10 Tips and Tricks to Handle JavaScript Objects

Charuka Herath • 8 min read • Nov 19, 2024 • Updated

An object is the basic building block of programs in JavaScript, used in building classes and complex data, and as an integral part of object-oriented programming.

I have used JavaScript daily as a full-stack software developer for the last five-odd years. Objects of JavaScript have played a vital role.

In this article, I will share 10 tricks and tips you can use as a JavaScript developer to manipulate and work efficiently with JavaScript objects.

1. Creating an absolutely empty object

Creating empty objects seems like nothing but using **{}**. However, have you noticed that object methods such as **proto** and **hasOwnProperty** are still there when you create an object in such a method? This is because using **{}** will create an object that inherits from the **Object** class.

If you need to create an absolutely empty object, it's better to use **Object.create(null)**, which creates an object that is not inherited from anything and has no properties.

```
let vehical = Object.create(null);

// vehicle.__proto__ === "undefined"
// There are no object properties, keys, or methods until we add
```


2. Combining two objects using the spread operator

There are many scenarios where you must combine two or more data sets from different sources. In such cases, there are multiple ways to do this in JavaScript.

The most commonly used method is using **Object.assign()**. This method takes multiple parameters. The first one is the assigned object, and the rest of the parameters are the objects we need to combine.

Copy

```
const name = { id: '1234', name: 'Charuka'};
const university = { id: '1234', university: 'Harvard'};
const PersonalDetails = Object.assign({}, name, university);

console.log(PersonalDetails);
// { id: '1234', name: 'Charuka', university: 'Harvard' }
```

However, without complicating things, you can use the spread operator to combine. You can spread any number of objects to combine them into a single object.

Copy

```
const PersonalDetails = { ...name, ...university };

console.log(PersonalDetails);
// { id: '1234', name: 'Charuka', university: 'Harvard' }
```

One important fact to note is that duplicate keys will override those of the preceding objects in both methods.

3a. Getting the lists of keys and values from an object

During development, there are occasions when we need to obtain only keys or only values from an object. Both of the following built-in functions are pretty straightforward:

- **Object.keys()**: used to get the list of keys.
- **Object.values()**: used to get the list of values.

```
const vehicle = { brand: 'BWM', year: 2022, type: 'suv'};
//get keys
console.log(Object.keys(vehicle)); // [ 'brand', 'year', 'type' ]

//get values
console.log(Object.values(vehicle)); // [ 'BWM', 2022, 'suv' ]
```

3b. Using hasOwnProperty() to check an item

When using a **for-in** loop, checking a property of an object can be useful to avoid iterating through the properties from the object's prototype. Instead of using an **if-else** block, here we can use **Object.hasOwnProperty()**.

```
const vehicle = { brand: 'BWM', year: 2022, type: 'suv'};
for (var item in vehicle) {
    if (vehicle.hasOwnProperty(item)) {
        console.log(item);
    };
};
// brand
// year
// type
```


4. Using splice instead of delete

When using the **delete** method, an application will replace an item with **undefined** instead of removing it from the array. So it is better to use **splice()** to delete an item from an array.

Let’s see what happens when using **delete**.

```
var arrayItems = ['a' , 2 , 'b', '3', 'c', '4'];
arrayItems.length; // returns 6
delete arrayItems[2]; // returns true
arrayItems.length; // returns 6
console.log(arrayItems); // [ 'a', 2, undefined, '3', 'c', '4' ]
```

When using **splice()**, the following occurs.

```
var arrayItems = ['a' , 2 , 'b', '3', 'c', '4'];
arrayItems.length; // returns 6
arrayItems.splice(2,1); // returns true
arrayItems.length; // returns 5
console.log(arrayItems); // [ 'a', 2, '3', 'c', '4' ]
```

The **delete** method should be used to delete an object property.

5. Cloning an object correctly

Assume you have an object and need to copy it to change its value, but the original object should be unchanged. There are two methods of how you can do that.

The first method is to use **Object.assign()**, which copies values of all enumerable properties from one object to another.

```
var initialVehicle = { brand: 'BWM', year: 2022, type: 'suv'};
var secondaryVehicle = Object.assign({}, initialVehicle);
console.log(secondaryVehicle); // { brand: 'BWM', year: 2022, type: 'suv'};
```

The second method is to copy the object using **JSON.parse()**.

```
var initialVehicle = { brand: 'BWM', year: 2022, type: 'suv'};
var secondaryVehicle = JSON.parse(JSON.stringify(initialVehicle));
console.log(secondaryVehicle); // { brand: 'BWM', year: 2022, type: 'suv'};
```

6. Selecting specific data from an object

There are a few methods to select keys from an object. The method you choose depends on what you want to do with the values. The following example shows an organized way of selecting data from an object.

Here, you can select the keys you need and pull them into a new object.

Copy

```
const selectObj = (obj, items) => {
  return items.reduce((result, item) => {
    result[item] = obj[item];
    return result;
  }, {});
};
const vehicle = { brand: 'BWM', year: 2022, type: 'suv'};
const selected = selectObj(vehicle, ['brand', 'type']);
console.log(selected); // { brand: 'BWM', type: 'suv' }
```


7. Removing keys from an object

Sometimes it is necessary to remove specific keys and their values from an object.

This might be necessary for a scenario where you are building an API and want to remove sensitive data.

The most suitable method is to write a reusable **remove** method that takes an object and a list of keys to be removed as inputs. You can then loop through each key to be removed and delete it from the object.

Copy

```
const remove = (object, removeList = []) => {
  const result = { ...object };
  removeList.forEach((item) => {
    delete result[item];
  });
  return result;
}

const vehicle = { brand: 'BWM', year: 2022, type: 'suv'}

const itemRemoved = remove(vehicle, ['year']);
console.log(itemRemoved); // Result { brand: 'BWM', type: 'suv' }
```

8. Pulling object data into an array

There are scenarios where you need to pull your object data into an array, such as a dropdown menu. You can use the **Object.entries()** function, which takes an object as its first argument and returns an array.

The returned object is an array of an array. The inner arrays will have two values: the first is the key, and the second is the value.

```
const vehicle = { brand: 'BWM', year: 2022, type: 'suv'}
console.log(Object.entries(vehicle));
// [ [ 'brand', 'BWM' ], [ 'year', 2022 ], [ 'type', 'suv' ] ]
```

9. Looping through a JavaScript object

There are several methods in JavaScript that can be used to loop through an object. I will compare two of the best methods I use.

The first method is to use **Object.entries()**, a function that avoids looking up each value in the original object.

```
const vehicle = { brand: 'BWM', year: 2022, type: 'suv'}
Object.entries(vehicle).forEach(
    ([key, value]) => console.log(key, value)
);
// brand BWM
```

```
// year 2022
// type suv
```

As a much better and clearer method, you can use object destructuring with **Object.entries()**.

Copy

```
const vehicle = { brand: 'BWM', year: 2022, type: 'suv'}
for (const [key, value] of Object.entries(vehicle)) {
    console.log(key, value);
}
// brand BWM
// year 2022
// type suv
```


10. Conditionally adding attributes to objects

Usually, developers use an **if-else** condition as a much longer method to add a new element to an object conditionally. However, the simplest way is to use object

destructuring and the spread operator.

```
const type = { type: 'suv' };
const vehicle = {
  brand: 'BMW',
  year: 2022,
  ...(!type ? {} : type)
}
console.log(vehicle); //{ brand: 'BMW', year: 2022, type: 'suv' }
```

Likewise, using different conditions, you can add as many elements as you like to an object.

Conclusion

Like any other programming language, JavaScript has many tricks to handle objects, letting us write our programs more simply and beautifully. This article discussed 10 of the tips and tricks I use most when dealing with objects.

I hope you found this article helpful. Thank you for reading.

Syncfusion's Essential JS 2 is the only suite you will need to build an app. It contains over 65 high-performance, lightweight, modular, and responsive UI components in

a single package. Download a free trial to evaluate them today.

Please let us know if you have any questions in the comments section below. You may also get in touch with us via our support forum, support portal, or feedback portal. We are delighted to assist you!

Related blogs

- Null vs. Undefined in JavaScript
- 7 Functional Programming Techniques for JavaScript Developers
- JavaScript String Manipulation Techniques Every Developer Should Know
- JavaScript Debugging with VS Code and Chrome

MEET THE AUTHOR

Charuka Herath

I am a full-stack software Engineer and a person who has a passion and loves writing and reading. I have more than 05 years of experience in the IT industry, as an Engineer, as a content writer and as an editor.