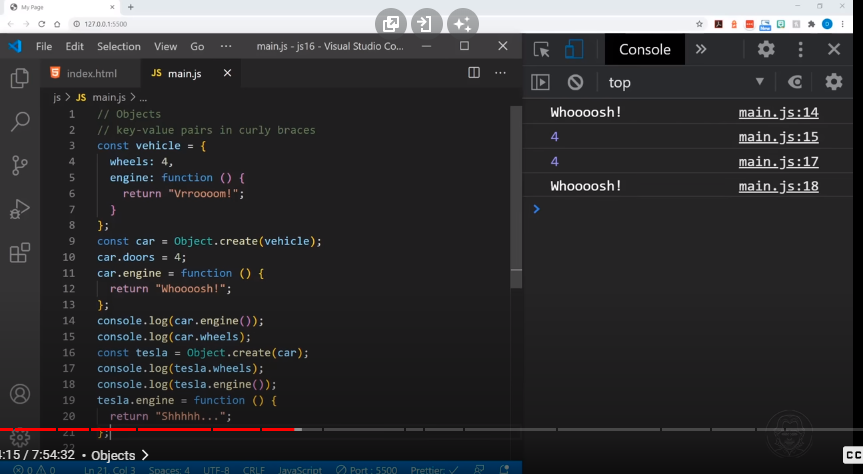


**Object() constructor**

The Object() constructor turns the input into an object. Its behavior depends on the input's type.







Syntax

JS

new Object(value)

Object(value)

Note: Object() can be called with or without new, but sometimes with different effects. See Return value.

**Parameters**

value

1. Any value.
2. Return value

When the Object() constructor itself is called or constructed, its return value is an object:

If the value is null or undefined, it creates and returns an empty object.

If the value is an object already, it returns the value.

Otherwise, it returns an object of a type that corresponds to the given value. For example, passing a BigInt primitive returns a BigInt wrapper object.

When Object() is implicitly called via super() in the constructor of a class that extends Object, it initializes a new object with new.target.prototype as its prototype. Any value passed to super() is ignored — for example, even if you pass a number, the this value inside the constructor does not become a Number instance.

Examples

Creating a new Object

JS

const o = new Object();

o.foo = 42;

console.log(o);

// { foo: 42 }

Using Object given undefined and null types

The following examples store an empty Object object in o:

JS

const o = new Object();

JS

const o = new Object(undefined);

JS

const o = new Object(null);

Obtaining wrapper objects for BigInt and Symbol

The BigInt() and Symbol() constructors throw an error when called with new, to prevent the common mistake of creating a wrapper object instead of the primitive value. The only way to create a wrapper object for these types is to call Object() with them:

JS

const numberObj = new Number(1);

console.log(typeof numberObj); // "object"

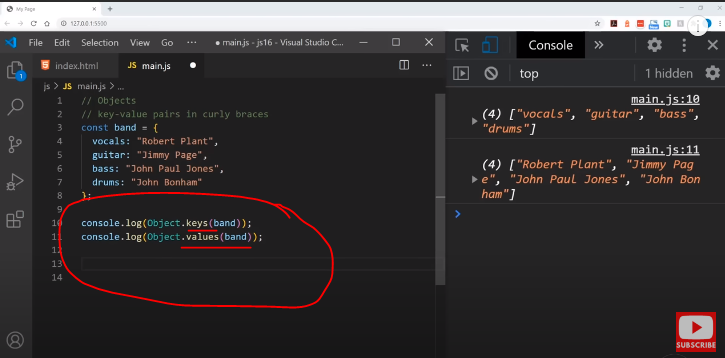
const bigintObj = Object(1n);

console.log(typeof bigintObj); // "object"

const symbolObj = Object(Symbol("foo"));

console.log(typeof symbolObj); // "object"

**objects keys and values:**



**Destructuring objects:**

let person = {

firstName: 'John',

lastName: 'Doe'

};

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;

ES6 introduces the object destructuring syntax that provides an alternative way to assign properties of an object to variables:

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

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;