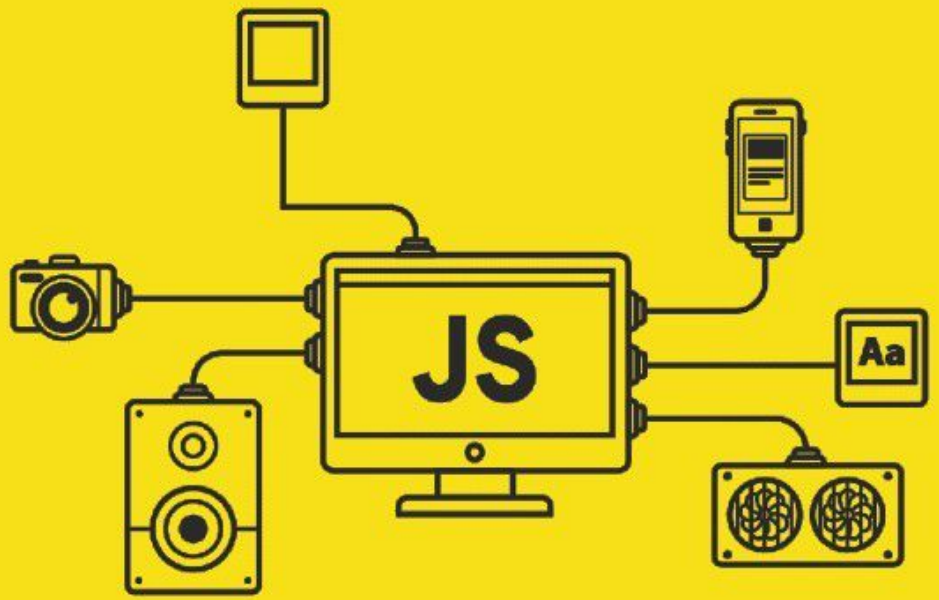

JS

JavaScript best practices

Swipe-up

1

1. Consider Using let and const

The let keyword allows us to create local variables that are scoped within their own block. The const keyword allows us to create local block-scoped variables whose value cannot be reassigned. You should consider using the let and const keywords in appropriate situations when declaring your variables. Keep in mind that the const keyword only prevents reassignment. It does not make the variable immutable.

```
var person_name = "Adam";
let name_length = person_name.length;
const fav_website = "code.tutsplus.com";

if(person_name.length < 5) {
    var person_name = "Andrew";
    let name_length = person_name.length;

    // Throws an error if not commented out!
    // fav_website = "webdesign.tutsplus.com";

    console.log(`${person_name} has ${name_length} characters.`);
    // Output: Andrew has 6 characters.
}

console.log(`${person_name} has ${name_length} characters.`);
// Output: Andrew has 4 characters.
```

In the above example, the value of the person_name variable was updated outside the if block as well, after we modified it inside the block. On the other hand, name_length was block-scoped, so it retained its original value outside the block.

Swipe-up

JS

2

2. eval() Is Bad

For those unfamiliar, the eval() function gives us access to JavaScript's compiler. Essentially, we can execute a string's result by passing it as a parameter of eval().

Not only will this decrease your script's performance substantially, but it also poses a huge security risk because it grants far too much power to the passed-in text. Avoid it!

3

3. Comment Your Code

It might seem unnecessary at first, but trust me, you want to comment your code as well as possible.
What happens when you return to the project months later, only to find that you can't easily remember what your line of thinking was? Or what if one of your colleagues needs to revise your code? Always, always comment important sections of your code.

```
// Cycle through array and echo out each name.
for(var i = 0, len = array.length; i < len; i++) {
   console.log(array[i]);
}
```


4

4. Consider Using let and const

The let keyword allows us to create local variables that are scoped within their own block.
The const keyword allows us to create local block-scoped variables whose value cannot be reassigned.
You should consider using the let and const keywords in appropriate situations when declaring your variables.

```
var person_name = "Adam";
let name_length = person_name.length;
const fav_website = "code.tutsplus.com";

if(person_name.length < 5) {
    var person_name = "Andrew";
    let name_length = person_name.length;

    // Throws an error if not commented out!
    // fav_website = "webdesign.tutsplus.com";

    console.log(`${person_name} has ${name_length} characters.`);
    // Output: Andrew has 6 characters.
}

console.log(`${person_name} has ${name_length} characters.`);
// Output: Andrew has 4 characters.
```

In the above example, the value of the person_name variable was updated outside the if block as well, after we modified it inside the block. On the other hand, name_length was block-scoped, so it retained its original value outside the block.

5

5. Give descriptive names to variables and functions

Make the names of your variables, functions, and other code structures easy to understand for anyone who works with your code. Don’t use names that are either too short or too long, and take care that they succinctly describe their own purpose.

Some good examples are (from the DOM API): - Properties (variables): firstChild, isConnected, nextSibling - Methods (functions): getResponseHeader(), addEventListener() - Objects: XMLHttpRequest, NodeList - Interfaces: HTMLCollection, EventTarget

Also keep in mind that JavaScript is a case-sensitive language. As camel case capitalization is the most common naming system in JavaScript programming, you should use lower camelcase for variables and functions and upper camelcase for classes and interfaces for better code readability.

6

6. Use shorthands, but be careful with them

JavaScript includes many shorthands that allow you to write code faster and make your scripts load faster.

There are older shorthands that have been part of the JavaScript syntax for ages, such as the if true shorthand:

```
/* Longhand )*/
if (value = true) { … }

/* Shorthand */
if (value) { … }
```

And of course, there are newer ones, too, such as arrow functions introduced in ECMAScript 6:

```
// Longhand
let add = function(a, b) {
    return a + b;
}

// Shorthand
let add = (a,b) => a + b;
```

Sometimes, however, shorthands might return surprising results. So, always be sure of what you're doing, check the documentation, find relevant JavaScript code examples, and test the outcome.

Swipe-up

JS

7

7. Use for...of instead of for loops

Loops can get costly performance-wise because you repeat the same operation over and over again. However, if you optimize them, you can make them run faster.

There are many JavaScript best practices to write more performant loops, such as avoiding nesting, keeping DOM manipulation outside of loops, and declaring a separate variable for the length of the loop (e.g. let cityCount = cities.length).

Using the for...of statement instead of for is such a JavaScript coding practice, too. This syntax has been introduced by ECMAScript 6, and it includes a built-in iterator so that you don’t have to define the i variable and the length value:

7

7. Use for...of instead of for loops

```
// with for loop
let cities = ["New York", "Paris", "Beijing", "Sao Paulo", "Auckland"];
let cityCount = cities.length;

for(let i = 0; i < cityCount; i++) {
    console.log( cities[i] );
}

// with for...of loop
let cities = ["New York", "Paris", "Beijing", "Sao Paulo", "Auckland"];

for(city of cities) {
    console.log(city);
}
```

You can use the for...of loop to iterate over any iterable object, such as arrays, strings, nodelists, maps, and more.

8

8. Use {} Instead of new Object()

There are multiple ways to create objects in JavaScript. Perhaps the more traditional method is to use the new constructor, like so:

```
var o = new Object();
o.name = 'Jeffrey';
o.lastName = 'Way';
o.someFunction = function() {
   console.log(this.name);
}
```

However, this method receives the "bad practice" stamp. It's not actually harmful, but it is a bit verbose and unusual. Instead, I recommend that you use the object literal method.

9

9. Learn Unit Testing

When I first started adding unit tests as a developer, I frequently discovered bugs. Tests are the most effective way to ensure that your code is error-free. Jest is a great place to start, but there are other options that are just as easy to use. Before any code is deployed, it should be subjected to unit testing to ensure that it fulfills quality standards. This promotes a dependable engineering environment that prioritizes quality. Unit testing saves time and money during the product development lifecycle, and it helps developers design better, more efficient code.

8

8. Use {} Instead of new Object()

Better

```
var o = {
   name: 'Jeffrey',
   lastName = 'Way',
   someFunction : function() {
      console.log(this.name);
   }
};
```

Note that if you simply want to create an empty object, {} will do the trick.

```
var o = {};
```


10

10. Use the Spread Operator

Have you ever been in a situation where you wanted to pass all the items of an array as individual elements to some other function or you wanted to insert all the values from one array into another? The spread operator (...) allows us to do exactly that.

Here is an example:

```
let people = ["adam", "monty", "andrew"]
let more_people = ["james", "jack", ...people, "sajal"]

console.log(more_people)
// Output: Array(6) [ "james", "jack", "adam", "monty", "andrew", "sajal" ]
```

JS

Swipe-up

11

11. Don't initialize things with “undefined”

Something is “undefined” when it lacks value. Let’s agree that assigning “no value” as a “value” for something is a pretty weird concept right? Since JavaScript already makes things “undefined” how can you tell whether something is undefined because of you or JavaScript? It makes it hard to debug why things are “undefined” so prefer setting things to “null” instead.

12

12. Be Careful With for ... in Statements

When looping through items in an object, you might find that you retrieve method functions or other inherited properties as well. In order to work around this, always wrap your code in an if statement which filters with hasOwnProperty.

```
for (key in object) {
   if (object.hasOwnProperty(key) {
      ...then do something...
   }
}
```


13

13. Self-Executing Functions

Rather than calling a function, it's quite simple to make a function run automatically when a page loads or a parent function is called. Simply wrap your function in parentheses, and then append an additional set, which essentially calls the function.

```
(function doSomething() {
    return {
        name: 'jeff',
        lastName: 'way'
    };
})();
```


14

14. async and await

You can use the async keyword to create asynchronous functions, which always return a promise either explicitly or implicitly. Asynchronous functions that you create can take advantage of the await keyword by stopping execution until the resolution of returned promises. The code outside your async function will keep executing normally.

```
async function delayed_hello() {

    console.log("Hello Adam!");

    let promise = new Promise((resolve) => {
        setTimeout(() => resolve("Hello Andrew!"), 2000)
    });

    let result = await promise;

    console.log(result);
}

console.log("Hello Monty!");

delayed_hello();

console.log("Hello Sajal!");

/*
Hello Monty!
Hello Adam!
Hello Sajal!
Hello Andrew!
*/
```


14

14. async and await

n the above example, "Hello Andrew" is logged after two seconds, while all other hellos are logged immediately. The call to the delayed_hello() function logs "Hello Adam" immediately but waits for the promise to resolve in order to log "Hello Andrew".

15

15. Use Arrow Functions

Another essential feature added to JavaScript recently is arrow functions. They come with a slew of benefits. To begin with, they make the functional elements of JavaScript more appealing to the eye and easier to write.

Take a look at how we would implement a filter without arrow functions:

```
const nums = [1,2,3,4,5,6,7,8];

const even_nums = nums.filter( function (num) { return num%2 == 0; } )
```

Here, the callback function we pass to the filter returns true for any even number.

Arrow functions make this much more readable and concise though:

15

15. Use Arrow Functions

```
const nums = [1,2,3,4,5,6,7,8];

const even_nums = nums.filter(num => num%2 == 0)
```

Another notable benefit of arrow functions is that they do not define a scope, instead being within the parent scope. This prevents many of the issues that can occur when using the this keyword. There are no bindings for this in the arrow functions. this has the same value inside the arrow function as it does in the parent scope. However, this means arrow functions can't be used as constructors or methods.

16

16. Read, Read, Read...

While I'm a huge fan of web development blogs (like this one!), there really isn't a substitute for a book when grabbing some lunch, or just before you go to bed. Always keep a web development book on your bedside table. Here are some of my JavaScript favorites.