# JavaScript Trends in 2021

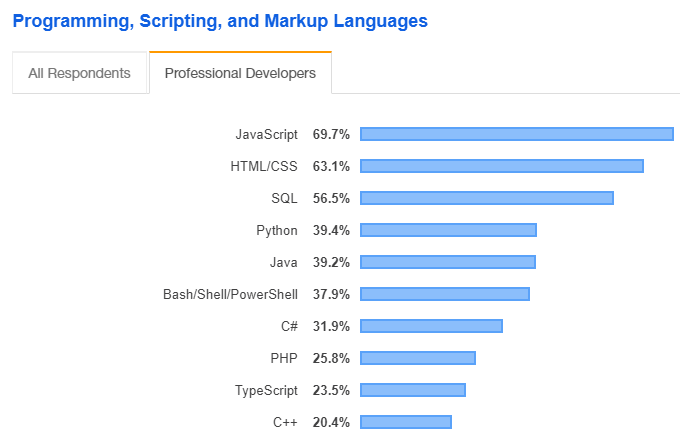
2022 is just around the corner, as unbelievable as that sounds. If you’re curious about what the future of the programming world might be, you’re in the right place. We tried to analyze trends in 2022 and want to share some insights into key directions for the JavaScript ecosystem.

Read on to find out!

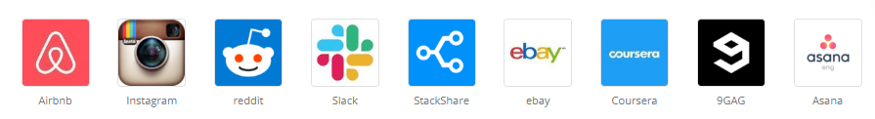
# JavaScript Language Keeps Going Strong

For years in a row, JavaScript is the most sought-after and fast-growing programming language. It remains one of the smartest choices when it comes to the development of interactive web interfaces since it’s supported by all modern browsers.

As the [annual survey held by Stack Overflow](https://insights.stackoverflow.com/survey/2019/#technology) shows, about 70 percent of 72.525 professional developers stated they use JavaScript. Moreover, it’s one of the most wanted languages meaning that 17.8% of respondents have not yet used it but want to learn it.



According to the data [provided by stackshare.io](https://www.stackshare.io/javascript), over 10400 companies worldwide use JavaScript in their stacks. The language is the heart of any big tech company, such as PayPal (likewise, the online payment giant was one of the earliest adopters of NodeJS), Netflix, Groupon, Walmart, and LinkedIn.



After all, [16 from 25 US unicorn companies](https://www.codingdojo.com/blog/unicorn-languages-report) (the top privately-held startups valued at over $1 billion) mention JavaScript in their technology stacks. It’s therefore unlikely that JavaScript goes off the grid in the near future.

JavaScript Best Practices

Avoid Global Variables

Minimize the use of global variables.

This includes all data types, objects, and functions.

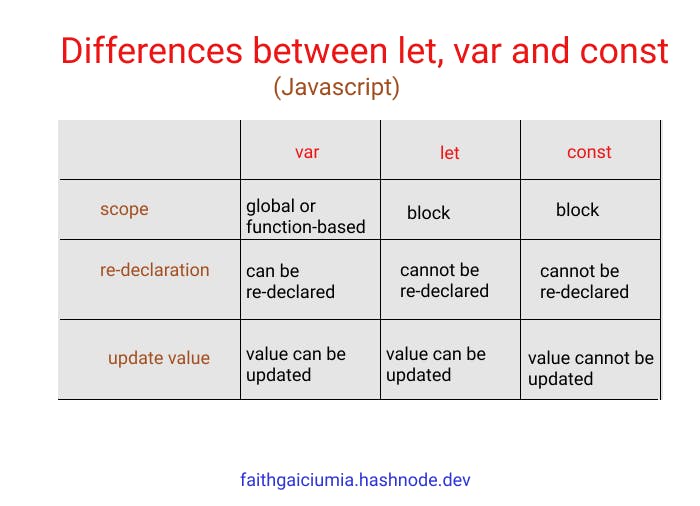
Global variables and functions can be overwritten by other scripts.

Use local variables instead

Always Declare Local Variables

All variables used in a function should be declared as **local** variables.

Local variables **must** be declared with the var keyword or the let keyword, otherwise they will become global variables.



Declarations on Top

It is a good coding practice to put all declarations at the top of each script or function.

This will:

* Give cleaner code
* Provide a single place to look for local variables
* Make it easier to avoid unwanted (implied) global variables
* Reduce the possibility of unwanted re-declarations

// Declare at the beginning  
let firstName, lastName, price, discount, fullPrice;  
  
// Use later  
firstName = "John";  
lastName = "Doe";  
  
price = 19.90;  
discount = 0.10;  
  
fullPrice = price - discount;

This also goes for loop variables:

for (let i = 0; i< 5; i++) {

Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

* Give cleaner code
* Provide a single place to initialize variables
* Avoid undefined values

// Declare and initiate at the beginning  
let firstName = "",  
let lastName = "",  
let price = 0,  
let discount = 0,  
let fullPrice = 0,  
const myArray = [],  
const myObject = {};

Initializing variables provides an idea of the intended use (and intended data type).

Declare Objects with **const**

Declaring objects with const will prevent any accidential change of type:

Example

let car = {type:"Fiat", model:"500", color:"white"};  
car = "Fiat";      // Changes object to string

const car = {type:"Fiat", model:"500", color:"white"};  
car = "Fiat";      // Not possible

Declare Arrays with **const**

Declaring arrays with const will prevent any accidential change of type:

Example

let cars = ["Saab", "Volvo", "BMW"];  
cars = 3;    // Changes array to number

const cars = ["Saab", "Volvo", "BMW"];  
cars = 3;    // Not possible

Don't Use new Object()

* Use "" instead of new String()
* Use 0 instead of new Number()
* Use false instead of new Boolean()
* Use {} instead of new Object()
* Use [] instead of new Array()
* Use /()/ instead of new RegExp()
* Use function (){} instead of new Function()

Example

let x1 = "";             // new primitive string  
let x2 = 0;              // new primitive number  
let x3 = false;          // new primitive boolean  
const x4 = {};           // new object  
const x5 = [];           // new array object  
const x6 = /()/;         // new regexp object  
const x7 = function(){}; // new function object

Beware of Automatic Type Conversions

JavaScript is loosely typed.

A variable can contain all data types.

A variable can change its data type:

Example

let x = "Hello";     // typeof x is a string  
x = 5;               // changes typeof x to a number

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_best_typeof)

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

When doing mathematical operations, JavaScript can convert numbers to strings:

Example

let x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number  
let x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string  
let x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string  
let x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number  
let x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number  
let x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number  
let x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_best_valueof)

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

Example

"Hello" - "Dolly"    // returns NaN

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_best_string_op1)

Use === Comparison

The == comparison operator always converts (to matching types) before comparison.

The === operator forces comparison of values and type:

Example

0 == "";        // true  
1 == "1";       // true  
1 == true;      // true  
  
0 === "";       // false  
1 === "1";      // false  
1 === true;     // false

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_best_comparison)

Use Parameter Defaults

If a function is called with a missing argument, the value of the missing argument is set to undefined.

Undefined values can break your code. It is a good habit to assign default values to arguments.

Example

function myFunction(x, y) {  
  if (y === undefined) {  
    y = 0;  
  }  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_best_parameter_default)

End Your Switches with Defaults

Always end your switch statements with a default. Even if you think there is no need for it.

Example

switch (new Date().getDay()) {  
  case 0:  
    day = "Sunday";  
    break;  
  case 1:  
    day = "Monday";  
    break;  
  case 2:  
    day = "Tuesday";  
    break;  
  case 3:  
    day = "Wednesday";  
    break;  
  case 4:  
    day = "Thursday";  
    break;  
  case 5:  
    day = "Friday";  
    break;  
  case 6:  
    day = "Saturday";  
    break;  
  default:  
    day = "Unknown";  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_break_switch)

Avoid Number, String, and Boolean as Objects

Always treat numbers, strings, or booleans as primitive values. Not as objects.

Declaring these types as objects, slows down execution speed, and produces nasty side effects:

Example

let x = "John";               
let y = new String("John");  
(x === y) // is false because x is a string and y is an object.

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_best_object_string1)

Or even worse:

Example

let x = new String("John");               
let y = new String("John");  
(x == y) // is false because you cannot compare objects.

# JavaScript Common Mistakes(Quality code)

## Accidentally Using the Assignment Operator

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

This if statement returns false (as expected) because x is not equal to 10:

let x = 0;  
if (x == 10)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_equal_1)

This if statement returns true (maybe not as expected), because 10 is true:

let x = 0;  
if (x = 10)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_equal_2)

This if statement returns false (maybe not as expected), because 0 is false:

let x = 0;  
if (x = 0)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_equal_3)

An assignment always returns the value of the assignment.

## Expecting Loose Comparison

In regular comparison, data type does not matter. This if statement returns true:

let x = 10;  
let y = "10";  
if (x == y)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_1)

In strict comparison, data type does matter. This if statement returns false:

let x = 10;  
let y = "10";  
if (x === y)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_2)

It is a common mistake to forget that switch statements use strict comparison:

This case switch will display an alert:

let x = 10;  
switch(x) {  
  case 10: alert("Hello");  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_3)

This case switch will not display an alert:

let x = 10;  
switch(x) {  
  case "10": alert("Hello");  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_4)

## Confusing Addition & Concatenation

**Addition** is about adding **numbers**.

**Concatenation** is about adding **strings**.

In JavaScript both operations use the same + operator.

Because of this, adding a number as a number will produce a different result from adding a number as a string:

let x = 10;  
x = 10 + 5;       // Now x is 15  
  
let y = 10;  
y += "5";        // Now y is "105"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_add_1)

When adding two variables, it can be difficult to anticipate the result:

let x = 10;  
let y = 5;  
let z = x + y;     // Now z is 15  
  
let x = 10;  
let y = "5";  
let z = x + y;     // Now z is "105"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_add_2)

## Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits **Floating point numbers** (Floats).

All programming languages, including JavaScript, have difficulties with precise floating point values:

let x = 0.1;  
let y = 0.2;  
let z = x + y            // the result in z will not be 0.3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_floats)

To solve the problem above, it helps to multiply and divide:

### Example

let z = (x \* 10 + y \* 10) / 10;       // z will be 0.3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_floats_ok)

## Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

### Example 1

let x =  
"Hello World!";

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_string_1)

But, breaking a statement in the middle of a string will not work:

### Example 2

let x = "Hello  
World!";

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_string_2)

You must use a "backslash" if you must break a statement in a string:

### Example 3

let x = "Hello \  
World!";

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_string_3)

## Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

if (x == 19);  
{  
  // code block   
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_semicolon)

## Breaking a Return Statement

It is a default JavaScript behavior to close a statement automatically at the end of a line.

Because of this, these two examples will return the same result:

### Example 1

function myFunction(a) {  
  let power = 10   
  return a \* power  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_1)

### Example 2

function myFunction(a) {  
  let power = 10;  
  return a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_2)

JavaScript will also allow you to break a statement into two lines.

Because of this, example 3 will also return the same result:

### Example 3

function myFunction(a) {  
  let  
  power = 10;   
  return a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_3)

But, what will happen if you break the return statement in two lines like this:

### Example 4

function myFunction(a) {  
  let  
  power = 10;   
  return  
  a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_4)

The function will return undefined!

Why? Because JavaScript thought you meant:

### Example 5

function myFunction(a) {  
  let  
  power = 10;   
  return;  
  a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_5)

## Explanation

If a statement is incomplete like:

let

JavaScript will try to complete the statement by reading the next line:

power = 10;

But since this statement is complete:

return

JavaScript will automatically close it like this:

return;

This happens because closing (ending) statements with semicolon is optional in JavaScript.

JavaScript will close the return statement at the end of the line, because it is a complete statement.

Never break a return statement.

## Accessing Arrays with Named Indexes

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** use **numbered indexes**:

### Example

const person = [];  
person[0] = "John";  
person[1] = "Doe";  
person[2] = 46;  
person.length;       // person.length will return 3  
person[0];           // person[0] will return "John"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_array_associative_1)

In JavaScript, **objects** use **named indexes**.

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.

After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

### Example:

const person = [];  
person["firstName"] = "John";  
person["lastName"] = "Doe";  
person["age"] = 46;  
person.length;      // person.length will return 0  
person[0];          // person[0] will return undefined

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_array_associative_2)

## Ending Definitions with a Comma

Trailing commas in object and array definition are legal in ECMAScript 5.

### Object Example:

person = {firstName:"John", lastName:"Doe", age:46,}

### Array Example:

points = [40, 100, 1, 5, 25, 10,];

## Undefined is Not Null

JavaScript objects, variables, properties, and methods can be undefined.

In addition, empty JavaScript objects can have the value null.

This can make it a little bit difficult to test if an object is empty.

You can test if an object exists by testing if the type is undefined:

### Example:

if (typeof myObj === "undefined")

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_object_undefined)

But you cannot test if an object is null, because this will throw an error if the object is undefined:

### Incorrect:

if (myObj === null)

To solve this problem, you must test if an object is not null, and not undefined.

But this can still throw an error:

### Incorrect:

if (myObj !== null && typeof myObj !== "undefined")

Because of this, you must test for not undefined before you can test for not null:

### Correct:

if (typeof myObj !== "undefined" &&myObj !== null)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_object_null)