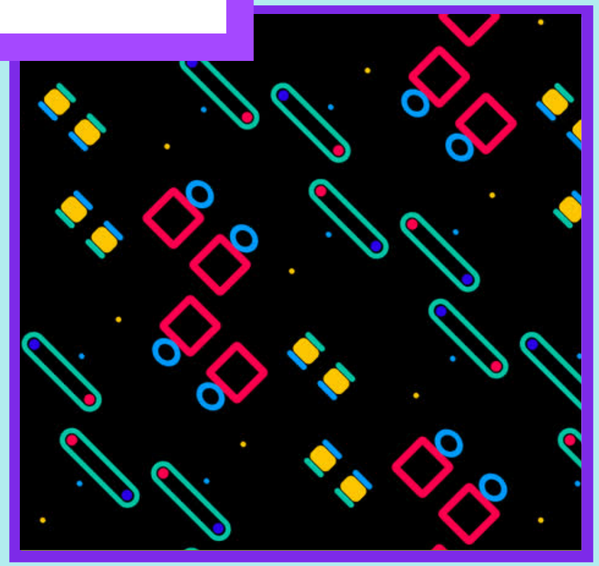
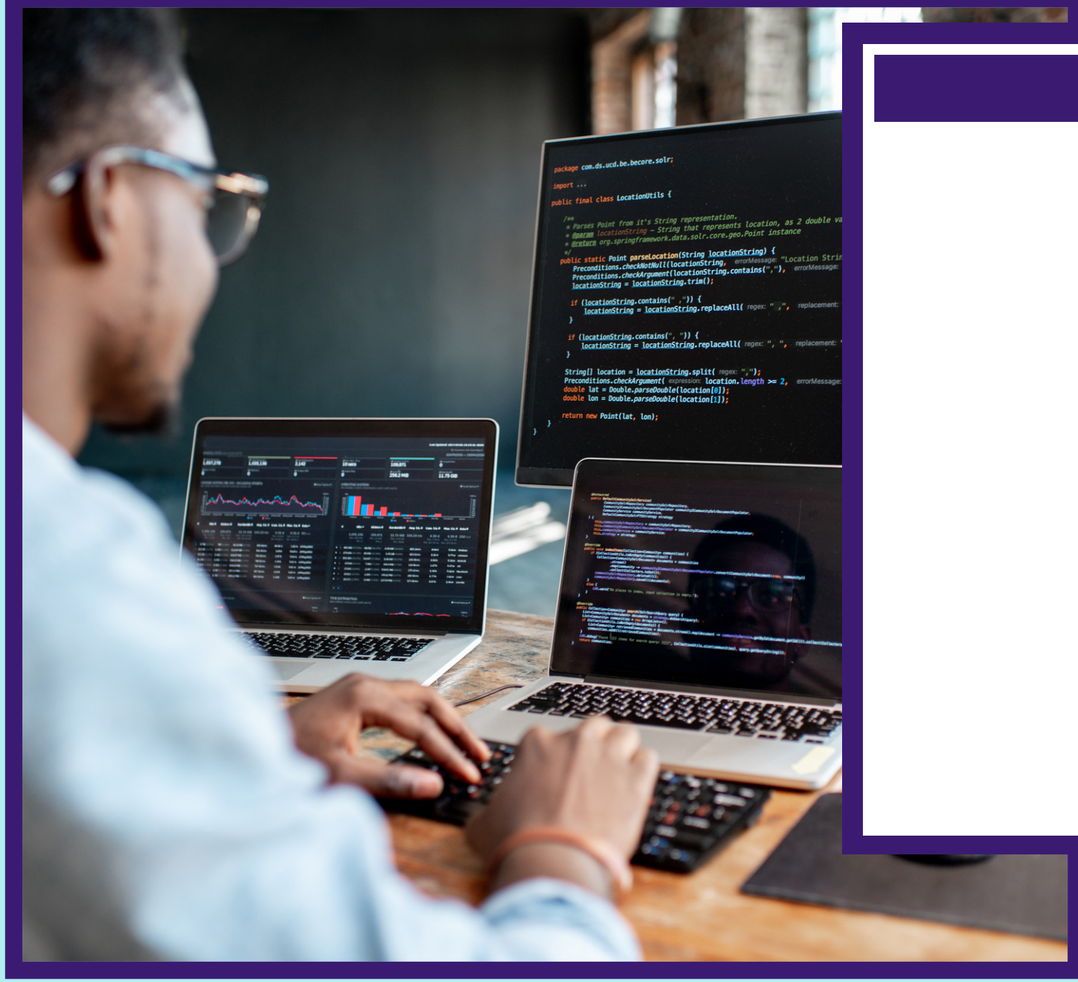
Javascript Syntax

Prepared by
Denis Tokarev

Canva





Basic JS syntax

The syntax, expressions, and operators

VARIABLES & CONSTANTS

You can define them using one of the keywords →

```
01 | // An immutable value available from the line
02 | // where the constant is defined
03 | const constValue = 'hey';
04 |
05 | // A mutable value available from the line
06 | // where the variable is defined
07 | let variableValue = 0;
08 |
09 | // A mutable value, old syntax. Hoists to the beginning
10 | // of the function scope or to the global scope,
11 | // whatever is closer. Not recommended for use.
12 | var anotherVariableValue = false;
13 |
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

PRINTING ARBITRARY VALUES TO CONSOLE

Just use **console.log** →

```
01 | // Let's say we have a constant storing a string
02 | const name = 'Bindi';
03 |
04 | // And a variable storing a number
05 | let age = 22;
06 |
07 | // And an object
08 | let wildlifeWarrior = {
09 |   name: name,
10 |   age: age,
11 |   gender: 'f',
12 | };
13 |
14 | // This will print 'Hello, world!' to console
15 | console.log('Hello, world!');
16 |
17 | // Additionally, you can print any value
18 | console.log(name); // Prints 'Bindi'
19 | console.log('name:', name); // Prints 'name: Bindi'
20 | console.log('age: ' + age); // Prints 'age: 22'
21 |
22 | // This will print an object as a string
23 | console.log('Person:', wildlifeWarrior);
24 | // Person: { name: 'Bindi', age: 22, gender: 'f' }
25 |
```

DATA TYPES

JavaScript is a dynamically typed language, meaning that the same variable can contain values of different types →

```
01 | let value;
02 |
03 | // A float or integer number
04 | value = 0.1;
05 |
06 | // A string, double quotes "" can also be used
07 | value = 'some-string';
08 |
09 | // In JavaScript, null is a separate type
10 | value = null;
11 |
12 | // This type represents no value and no variable
13 | value = undefined;
14 |
15 | // A boolean value, either true or false
16 | value = true;
17 |
18 | // A unique value
19 | value = Symbol();
20 |
21 | // An object, which is basically a dictionary
22 | // of dynamically typed values identified by their keys
23 | value = { key: 'hey', anotherKey: 10 };
24 |
25 |
```

DATA TYPES

Functions are first-class citizens, meaning that they can be assigned to variables and can be referenced →

```
let value;

// An anonymous function
value = (param1, param2) => {
  console.log(param1, param2);
};

// A normal function
value = function(param) {
  console.log('param =', param);
};

// Function can be defined without assigning it
// to a variable, in that case, it hoists
function foo(bar) {
  // If number, adds 1 to it, otherwise concatenates it
  return bar + 1;
}

// Calling a function
const result = foo(10);
value(result); // Prints 'param = 11'

// A big integer number
value = BigInt(9007199254740991);
```

DATA TYPES

You can always find out the type of the value that the varable currently holds, at runtime →

However, in some cases, you'd need a more complex check

```
01 | let value;
02 |
03 | value = 0.1;
04 | console.log(typeof value); // Prints 'number'
05 |
06 | value = 'some-string';
07 | console.log(typeof value); // Prints 'string'
08 |
09 | value = null;
10 | console.log(typeof value); // Prints 'object'
11 |
12 | value = undefined;
13 | console.log(typeof value); // Prints 'undefined'
14 |
15 | value = true;
16 | console.log(typeof value); // Prints 'boolean'
17 |
18 | value = Symbol();
19 | console.log(typeof value); // Prints 'symbol'
20 |
21 |
22 |
23 |
24 |
25 |
```

DATA TYPES

You can always find out the type of the value that the varable currently holds, at runtime →

However, in some cases, you'd need a more complex check

```
01 | let value;
02 |
03 | value = { key: 'hey', anotherKey: 10 };
04 | console.log(typeof value); // Prints 'object'
05 |
06 | value = function() { /* ... */ };
07 | console.log(typeof value); // Prints 'function'
08 |
09 | value = BigInt(9007199254740991);
10 | console.log(typeof value); // Prints 'bigint'
11 |
12 |
13 |
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

DATA TYPES

undefined is also the type of a variable that hasn't been assigned a value yet →

#justjavascriptythings

```
01 | let value;
02 |
03 | console.log(typeof value); // Prints 'undefined'
04 |
05 | // For more information on data types,
06 | // visit the MDN page:
07 | // http://mdn.io/Data_structures
08 |
09 | // I also recommend watching this old but gold talk
10 | // by Gary Bernhardt:
11 | // https://www.destroyallsoftware.com/talks/wat
12 | // This talk makes fun of some JS things like
13 | undefined == null
14 | // being true, and
15 | undefined === null
16 | // being false
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

TYPE CONVERSIONS

It's possible to convert values of one type into another, for some of the supported types →

```
01 | let value = 10;
02 |
03 | let strValue = value.toString();
04 | // or: let strValue = value + '';
05 |
06 | let numValue = parseInt(strValue, 10);
07 | // or: let numValue = parseFloat(strValue);
08 | // or: let numValue = +strValue;
09 |
10 | let bigIntValue = BigInt(numValue);
11 | // or: let bigIntValue = BigInt(strValue);
12 |
13 | let numValue = Number(bigIntValue);
14 | // or: let numValue = parseInt(bigIntValue, 10);
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

COMPARING VALUES

There is a strict equality check and a loose equality check, and the strict check is generally preferred whenever possible →

```
let value = 10;
let anotherValue = '10';

// With a loose equality check (aka '=='), two variables
// are considered as holding identical values
console.log(value == anotherValue); // Prints 'true'

// With a strict equality check (aka '==='), the result
// will indicate that the stored values are different
console.log(value === anotherValue); // Prints 'false'

// Comparing with an implicit type case can result
// in a pretty weird outcome and should be avoided
// when possible
const str = '[object Object]';
const obj = { catsSay: 'meow', dogsSay: 'woof' };
console.log(str == obj); // Prints 'true' ¯\_(ツ)_/¯
console.log(str === obj); // Prints 'false'

// For more information, please check out these pages:
// https://mdn.io/Equality_comparisons_and_sameness
// http://ecma-international.org/ecma-262/5.1/#sec-11.9.3


```

PRIMITIES

Object, Array, Map, Set

```
01 | // An object is a dictionary of dynamically typed values
02 | const obj = { key: 'some-value', anotherKey: 10.2 };
03 | console.log(obj.key); // Prints 'some-value'
04 | console.log(obj['key']); // Prints 'some-value'
05 |
06 | // An array is an indexed list of dynamically typed
07 | // values. Every Array is also an object.
08 | const arr = ['first value', 2, obj];
09 | console.log(arr[0]); // Prints 'first value'
10 | console.log(arr.length); // Prints '3'
11 | console.log(typeof arr); // Prints 'object'
12 |
13 | // A Map is an object that can use any value as a key
14 | // (not only strings) and preserves the original
15 | // element order.
16 | const map = new Map();
17 | map.set('one', 1);
18 | console.log(map.get('one')); // Prints '1'
19 |
20 | // A Set is an object that contains unique values
21 | const set = new Set();
22 | set.add('one');
23 | set.add('one'); // 'one' isn't added the second time
24 | console.log(set.has('one')); // Prints 'true'
25 |
```

CONDITIONALS

Conditionals look like their alternatives in other C-like languages

The value in parentheses automatically gets casted to a boolean value

```
01 | // One-liner if condition
02 | if (condition) doSomething(); // If condition is truthy
03 |
04 | // Multiline if
05 | if (condition) {
06 |   // If condition is truthy
07 | }
08 |
09 | // If with else
10 | if (confition) {
11 |   // If condition is truthy
12 | } else {
13 |   // If condition is falsy
14 | }
15 |
16 | // If with multiple branches
17 | if (condition1) {
18 |   // If condition1 is truthy
19 | } else if (condition2) {
20 |   // If condition2 is truthy
21 | } else {
22 |   // If both condition1 and condition2 are falsy
23 | }
24 |
25 |
```

CONDITIONALS

Using the **switch** statement often is more convenient and improves the code readability →

Don't forger about the **break** at the end of each case!

```
01 | const value = 5;
02 |
03 | // For mulitple branches, it's often more
04 | // convenient to use a switch-case block
05 | switch (value) {
06 |   case 0:
07 |     // If value equals to 0, the below code runs
08 |     console.log('No items in the bag!')
09 |     break;
10 |   case 1:
11 |     // If value equals to 1, the below code runs
12 |     console.log('One item is in the bag!');
13 |     break;
14 |   case 2:
15 |     // If value equals to 2, the below code runs
16 |     console.log('A couple of items is in the bag!');
17 |     break;
18 |   default:
19 |     // If value isn't 0, 1, or 2, the below code runs
20 |     console.log('Lots of items are in the bag');
21 | }
22 |
23 |
24 |
25 |
```

CONDITIONALS

Inline if, aka the Ternary operator, is a convenient way to write less code and make it easier to understand →

```
01 | // Inline if (aka the ternary operator)
02 | condition ? ifTrue() : ifFalse();
03 |
04 | // It is an operator, meaning that it returns a value
05 | const result = condition ? valIfTrue() : valIfFalse();
06 |
07 | // Additionally, you can use lazy || and && operators
08 | condition && ifTrue() || ifFalse();
09 |
10 | // The above also returns a value
11 | // (this is also known as rvalue in some languages)
12 | const result = condition && valIfTrue() || valIfFalse();
13 |
14 | // Thanks to the && operator, it's possible to write
15 | // a short version of a one-liner if:
16 | condition && runSomething();
17 | // which is a single line equivalent to
18 | if (condition) {
19 |   runSomething();
20 | }
21 |
22 |
23 |
24 |
25 |
```

CONDITIONALS

As said above, the condition inside parentheses gets auto-casted to a boolean value

However, to avoid accidental unexpected bugs, still, you must understand how the type casting works →

```
01 | if (true) {
02 |   console.log('Always prints');
03 | }
04 |
05 | if (false) {
06 |   console.log('Never prints');
07 | }
08 |
09 | if ('hey') {
10 |   console.log('Always prints');
11 | }
12 |
13 | if ('') {
14 |   console.log('Never prints');
15 | }
16 |
17 | if ('0') {
18 |   console.log('Always prints');
19 | }
20 |
21 | if (0) {
22 |   console.log('Never prints');
23 | }
24 |
25 |
```

CONDITIONALS

As said above, the condition inside parentheses gets auto-casted to a boolean value

However, to avoid accidental unexpected bugs, still, you must understand how the type casting works →

```
01 | if (1) {
02 |   console.log('Always prints');
03 | }
04 |
05 | if (null) {
06 |   console.log('Never prints');
07 | }
08 |
09 | if (undefined) {
10 |   console.log('Never prints');
11 | }
12 |
13 | if ({ x: 'test' }) {
14 |   console.log('Always prints');
15 | }
16 |
17 | if (Symbol()) {
18 |   console.log('Always prints');
19 | }
20 |
21 | if (function x() {}) {
22 |   console.log('Always prints');
23 | }
24 |
25 |
```

CONDITIONALS

And sometimes, it gets really tricky →

```
01 | if (BigInt(1020202)) {
02 |   console.log('Always prints');
03 | }
04 |
05 | if (BigInt(0)) {
06 |   console.log('Never prints');
07 | }
08 |
09 | if ('1' - 1) {
10 |   console.log('Never prints');
11 | }
12 |
13 | if (+'0') {
14 |   console.log('Never prints');
15 | }
16 |
17 | if (new String('')) {
18 |   console.log('Always prints');
19 | }
20 |
21 | if (new Number(0)) {
22 |   console.log('Always prints');
23 | }
24 |
25 |
```

LOOPS

JavaScript supports the same syntax for the loops that many curly bracket syntax languages like C do

```
01 | // Good old for loop
02 | for (let i = 0; i < 100; i++) {
03 |   console.log(i);
04 | }
05 |
06 | // An equivalent while loop
07 | let i = 0;
08 | while (i < 100) {
09 |   console.log(i);
10 |   i++;
11 | }
12 |
13 | // Or equivalent do...while loop
14 | let i = 0;
15 | do {
16 |   ++i;
17 |   if (i >= 100) {
18 |     break;
19 |   }
20 |
21 |   console.log(i);
22 | } while (true);
23 |
24 |
25 |
```

CLASSES

Modern JavaScript also supports classes, allowing for a convenient way to implement abstractions and ship the data along with the functions that can process it →

```
01 | // Defining a new class
02 | class Snack {
03 |   // A consuctor that will be auto-called
04 |   constructor(calories, name) {
05 |     this.caloriesRemaining = calories;
06 |     this.name = name;
07 |   }
08 |
09 |   // One of attached functions, usually called 'methods'
10 |   chew() {
11 |     this.caloriesRemaining -= 100;
12 |   }
13 | }
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

CLASSES

A JavaScript class can be extended and can have static members

```
01 | class Snack {
02 |   constructor(calories, name) {
03 |     this.caloriesRemaining = calories;
04 |     this.name = name;
05 |   }
06 |   chew() {
07 |     this.caloriesRemaining -= 100;
08 |   }
09 | }
10 |
11 | class Pizza extends Snack {
12 |   static caloriesPerGram = 2.66;
13 |
14 |   constructor(weightInGrams) {
15 |     super(Pizza.caloriesPerGram * weightInGrams, 'XL');
16 |   }
17 | }
18 |
19 | class Crisps extends Snack {
20 |   static caloriesPerGram = 5.36;
21 |
22 |   constructor(weightInGrams) {
23 |     super(Crisps.caloriesPerGram * weightInGrams, 'Thins');
24 |   }
25 | }
```

CLASSES

With classes, you can use all features and patterns established in object-oriented programming

However, internally, each class is just a constructor function, because JavaScript OOP is based on 'prototyping'

```
01 | class Meal {
02 |   constructor(snacks) {
03 |     this.snacks = snacks;
04 |   }
05 |
06 |   eatSome() {
07 |     for (let i = 0; i < this.snacks.length; i++) {
08 |       const snack = this.snacks[i];
09 |       console.log('Chewing', snack.name);
10 |       snack.chew();
11 |     }
12 |   }
13 |
14 |    logCalories() {
15 |     for (let i = 0; i < this.snacks.length; i++) {
16 |       const snack = this.snacks[i];
17 |       console.log(snack.name, ': ', snack.caloriesRemaining);
18 |     }
19 |    }
20 | }
21 |
22 | console.log(typeof Meal); // Prints 'function'
23 |
24 |
25 |
```

CLASSES

A class is just a constructor function, and a class instance is an object

```
const snacks = [
  new Pizza(800),
  new Crisps(150),
];

const meal = new Meal(snacks);
meal.logCalories();
meal.eatSome();
meal.logCalories();

console.log(typeof Meal); // Prints 'function'
console.log(typeof meal); // Prints 'object'
```

SIMPLE FIBONACCI SEQUENCE GENERATOR IN JAVASCRIPT

So hopefully, this example's got a bit easier to understand →

```
01 | const fib = (n) => {
02 |   if (n === 0) {
03 |     return 0;
04 |   }
05 |
06 |   if (n === 1) {
07 |     return 1;
08 |   }
09 |
10 |   let current = 0;
11 |   let previous = 1;
12 |   let prePrevious = 0;
13 |   // Iterative calculation
14 |   for (let i = 0; i < n; i++) {
15 |     prePrevious = previous;
16 |     previous = current;
17 |     current = prePrevious + previous;
18 |   }
19 |
20 |   return current;
21 | };
22 |
23 | console.log(fib(5));
24 |
25 |
```