ES6 cheatsheet

A quick reference cheatsheet of what's new in JavaScript for ES2015, ES2016, ES2017, ES2018 and beyond

Getting Started

Block-scoped

Let

```
function fn () {
  let x = 0
  if (true) {
    let x = 1 // only inside this `if`
  }
}
```

Const

```
const a = 1
```

`let` is the new `var`. Constants (`const`) work just like `let`, but cannot be reassigned. See: Let and const

Template Strings

Interpolation

```
const message = `Hello ${name}`
```

Multi-line string

```
const str = `
hello
the world
`
```

Templates and multiline strings. See: template strings

Binary and octal literals

```
let bin = 0b1010010
let oct = 0o755
```


See: Binary and Octal Literals

Exponential Operator

```
const byte = 2 **8
```

Same as: Math.pow(2, 8)

New library additions

New string methods

```
"hello".repeat(3)
"hello". includes("ll")
"hello". startsWith("he")
"hello".padStart(8)     // "hello"
"hello".padEnd(8)       // "hello"
"hello".padEnd(8, '!')  // hello!!!
"\u1E9B\u0323".normalize("NFC")
```

New Number Methods

```
Number.EPSILON
Number.isInteger(Infinity) // false
Number.isNaN("NaN")         // false
```

New Math methods

```
Math.acosh(3)    // 1.762747174039086
Math.hypot(3, 4)  // 5
Math.imul(Math.pow(2, 32) -1, Math.pow(2, 32) -2
```

New Array methods

```
//return a real array
Array.from(document.querySelectorAll("*"))
//similar to new Array(...), but without the spe
Array.of(1, 2, 3)
```

See: New library additions

kind

```
class Circle extends Shape {
```

Constructor

```
constructor (radius) {
  this.radius = radius
}
```

method

```
getArea () {
  return Math.PI *2 *this.radius
}
```

Call the superclass method

```
expand(n) {
  return super.expand(n) *Math.PI
}
```

Static methods

```
static createFromDiameter(diameter) {
  return new Circle(diameter /2)
}
```

Syntactic sugar for prototypes. See: classes

Private class

The javascript default field is public (`public`), if you need to indicate private, you can use (`#`)

```
class Dog {
  #name;
  constructor(name) {
    this.#name = name;
  }
  printName() {
    // Only private fields can be called inside
    console.log(`Your name is ${this.#name}`)
  }
}

const dog = new Dog("putty")
//console.log(this.#name)
//Private identifiers are not allowed outside cl
dog.printName()
```

Static private class

```
class ClassWithPrivate {
  static #privateStaticField;
static #privateStaticFieldWithInitializer = 42;

  static #privateStaticMethod() {
    // …
  }
}
```

Promises

make the commitment

```
new Promise((resolve, reject) => {
  if (ok) { resolve(result) }
  else { reject(error) }
})
```

for asynchronous programming. See: Promises

Using Promises

```
promise
  .then((result) => { ··· })
  .catch((error) => { ··· })
```

Using Promises in finally

```
promise
  .then((result) => { ··· })
  .catch((error) => { ··· })
  .finally(() => {
    /*logic independent of success/error */
  })
```

The handler is called when the promise is fulfilled or rejected

Promise function

```
Promise.all(···)
Promise.race(···)
Promise.reject(···)
Promise.resolve(···)
```

Async-await

```
async function run () {
  const user = await getUser()
  const tweets = await getTweets(user)
  return [user, tweets]
}
```

`async` functions are another way to use functions. See: Async Function

Destructuring

Destructuring assignment

Arrays

```
const [first, last] = ['Nikola', 'Tesla']
```

Objects

```
let {title, author} = {
  title: 'The Silkworm',
  author: 'R. Galbraith'
}
```

Supports matching arrays and objects. See: Destructuring

Defaults

```
const scores = [22, 33]
const [math = 50, sci = 50, arts = 50] = scores
```

```
//Result:
//math === 22, sci === 33, arts === 50
```

A default value can be assigned when destructuring an array or object

Function parameters

```
function greet({ name, greeting }) {
  console.log(`${greeting}, ${name}!`)
}
```

```
greet({ name: 'Larry', greeting: 'Ahoy' })
```

Destructuring of objects and arrays can also be done in function parameters

Defaults

```
function greet({ name = 'Rauno' } = {}) {
  console.log(`Hi ${name}!`);
}
```

```
greet() // Hi Rauno!
greet({ name: 'Larry' }) // Hi Larry!
```

Reassign keys

```
function printCoordinates({ left: x, top: y }) {
  console.log(`x: ${x}, y: ${y}`)
}
```

```
printCoordinates({ left: 25, top: 90 })
```

This example assigns `x` to the value of the `left` key

Loop

```
for (let {title, artist} of songs) {
  ...
}
```

Assignment expressions also work in loops

Object Deconstruction

```
const { id, ...detail } = song;
```

Use the `rest(...)` operator to extract some keys individually and the rest of the keys in the object

Spread operator Spread

Object Extensions

with object extensions

```
const options = {
  ...defaults,
  visible: true
}
```

No object extension

```
const options = Object.assign(
  {}, defaults,
  { visible: true })
```

The object spread operator allows you to build new objects from other objects. See: Object Spread

Array Expansion

with array extension

```
const users = [
  ...admins,
  ...editors,
  'rstacruz'
]
```

No array expansion

```
const users = admins
  .concat(editors)
  .concat([ 'rstacruz' ])
```

The spread operator allows you to build new arrays in the same way. See: Spread operator

Functions

Function parameters

Default parameters

```
function greet (name = 'Jerry') {
  return `Hello ${name}`
}
```

Rest parameters

```
function fn(x, ...y) {
  // y is an array
  return x * y.length
}
```

Extensions

```
fn(...[1, 2, 3])
//same as fn(1, 2, 3)
```

Default (default), rest, spread (extension). See: function parameters

Arrow function

Arrow functions

```
setTimeout(() => {
  ...
})
```

with parameters

```
readFile('text.txt', (err, data) => {
  ...
})
```

implicit return

```
arr.map(n => n*2)
//no curly braces = implicit return
//Same as: arr.map(function (n) { return n*2 })
arr.map(n => ({
  result: n*2
}))
//Implicitly returning an object requires parenthe
```

Like a function, but preserves `this`. See: Arrow functions

Parameter setting default value

```
function log(x, y = 'World') {
  console.log(x, y);
}

log('Hello') // Hello World
log('Hello', 'China') // Hello China
log('Hello', '') // Hello
```

Used in conjunction with destructuring assignment defaults

```
function foo({x, y = 5} = {}) {
  console.log(x, y);
}

foo() // undefined 5
```

name attribute

```
function foo() {}
foo.name // "foo"
```

length property

```
function foo(a, b){}
foo.length // 2
```

Objects

Shorthand Syntax

```
module.exports = { hello, bye }
```

same below:

```
module.exports = {
  hello: hello, bye: bye
}
```

See: Object Literals Enhanced

method

```
const App = {
  start () {
    console.log('running')
  }
}
//Same as: App = { start: function () {···} }
```

See: Object Literals Enhanced

Getters and setters

```
const App = {
  get closed () {
    return this.status === 'closed'
  },
  set closed (value) {
    this.status = value ? 'closed' : 'open'
  }
}
```

See: Object Literals Enhanced

Computed property name

```
let event = 'click'
let handlers = {
  [`on${event}`]: true
}
//Same as: handlers = { 'onclick': true }
```

See: Object Literals Enhanced

Extract value

```
const fatherJS = { age: 57, name: "Zhang San" }
Object.values(fatherJS)
//[57, "Zhang San"]
Object.entries(fatherJS)
//[["age", 57], ["name", "Zhang San"]]
```

Modules module

Imports import

```
import 'helpers'
//aka: require('···')
```

```
import Express from 'express'
//aka: const Express = require('···').default ||
```

```
import { indent } from 'helpers'
//aka: const indent = require('···').indent
```

```
import *as Helpers from 'helpers'
//aka: const Helpers = require('···')
```

```
import { indentSpaces as indent } from 'helpers'
//aka: const indent = require('···').indentSpace
```

`import` is the new `require()`. See: Module imports

Exports export

```
export default function () { ··· }
//aka: module.exports.default = ···
```

```
export function mymethod () { ··· }
//aka: module.exports.mymethod = ···
```

```
export const pi = 3.14159
//aka: module.exports.pi = ···
```

```
const firstName = 'Michael';
const lastName = 'Jackson';
const year = 1958;
export { firstName, lastName, year };
```

```
export *from "lib/math";
```

`export` is the new `module.exports`. See: Module exports

as keyword renaming

```
import {
  lastName as surname // import rename
} from './profile.js';

function v1() { ... }
function v2() { ... }

export { v1 as default };
//Equivalent to export default v1;

export {
  v1 as streamV1, // export rename
  v2 as streamV2, // export rename
  v2 as streamLatestVersion // export rename
};
```

Dynamically load modules

```
button.addEventListener('click', event => {
  import('./dialogBox.js')
    .then(dialogBox => {
      dialogBox. open();
    })
    .catch(error => {
      /*Error handling */
    })
});
```

ES2020 Proposal introduce `import()` function

import() allows module paths to be dynamically generated

```
const main = document.querySelector('main')

import(`./modules/${someVariable}.js`)
  .then(module => {
    module.loadPageInto(main);
  })
  .catch(err => {
    main.textContent = err.message;
  });
```

import.meta

ES2020 Added a meta property `import.meta` to the `import` command, which returns the meta information of the current module

```
new URL('data.txt', import.meta.url)
```

In the Node.js environment, `import.meta.url` always returns a local path, that is, a string of the `file:URL` protocol, such as `file:/// home/user/foo.js`

Import Assertions

static import

```
import json from "./package.json" assert {type: "json"}
//Import all objects in the json file
```

Dynamic Import

```
const json =
    await import("./package.json", { assert: { type: "json" } })
```

Generators

Generator function

```
function*idMaker () {
  let id = 0
  while (true) { yield id++ }
}
```

```
let gen = idMaker()
gen.next().value // → 0
gen.next().value // → 1
gen.next().value // → 2
```

it's complicated. See: Generators

For..of + iterator

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
  return { done: false, value: cur }
      }
    }
  }
}

for (var n of fibonacci) {
  // truncate sequence at 1000
  if (n > 1000) break;
  console.log(n);
}
```

For iterating over generators and arrays. See: For..of iteration

Relationship with Iterator interface

```
var gen = {};
gen[Symbol.iterator] = function*() {
  yield 1;
  yield 2;
  yield 3;
};

[...gen] // => [1, 2, 3]
```

The `Generator` function is assigned to the `Symbol.iterator` property, so that the `gen` object has the `Iterator` interface, which can be traversed by the `...` operator

Symbol.iterator property

```
function*gen() { /*some code */}
var g = gen();

g[Symbol.iterator]() === g // true
```

`gen` is a `Generator` function, calling it will generate a traverser object `g`. Its `Symbol.iterator` property, which is also an iterator object generation function, returns itself after execution

see also

Learn ES2015(babeljs.io)
ECMAScript 6 Features Overview (github.com)