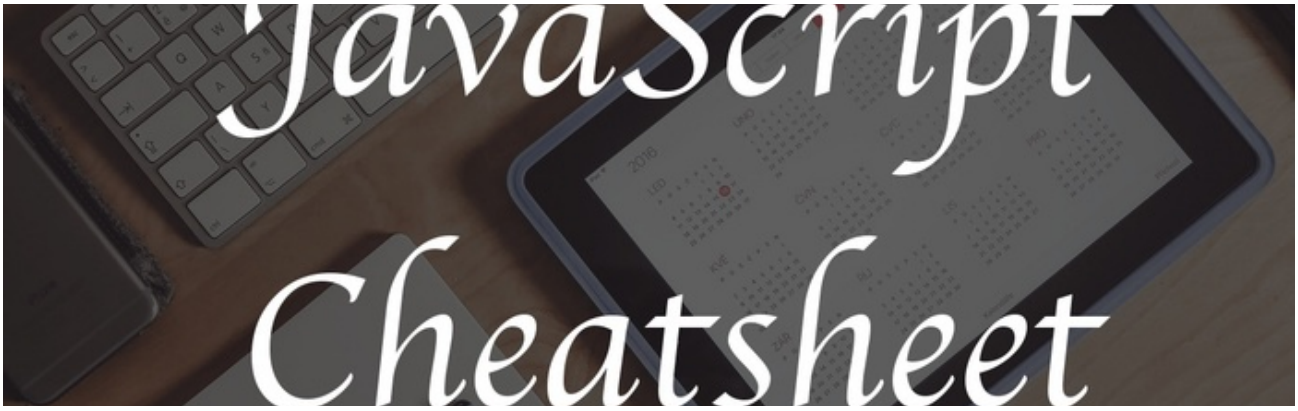
Johnny B. (Ionică Bizău) FOLLOW
Full Stack, *nix user, Pianist, College Dropout, Vegetarian, Jesus follower

# The Ultimate JavaScript Cheat Sheet

Published Nov 08, 2016




This is a quick overview of the JavaScript language. Reading this from the beginning to end is good, but going to a specific section is good as well.

These days JavaScript runs on browsers, servers, powers command line tools, and more. In this cheat sheet, we will include a couple of browser features you can access with JavaScript on the client side and platform-agnostic features you can use either on the client or server.

*(READ MORE: JavaScript Best Practices: Tips & Tricks to Level Up Your Code)*

## Quick Example

When it runs in the browser, you can set up some GUI (graphical interface) elements such as text inputs, buttons, and lots more.

Let's create a small app which checks if a number is odd or even. Save the following piece of code into a file named `odd-or-even.html` :

```
<!DOCTYPE html>
<html>
    <head>
        <
```



```
        <input type="number" id="inputNumber"><button id="runBtn">Odd or Ev
        <p class="result">Enter a number and click the button to find out i
        <script type="text/javascript">
            // Use a wrapping function to prevent global variables
            (function () {
                // Enable the strict mode (in short, it tells us some poter
                "use strict";

                // You can select elements using document.querySelector
                var $result = document.querySelector(".result");

                // Another way is using getElementById(<id>)
                var $input = document.getElementById("inputNumber");
                var $btn = document.getElementById("runBtn");

                // Append a click handler on the button
                $btn.addEventListener("click", function (event) {

                    // Validate the input
                    if (!$input.value) {
                        return alert("Please provide a number.");
                    }

                    // When clicking on the button, take the value from inp
                    // Also convert it into a number
                    var x = +$input.value;

                    // Create t a new variable that will contain the result
                    // ("odd" or "event")
                    var result;

                    // Check if the number is even
                    if (x % 2 === 0) {
                      result = "even";
                    } else {
                      result = "odd";
                    }

                    // Tell the user if the number is odd or even
                    $result.textContent = x.toString() + " is " + result;
                });
            })();
        </script>
    </bod
</html>
```



Now open this in your browser and play with it: write a number in the input, click the button and enjoy.

## Notes

- You don't have to compile JavaScript. You can interpret it using a browser or platforms like Node.js.
- Semicolons are optional. Really. It's up to you if you decide using them or not. In this blog post you'll find why it's (not) good to use semicolons everywhere. In short, you decide if you like to write the code with semicolons or not.
- Some examples contain code written in ES2015 (ES6) may not work on older browsers/interpreters. In such cases, simply paste the code in this online ES6 transpiler and you will get the ES5 equivalent.
- This document may be updated and improved from time to time.

### Comments

Single line comments start with `//`. For multi-line commands, you use `/* ... */`

```
// This is a single line comment

/*
 And this is
 a multi-line
 comment
 */
```

### Variables

You can create variables using `var`, `let`, `const` (the last two ones are available only in ES6).

Variable names can't start with a number or contain spaces. They can contain letters, numbers, underscores, or $. Variable names are case sensitive.



```
// This is how you define a variable
// `x` will be `undefined`
var x;

// Declare a constant (the convention is to use CAPS for constants)
const FOO = 42;

// Declare another two variables, using `var` and `let`
var hello = 'world';
let bar = 'baz';
```

Variables don't have a type. As long as they are not constants, you can change their data:

```
let foo = 42;
foo = 'bar';
foo
→ 'bar'
```

### `let` vs `var`

`let` and `var` are pretty similar, but unlike `var`, `let` declares a block scope local variable, optionally initializing it to a value.

`let` allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. The `var` keyword which defines a variable to an entire function regardless of block scope.

Let's look at an example:



```
var a = 42;
if (true) {
  var b = 7;
  let c = 123;
}

// Obviously, we can access the `a` variable
console.log(a);
// => 42

// We can access `b` as well, even it was assigned in the
// if block (see below what happens if the expressions is
// false).
console.log(b);
// => 7

// We cannot access `c` here because it was declared using
// `let`, inside of the `if` block, therefore it's only
// accessible in the `if` block.
console.log(c);
// => Error: c is not defined
```

Even the if the `if` condition is false, we can still access the `var` declaration inside of the `if` block. In this case, the value will be `undefined` .

```
if (false) {
  var b = 7;
}

console.log(b);
// => undefined
```

This happens because of the way variables are declared and assigned. The above code is similar to:



```
// Create the b variable (will take `undefined` by default)
var b;

if (false) {
  // Eventually assign it
  b = 7;
}

// Since we do *not* assign it, the value remains `undefined`
```

A really nice use-case to use this difference between `let` and `var` is the `for` loops with something async in it.

Notice how in the first example, we get the final value of `i` (because we print it after one second). In the other examples (either we manually create a new scope using a function, or use `let` —which does that for us), the `i` value *inside* of the `setTimeout` (after one second) is the *current* value of *i* when the `setTimeout` was called.

```
for (var i = 0; i < 7; ++i) {
  setTimeout(function () {
    console.log(i);
    // => 8
    // => 8
    // => 8
    // => 8
    // => 8
    // => 8
    // => 8
    // => 8
  }, 1000);
}

for (let i = 0; i < 7; ++i) {
  setTimeout(function () {
    console.log(i);
    // => 0
    // => 1
    // => 2
    // =>
```



```
    // => 7
  }, 1000);
}

// The above for loop, converted in ES5 looks like this:
var _loop = function (i) {
  setTimeout(function () {
    console.log(i);
    // => 0
    // => 1
    // => 2
    // => 3
    // => 4
    // => 5
    // => 6
    // => 7
  }, 1000);
};

for (var i = 0; i < 7; ++i) {
  _loop(i);
}
```

### How does `const` work?

`const` is simple: you can use it for variables which values remain the same.

```
const PI = 3.14;

PI = 3;
→ TypeError: Assignment to constant variable.
```

Note, this won't freeze the objects:



```
// Create an object
const myObj = { foo: 'bar' };

// Update the `foo` field
myObj.foo = 42;

// Add a new field
myObj.hello = 'world';

myObj
→ { foo: 42, hello: 'world' }

// Tho, if you try to reset the whole thing, you can't
myObj = {};
→ TypeError: Assignment to constant variable.
```

### `if` statemenets

Use `if (expression) { ... } else { ... }` to do something if `expression` is true or not.

```
let foo = 42;

if (foo > 40) {
    // do something
} else {
    // do something else
}
```

### Switches



```
let planet = 'Earth';

switch (planet) {
  case "Mercury":
  case "Venus":
    console.log("Too hot here.");

  case 'Earth' :
    console.log("Welcome home!");
    break;

  case 'Mars' :
    console.log("Welcome to the other home!");
    break;

  case "Jupiter":
  case "Saturn":
  case "Uranus":
  case "Neptune":
  case "Pluto":
    console.log("You may get gold here.");
    break;

  default:
    console.log("Seems you found another planet.");
    break;
}
```

## Primitive values

The following ones, are primitives:

- Booleans: `false` , `true`
- Numbers: `42` , `3.14` , `0b11010` , `0x16` , `NaN` (check out *The magic of numbers*)
- Strings: `'Earth'` , `"Mars"`
- Special values: `undefined` , `null`

## Objects

The comm



```
let myObj = { world: "Earth" };
```

**Attention**: Objects are compared by reference. That being said, we have this:

```
let firstObj = {};
let secondObj = {};

// Check if they are equal
firstObj === secondObj
→ false

// Comparing an object with itself...
firstObj === firstObj
→ true

// Let's point the secondObj to firstObj
secondObj = firstObj

// Now they are equal
firstObj === secondObj
→ true
```

Attention: you have no guarantee that adding the object keys in a specific order will get them in the same order. Object keys are *not* ordered, even in general JavaScript interpreters which will iterate them in the order of adding them in the object (again, do *not* rely on this feature).

## Arrays

In addition to objects, the array data is ordered by indexes. Arrays are actually objects, having the indexes (numbers from `0` to `legnth - 1` ) as *keys*.

```
let fruits = ["apples", "pears", "oranges"];

fruits[1]
→ "pears"
```



## Access, add, remove, and update elements

```
> let fruits = ["Apple"]

// Add to the end of the array
> fruits.push("Pear")
2 // < The new length of the array

 [ 'Apple', 'Pear' ]
 //         ^ This was just added

// Add to the start of the array
> fruits.unshift("Orange")
3 // < The new length of the array

 [ 'Orange', 'Apple', 'Pear' ]
 // ^ This was just added

// Access the element at index 1 (the second element)
> fruits[1]
'Apple'

// How many items do we have?
> fruits.length
3

// Turn the Apple into Lemon
> fruits[1] = "Lemon"
'Lemon'

 [ 'Orange', 'Lemon', 'Pear' ]
 //          ^ This was before an Apple

// Insert at index 1 a new element
> fruits.splice(1, 0, "Grapes")
[] // < Splice is supposed to delete things (see below)
   //   In this case we're deleting 0 elements and
   //   inserting one.

 [ 'Orange', 'Grapes', 'Lemon', 'Pear' ]
 //          ^ This was added.

// Get the Lemon's index
> fruits
```



```
> fruits.splice(2, 1)
[ 'Lemon' ]

 [ 'Orange', 'Grapes', 'Pear' ]
 //                   ^ Lemon is gone

// Remove the last element
> fruits.pop()
'Pear'

 [ 'Orange', 'Grapes' ]
 //                  ^ Pear is gone

// Remove the first element
> fruits.shift()
'Orange'

   [ 'Grapes' ]
 // ^ Orange is gone
```

## Iterating over Arrays

There are few ways to loop trough an array.



```
// Using for-loop
var arr = [42, 7, -1]
for (var index = 0; index < arr.length; ++index) {
    var current = arr[index];
    /* Do something with `current` */
}

// Others prefer defining an additional `length` variable:
for (var index = 0, length = arr.length; index < length; ++index) {
    var current = arr[index];
    /* ... */
}

// Another way i using `forEach`:
arr.forEach((current, index, inputArray) => {
    /* Do something with `current` */
});

// Or using the for ... of:
for (let current of arr) {
    /* current will be the current element */
}
```

## Functions

There are a couple of ways to define functions in JavaScript. The common uses are:

```
function sum (a, b) {
    return a + b;
}

var sum = function (a, b) {
    return a + b;
}

// Using the ES6 arrow functions
let sum = (a, b) => a + b;
```



```
let result = sum(40, 2);
// => 42
```

## Constructors and Classes

There are a couple of ways you can obtain a class-like functionality in JavaScript.

### Factory functions

Creating an object and returning it.

```
function Person (name) {
   var self = {};

   // Public field
   self.name = name;

   // Public method
   self.getName = function () {
     // `this` is the `self`
     return this.name;
   };

   return self;
}

var p = Person("Alice");
console.log(p.getName());
// => "Alice"
```

### Using `prototype`s

By adding a method in the `prototype` object of a function, you're adding that method as a public method to all the instances of that class.



```
function Person (name) {
  this.name = name;
}

Person.prototype.getName = function () {
  // `this` is the `self`
  return this.name;
};

var p = new Person("Bob");
console.log(p.getName());
// => "Bob"
```

### Using ES6 `class` es

```
class Person {
  constructor (name) {
    this.name = name;
  }
  getName () {
    return this.name;
  }
}

var p = new Person("Carol");
console.log(p.getName());
// => "Bob"
```

It's also very easy to inherit classes in ES6:



```
class Musician extends Person {
   constructor (name, instrument) {
     // Call the base class
     super(name);

     this.instrument = instrument;
   }

   play () {
     console.log(`${this.getName()} is playing ${this.instrument}`);
   }
}

var me = new Musician("Johnny B.", "piano");

// Get the name of the musician, who is also a person
console.log(me.getName());
// => "Johnny B."

me.play();
// => "Johnny B. is playing piano."
```

## Async vs Sync

Sometimes you have to wait a little bit for things to be done: such as when making a cake, you have to work on it and you have to wait a while for it to be ready. Most familiar things in our lives are asynchronous.

### Sync

Usually, synchronous operations send the response directly, using the `return` keyword:

```
// Synchronous sum
function sum (a, b) {
  return a + b;
}
var result = sum(40, 2);
// => 42
```



To have an async function, you need a source of async stuff. In this example, we will use the `setTimeout` function. It accepts a function as first argument (which is called *callback function*) and a number as second argument—representing the time delay after the function is called:

```
function asyncSum (a, b, cb) {
  setTimeout(function () {
    cb(a + b); // -----------+ This is the place
  }, 1000);    //            | where we call then
}              //            V callback function
asyncSum(40, 2, function (result) {
  console.log(result);
  // => 42
});
```

## What are callbacks?

`callback` s are functions which are sent as an argument to another function and are invoked when something happens.

```
function Cake() { /* Just a dummy class-like constructor for now */ }

// We won't make a ckae if we're busy
var busy = false;

function makeCake ( callback) {
  // If we're busy making a cake, send back an error
  if (busy) {
    return callback(new Error("Already busy with creating a cake. Please wa
  }
  // Well, if we weren't busy before, we are now
  busy = true;

  // Wait one second
  setTimeout(function () { // <- This is a callback function too. It is cal

    // After one second we call the callback function
    callback(null, new Cake());

    // Aj
```



```
makeCake(function (err, cake) {
  if (err) { console.error(err); }
  console.log("Made a cake.");
  // => "Made a cake."
});

// This will end with an error because we're busy already
makeCake(function (err, cake) {
  if (err) { console.error(err); }
  // !=> "Already busy with creating a cake. Please wait a bit and try agai
  console.log("Made a cake.");
});

// Wait 2 seconds
setTimeout(function () {
  // This will work again
  makeCake(function (err, cake) {
    if (err) { console.error(err); }
    console.log("Made a cake.");
    // => "Made a cake."
  });
}, 2000);
```

## Promises

There is a thing called *Promise*. Let's see an example:

```
function sum (a, b) {
   return Promise(function (resolve, reject) {
     // Let's wait a second and only then send the response
     setTimeout(function () {
       if (typeof a !== "number" || typeof b !== "number") {
         return reject(new TypeError("Please provide two numbers to sum."));
       }
       resolve(a + b);
     }, 1000);
   });
}

var myPromise = sum(40, 2);

myPromsise.then(function (result) {
  // After one second we have the response here
  console.log("> Summed 40 + 2: ", result);

  // Let's pass some invalid data and return another promise
  return sum(null, "foo");
}).then(function () {
  // This won't be called because we have an error
}).catch(function (err) {
  // Instead, this `catch` handler is called: (after another second)
  console.error(err);
  // => Please provide two numbers to sum.
});
```

A promise can be in one of these three states:

- `pending` : the operation is pending
- `fulfilled` : the operation was finished
- `rejected` : an error appeared, so the promise is *rejected*

StackOverflow Documention has a good section about promises here.

### Create and `throw` errors

To create an error, you have to use the `Error` constructor:



```
let myError = new Error("Something went really wrong.");

// You can even append custom data here
myError.code = "MY_FANCY_ERROR_CODE";
```

To throw an error, you have to use the `throw` statement:

```
throw new Error("Something bad happened.");
```

There are few types of errors. For example, if you validate the arguments passed to a function, use `TypeError` :

```
function sayHello(message) {
  if (typeof message !== "string") {
    throw new TypeError("The message should be a string.");
  }
  console.log(message);
}
```

### Callbacks

When you have a callback interface it's friendlier to send the errors using the callback functions.

### Promises

In the Promises (even on the interface) it's friendlier to send the errors using the reject functions.

## Error handling

In general, it is a good practice to validate the data you're passing to function (especially when they are coming from the user input). This way you can avoid `TypeError` s to be thrown.

If there's a



```
function assert (truly) {
  if (!truly) {
    throw new Error("Not true.");
  }
}

try {
  // This will *not* throw an error
  assert(42 === 42);

  // This will throw an error, so that will go in the catch
  assert(42 === -1);

  // Everything put after the place where the error was thrown
  // is not executed
} catch (e) {
  console.error(e);
  // => Not true.
}
```

Note if you have an async function (either callback-based or which returns a promise), you'll do something like this:

```
// Callback
fetchDataFromServer(function (err, data) {
  if (err) {
    /* do something if you have an error */
  } else {
    /* Do something with `data` */
  }
});

// Promise
fetchDataFromServer().then(function (data) {
    /* Do something with `data` */
}).catch(function (err) {
    /* do something if you have an error */
});
```



You can increment a variable name `x` using `++x` or `x++`. Similarly, you can decrement it by `--x` or `x--`.

The difference is that `++x` (`--x`) returns the incremented (decremented) value, while `x++` (`x--`) returns the previous value of `x`.

```
// Let's declare x
let x = 42;

// Store in y, the result of ++x
let y = ++x;

// Let's see the result
console.log(`x is ${x} and y is ${y}`);
→ 'x is 43 and y is 43'

// Now, store in y the result of x++. Note, x is now 43.
y = x++;

// Let's see again
console.log(`x is ${x} and y is ${y}`);
→ 'x is 44 and y is 43'

// So, `y` is 43, because `x++` returned the pre-incremented value (which w
```

## Loops

This will print all the integers from `1` to `42`.

`for`

```
for (var i = 1; i <= 42; ++i) {
  console.log(i);
}
```

Using `for ... in ...` can be used to iterate object keys:



```
var name = {
  first: "Johnny",
  last: "B."
};

for (var key in name) {
  if (name.hasOwnProperty(key)) {
    console.log(key, name[key]);
    // "first", "Johnny"
    // "last", "B."
  }
}
```

In ES6 there is a `for ... of ...` as well. It's pretty neat since it iterates any iterable objects (arrays, strings etc).

```
let numbers = [-1, 7, 42, 64];
for (let num of numbers) {
  console.log(num);
}
// -1
// 7
// 42
// 64
```

### `while`

```
var i = 1;
while (i <= 42) {
    console.log(i);
    ++i;
}
```

### `do - while`

```
var i = 0;
do {
    ++i;
    console.log(i);
} while (i < 42);
```

## Strict Mode

The strict mode, when turned on, shows potential errors made by the developer:

```
a = 42;
// 42

"use strict";
b = 42;
// Error: `b` is not defined
```

## Regular expressions

Regular expressions are a lot of fun. They are delimited by slashes:

```
/^[0-9]+$/gm.test("a")
// => false

/^[0-9]+$/gm.test("42")
// => true
```

A regular expression has a pattern and flags. The pattern in this example is `^[0-9]+$` —meaning if the input is one or more digits (from `0` to `9`), we validate it— while the flags are `g` (global) and `m` (multiline).

regex101.com is a good resource for visualizing and sharing regular expressions with others:



Regular expressions can be used to match things in strings, such as numbers:



```
// Get all the numbers from this input (note you'll get an array of strings
> "Alice was born in 1974, so she's 42 years old in 2016.".match(/[0-9]+/g)
[ '1974', '42', '2016' ]

// Want to get numbers? Map them using the `Number` function
> "Alice was born in 1974, so she's 42 years old in 2016.".match(/[0-9]+/g)
[ 1974, 42, 2016 ]

// BE CAREFUL that if there's no match `match` will return `null`
> "Bob is seven years old.".match(/[0-9]+/g)
null
```

Also, they can be used to `replace` parts of the strings:

```
// Mask all the digits of a credit card number, but the last 4
> "1234234534564567".replace(/\d(?=\d{4})/g, "*");
'************4567'
```

### Useful Math functions and constants



```
// Get the PI number
> Math.PI
3.141592653589793

// Get the E number
> Math.E
2.718281828459045

// Maximum between two numbers
> Math.max(7, 42)
42

// Minimum between numbers (arguments)
> Math.min(7, 42, 255, 264)
7

// Power
> Math.pow(2, 4)
16

// Random number between 0 (inclusive) and 1 (exclusive)
> Math.random()
0.2114640628617317

// Round
> Math.round(41.8)
42
```

There are trigonometric functions as well: `sin`, `cos`, `tan`, `atan` etc. Note these require radians. *π radians === 180 degrees*.

## Dates

Use the `Date` constructor to create date objects.

```
var now = new Date();
now.getFullYear();
// => 2016
```



If you want to have fancy stuff like formatting dates, etc., use libraries like `moment` or `daty` .

---

If you notice anything that can be improved, simply leave a comment, and I'll be happy to make the changes. 📝

Es6 Cheatsheet JavaScript



**Johnny B. (Ionică Bizău)**

Full Stack, *nix user, Pianist, College Dropout, Vegetarian, Jesus follower

Hello! I am Johnny B. 🙏 (Actually, my Romanian friends call me Ionică, which in English is basically: Johnny). ...and I 💔 emojis! I started coding around 2010, starting with simple static websites, Wordpress applications and t...

FOLLOW

## 31 Replies

Leave a reply

Reply

**sonu** a year ago

```
for (var i = 0; i < 7; ++i) {
setTimeout(function () {
console.log(i);
// => 8
// => 8
```

Show more

Reply

**Abhijit Khanna** 2 years ago

Hi Johnny, thanks for putting this together. In the Constructors and Classes section,
Using ES6 classes subsection, where you list the output in comments, shouldn't it output Carol, and not Ben?

Reply

Show more replies

**Daniel Shotonwa**

## Building an API with GraphQL and Rails



Image from [fiver]

As you all know, Rails is very popular for creating a web application in a short amount of time. Today, developers tend to separate concerns in their application by creating an API for the backend and consume it with any library of their choice.