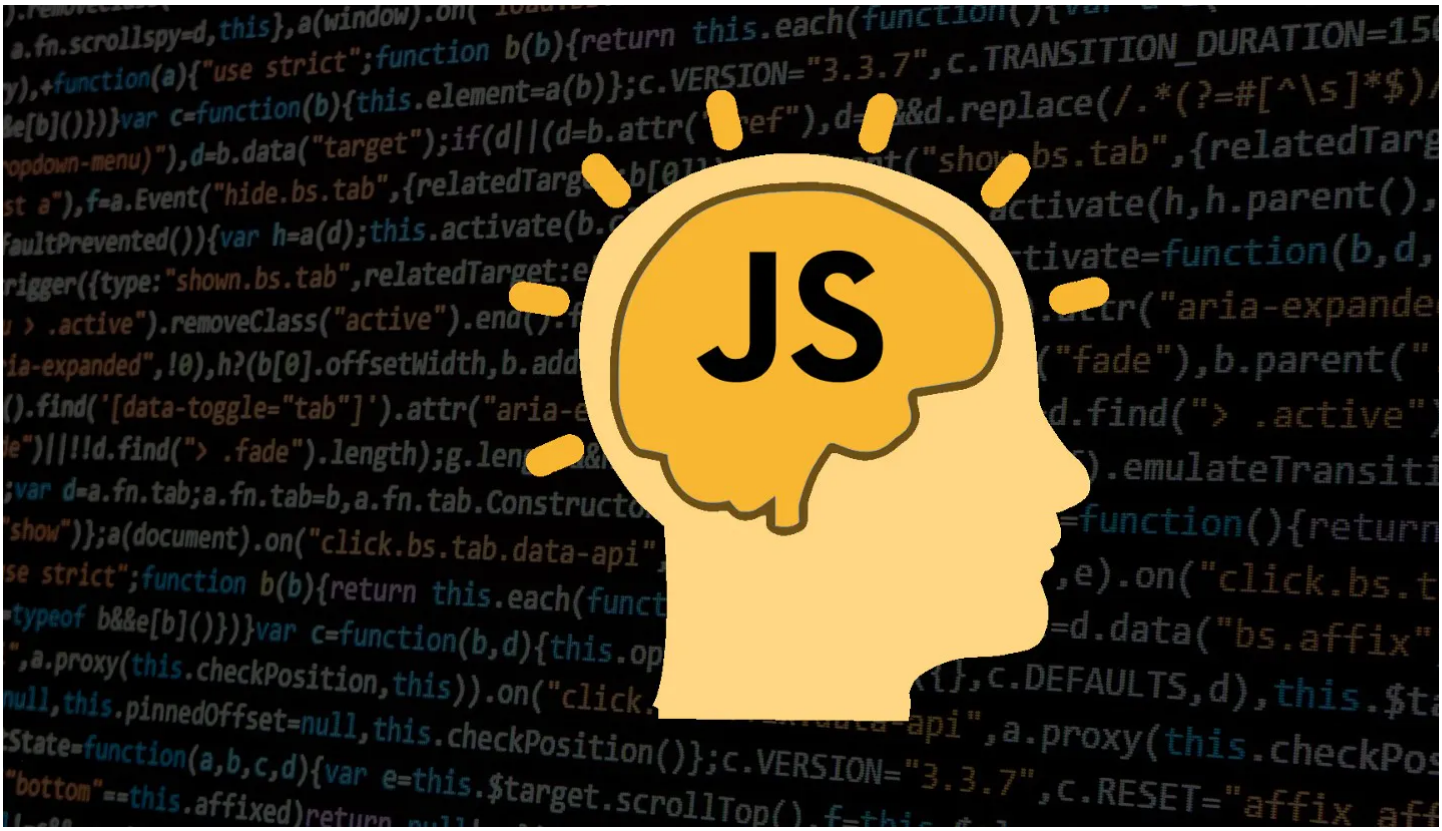
12 JavaScript Concepts That Will Level Up Your Development Skill

Originally published by Nick Scialli on February 10th 2019 ★ 96,421 reads



Audio Presented by

Speed: 1x

Read by:

@nas5w
Nick Scialli

Husband, dog dad, coffee monster. Software engineer at the @usds! Opinions are my own

JavaScript is a complex language. If you're a JavaScript developer at any level, it's important to understand its foundational concepts. This 12 concepts that are critical for any JS developer to understand, but in no way represents the full breadth of what a JS developer should kn

Note: If you enjoy this article, please give it a clap 👏 (or 50!) to help spread the word!

I will be continuously updating this list in a Github repository called JS Tips & Tidbits. Please star ⭐ and share if you want to follow along!

1. Value vs. Reference Variable Assignment

Understanding how JavaScript assigns to variables is foundational to writing bug-free JavaScript. If you don't understand this, you could eas
that unintentionally changes values.

JavaScript *always* assigns variables by value. But this part is **very** important: when the assigned value is one of JavaScript's five primitive ty
`Boolean`, `null`, `undefined`, `String`, and `Number`) the actual value is assigned. However, when the assigned value is an `Array`, `Function`,
reference to the object in memory is assigned.

Example time! In the following snippet, `var2` is set as equal to `var1`. Since `var1` is a primitive type (`String`), `var2` is set as equal to `var1`'s
and can be thought of as completely distinct from `var1` at this point. Accordingly, reassigning `var2` has not effect on `var1.`

```
let var1 = 'My string';
let var2 = var1;
```

```
var2 = 'My new string';
```

```
console.log(var1);
// 'My string'
console.log(var2);
// 'My new string'
```

Let's compare this with object assignment.

```
let var1 = { name: 'Jim' }
let var2 = var1;
```

```
var2.name = 'John';
```

```
console.log(var1);
// { name: 'John' }
console.log(var2);
// { name: 'John' }
```

One might see how this could cause problems if you expected behavior like primitive assignment! This can get especially ugly if you create unintentionally mutates an object.

2. Closures

Closure is an important JavaScript pattern to give private access to a variable. In this example, `createGreeter` returns an anonymous func access to the supplied `greeting`, “Hello.” For all future uses, `sayHello` will have access to this greeting!

```
function createGreeter(greeting) {
  return function(name) {
    console.log(greeting + ', ' + name);
  }
}
```

```
const sayHello = createGreeter('Hello');
sayHello('Joe');
// Hello, Joe
```

In a more real-world scenario, you could envision an initial function `apiConnect(apiKey)` that returns some methods that would use the AP case, the `apiKey` would just need to be provided once and never again.

```
function apiConnect(apiKey) {
  function get(route) {
    return fetch(`${route}?key=${apiKey}`);
  }
```

```
  function post(route, params) {
    return fetch(route, {
      method: 'POST',
      body: JSON.stringify(params),
        headers: {
          'Authorization': `Bearer ${apiKey}`
        }
      })
  }
```

```
  return { get, post }
}
```

```
const api = apiConnect('my-secret-key');
```

```
// No need to include the apiKey anymore
api.get('http://www.example.com/get-endpoint');
api.post('http://www.example.com/post-endpoint', { name: 'Joe' });
```

3. Destructuring

Don't be thrown off by JavaScript parameter destructuring! It's a common way to cleanly extract properties from objects.

```
const obj = {
  name: 'Joe',
  food: 'cake'
}
```

```
const { name, food } = obj;
```

```
console.log(name, food);
// 'Joe' 'cake'
```

If you want to extract properties under a different name, you can specify them using the following format.

```
const obj = {
  name: 'Joe',
  food: 'cake'
}
```

```
const { name: myName, food: myFood } = obj;
```

```
console.log(myName, myFood);
// 'Joe' 'cake'
```

In the following example, destructuring is used to cleanly pass the `person` object to the `introduce` function. In other words, destructuring c often is) used directly for extracting parameters passed to a function. If you're familiar with React, you probably have seen this before!

```
const person = {
  name: 'Eddie',
  age: 24
}
```

```
function introduce({ name, age }) {
  console.log(`I'm ${name} and I'm ${age} years old!`);
}
```

```
console.log(introduce(person));
// "I'm Eddie and I'm 24 years old!"
```

4. Spread Syntax

A JavaScript concept that can throw people off but is relatively simple is the spread operator! In the following case, `Math.max` can't be appli array because it doesn't take an array as an argument, it takes the individual elements as arguments. The spread operator `...` is used to p elements out the array.

```
const arr = [4, 6, -1, 3, 10, 4];

const max = Math.max(...arr);
```

```
console.log(max);
// 10
```

5. Rest Syntax

Let's talk about JavaScript rest syntax. You can use it to put any number of arguments passed to a function into an array!

```
function myFunc(...args) {
  console.log(args[0] + args[1]);
}
```

```
myFunc(1, 2, 3, 4);
// 3
```

6. Array Methods

JavaScript array methods can often provide you incredible, elegant ways to perform the data transformation you need. As a contributor to S
frequently see questions regarding how to manipulate an array of objects in one way or another. This tends to be the perfect use case for a

I will cover a number of different array methods here, organized by similar methods that sometimes get conflated. This list is in no way com
encourage you to review and practice all of them discussed on MDN (my favorite JavaScript reference).

map, filter, reduce
There is some confusion around the JavaScript array methods `map`, `filter`, `reduce`. These are helpful methods for transforming an array o
aggregate value.

- **map:** return array where each element is transformed as specified by the function

```
const arr = [1, 2, 3, 4, 5, 6];
const mapped = arr.map(el => el + 20);
```

```
console.log(mapped);
// [21, 22, 23, 24, 25, 26]
```

- **filter:** return array of elements where the function returns true

```
const arr = [1, 2, 3, 4, 5, 6];
const filtered = arr.filter(el => el === 2 || el === 4);
```

```
console.log(filtered);
// [2, 4]
```

- **reduce:** accumulate values as specified in function

```
const arr = [1, 2, 3, 4, 5, 6];
const reduced = arr.reduce((total, current) => total + current);
```

```
console.log(reduced);
// 21
```

find, findIndex, indexOf

The array methods `find`, `findIndex`, and `indexOf` can often be conflated. Use them as follows.

- **find**: return the first instance that matches the specified criteria. Does not progress to find any other matching instances.

```
const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const found = arr.find(el => el > 5);
```

```
console.log(found);
// 6
```

Again, note that while everything after `5` meets the criteria, only the first matching element is returned. This is actually super helpful in situat would normally `break` a for loop when you find a match!

- **findIndex**: This works almost identically to `find`, but rather than returning the first matching element it returns the *index* of the first match Take the following example, which uses names instead of numbers for clarity.

```
const arr = ['Nick', 'Frank', 'Joe', 'Frank'];
const foundIndex = arr.findIndex(el => el === 'Frank');
```

```
console.log(foundIndex);
// 1
```

- **indexOf:** Works almost identically to `findIndex`, but instead of taking a function as an argument it takes a simple value. You can use thi have simpler logic and don't need to use a function to check whether there is a match.

```
const arr = ['Nick', 'Frank', 'Joe', 'Frank'];
const foundIndex = arr.indexOf('Frank');
```

```
console.log(foundIndex);
// 1
```

push, pop, shift, unshift

There are a lot of great array method to help add or remove elements from arrays in a targeted fashion.

- **push:** This is a relatively simple method that adds an item to the end of an array. It modifies the array in-place and the function itself retu added to the array.

```
let arr = [1, 2, 3, 4];
const pushed = arr.push(5);
```

```
console.log(arr);
// [1, 2, 3, 4, 5]
console.log(pushed);
// 5
```

- **pop:** This removes the last item from an array. Again, it modifies the array in place. The function itself returns the item removed from the

```
let arr = [1, 2, 3, 4];
const popped = arr.pop();
```

```
console.log(arr);
// [1, 2, 3]
console.log(popped);
// 4
```

- **shift:** This removes the first item from an array. Again, it modifies the array in place. The function itself returns the item removed from the

```
let arr = [1, 2, 3, 4];
const shifted = arr.shift();
```

```
console.log(arr);
// [2, 3, 4]
console.log(shifted);
// 1
```

- **unshift:** This adds one or more elements to the beginning of an array. Again, it modifies the array in place. Unlike a lot of the other metho function itself returns the new length of the array.

```
let arr = [1, 2, 3, 4];
const unshifted = arr.unshift(5, 6, 7);
```

```
console.log(arr);
// [5, 6, 7, 1, 2, 3, 4]
console.log(unshifted);
// 7
```

splice, slice

These methods either modify or return subsets of arrays.

- **splice:** Change the contents of an array by removing or replacing existing elements and/or adding new elements. This method modifies t place.

The following code sample can be read as: at position `1` of the array, remove `0` elements and insert `b`.

```
let arr = ['a', 'c', 'd', 'e'];
arr.splice(1, 0, 'b')
```

- **slice:** returns a shallow copy of an array from a specified start position and before a specified end position. If no end position is specified array is returned. Importantly, this method **does not** modify the array in place but rather returns the desired subset.

```
let arr = ['a', 'b', 'c', 'd', 'e'];
const sliced = arr.slice(2, 4);
```

```
console.log(sliced);
// ['c', 'd']
console.log(arr);
// ['a', 'b', 'c', 'd', 'e']
```

sort

- **sort:** sorts an array based on the provided function which takes a first element and second element argument. Modifies the array in place returns negative or 0, the order remains unchanged. If positive, the element order is switched.

```
let arr = [1, 7, 3, -1, 5, 7, 2];
const sorter = (firstEl, secondEl) => firstEl - secondEl;
arr.sort(sorter);
```

```
console.log(arr);
// [-1, 1, 2, 3, 5, 7, 7]
```

Phew, did you catch all of that? Neither did I. In fact, I had to reference the MDN docs a lot while writing this—and that's okay! Just knowing methods are out there with get you 95% of the way there.

7. Generators

Don't fear the `*`. The generator function specifies what `value` is yielded next time `next()` is called. Can either have a finite number of yields `next()` returns an undefined `value`, or an infinite number of values using a loop.

```
function* greeter() {
  yield 'Hi';
  yield 'How are you?';
  yield 'Bye';
}
```

```
const greet = greeter();
```

```
console.log(greet.next().value);
// 'Hi'
console.log(greet.next().value);
// 'How are you?'
console.log(greet.next().value);
// 'Bye'
console.log(greet.next().value);
// undefined
```

And using a generator for infinite values:

```
function* idCreator() {
  let i = 0;
  while (true)
    yield i++;
}
```

```
const ids = idCreator();
```

```
console.log(ids.next().value);
// 0
console.log(ids.next().value);
// 1
console.log(ids.next().value);
// 2
// etc...
```

8. Identity Operator (===) vs. Equality Operator (==)

Be sure to know the difference between the identify operator (===) and equality operator (==) in JavaScript! The ==operator will do type con comparing values whereas the === operator will not do any type conversion before comparing.

```
console.log(0 == '0');
// true
console.log(0 === '0');
// false
```

9. Object Comparison

A mistake I see JavaScript newcomers make is directly comparing objects. Variables are pointing to references to the objects in memory, nc themselves! One method to actually compare them is converting the objects to JSON strings. This has a drawback though: object property guaranteed! A safer way to compare objects is to pull in a library that specializes in deep object comparison (e.g., lodash's isEqual).

The following objects appear equal but they are in fact pointing to different references.

```
const joe1 = { name: 'Joe' };
const joe2 = { name: 'Joe' };
```

```
console.log(joe1 === joe2);
// false
```

Conversely, the following evaluates as true because one object is set equal to the other object and are therefore pointing to the same refere only one object in memory).

```
const joe1 = { name: 'Joe' };
const joe2 = joe1;
```

```
console.log(joe1 === joe2);
// true
```

Make sure to review the Value vs. Reference section above to fully understand the ramifications of setting a variable equal to another variab pointing to a reference to an object in memory!

10. Callback Functions

Far too many people are intimidated by JavaScript callback functions! They are simple, take this example. The `console.log` function is bei callback to `myFunc`. It gets executed when `setTimeout` completes. That's all there is to it!

```
function myFunc(text, callback) {
  setTimeout(function() {
    callback(text);
  }, 2000);
}
```

```
myFunc('Hello world!', console.log);
// 'Hello world!'
```

11. Promises

Once you understand JavaScript callbacks you'll soon find yourself in nested "callback hell." This is where Promises help! Wrap your async Promise and `resolve` on success or `reject` on fail. Use "then" to handle success and `catch` to handle failure.

```
const myPromise = new Promise(function(res, rej) {
  setTimeout(function(){
    if (Math.random() < 0.9) {
      return res('Hooray!');
    }
    return rej('Oh no!');
  }, 1000);
```

```
});
```

```
myPromise
  .then(function(data) {
    console.log('Success: ' + data);
   })
   .catch(function(err) {
    console.log('Error: ' + err);
   });

// If Math.random() returns less than 0.9 the following is logged:
// "Success: Hooray!"
// If Math.random() returns 0.9 or greater the following is logged:
// "Error: On no!"
```

12. Async Await

Once you get the hang of JavaScript promises, you might like `async await`, which is just “syntactic sugar” on top of promises. In the follow create an `async` function and within that we `await` the `greeter` promise.

```
const greeter = new Promise((res, rej) => {
  setTimeout(() => res('Hello world!'), 2000);
})
```

```
async function myFunc() {
  const greeting = await greeter;
  console.log(greeting);
}
```

```
myFunc();
// 'Hello world!'
```

Conclusion

If you didn’t know any of these 12 concepts, you likely have grown at least a little in your knowledge of JavaScript! And if you knew them all

this was a chance to practice and grow your knowledge. What other concepts do you think are critical? Let me know in the comments.

9

by Nick Scialli @nas5w.
Husband, dog dad, coffee monster. Software engineer at the @usds! Opinions are my own

Read my stories

TAGS

#javascript #programming #front-end-development #nodejs #software-development

Related Stories

Subject Matter
You Might Not Need that Recursive Function in JavaScript by @nas5w
#javascript
How to Draw Generative NFT Mushrooms with Three.js by @ferluht
#nft
You Cannot Learn Coding without Problem-Solving Skills by @learnly
#learn-to-code
What it's Like to Code in a Developing Country by @lisandroseia
#coding
How to Implement Heap in Data Structure by @sandeepmishratech
#data-structures
An Intro to AI Powered Product Development by @fortuitapps
#product-development

THIS ARTICLE WAS FEATURED IN...

Habr Tproger Morioh Segmentfault Ma-no Infoq Tencent Flipboard Dev Diigo