

# ES2015+ cheatsheet

Proudly sponsored by

**Sustain Podcast #33**  How Font Awesome raised \$1mm on Kickstarter

*ethical* ad by CodeFund

A quick overview of new JavaScript features in ES2015, ES2016, ES2017, ES2018 and beyond.

## Block scoping

### Let

```
function fn () {  
  
  if (true) {  
  
  }  
  
}
```

### Const

```
const a = 1
```

`let` is the new `var`. Constants work just like `let`, but can't be reassigned. See: [Let and const](#)

## Backtick strings

### Interpolation

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

### Multiline strings

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

Templates and multiline strings. See: [Template strings](#)

## Binary and octal literals

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

See: [Binary and octal literals](#)

## New methods

### 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")
```

See: [New methods](#)

## Classes

```
class Circle extends Shape {
```

### Constructor

```
  this.radius = radius
}
```

### Methods

```
  return Math.PI * 2 * this.radius
}
```

### Calling superclass methods

```
  expand (n) {

  }
```

### Static methods

```
  return new Circle(diameter / 2)
}
}
```

Syntactic sugar for prototypes. See: [Classes](#)

## Exponent operator

```
// Same as: Math.pow(2, 8)
```

# Promises

## Making promises

```
if (ok) { resolve(result) }  
else { reject(error) }  
})
```

For asynchronous programming. See: [Promises](#)

## Using promises

```
promise
```

## Using promises with finally

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

The handler is called when the promise is fulfilled or rejected.

## Promise functions

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

## Async-await

```
async function run () {  
  
    return [user, tweets]  
}
```

async functions are another way of using functions.

See: [async function](#)

# Destructuring

## Destructuring assignment

Arrays

Objects

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

Supports for matching arrays and objects. See: [Destructuring](#)

## Default values

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

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

Default values can be assigned while destructuring arrays or objects.

## Function arguments

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

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

Destructuring of objects and arrays can also be done in function arguments.

## Default values

```
console.log(`Hi ${name}!`);  
}
```

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

## Reassigning keys

```
console.log(`x: ${x}, y: ${y}`)  
}
```

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

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

## Loops

```
...  
}
```

The assignment expressions work in loops, too.

## Object destructuring

Extract some keys individually and remaining keys in the object using rest (...) operator

# Spread

## Object spread

with Object spread

```
const options = {  
  visible: true  
}
```

without Object spread

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

The Object spread operator lets you build new objects from other objects.

See: [Object spread](#)

## Array spread

with Array spread

```
const users = [  
  
  'rstacruz'  
]
```

without Array spread

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

The spread operator lets you build new arrays in the same way.

See: [Spread operator](#)

# Functions

## Function arguments

Default arguments

```
    return `Hello ${name}`  
  }
```

#### Rest arguments

```
    // y is an Array  
    return x * y.length  
  }
```

#### Spread

```
    // same as fn(1, 2, 3)
```

Default, rest, spread. See: [Function arguments](#)

## Fat arrows

#### Fat arrows

```
    ...  
  })
```

#### With arguments

```
    ...  
  })
```

#### Implicit return

```
    // No curly braces = implicit return  
    // Same as: numbers.map(function (n) { return n * 2 })
```

```
    // Implicitly returning objects requires parentheses around the object
```

Like functions but with `this` preserved. See: [Fat arrows](#)

# Objects

## Shorthand syntax

```
module.exports = { hello, bye }  
// Same as: module.exports = { hello: hello, bye: bye }
```

See: [Object literal enhancements](#)

## Methods

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

See: [Object literal enhancements](#)

## Getters and setters

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

See: [Object literal enhancements](#)

## Computed property names

```
let event = 'click'  
let handlers = {  
  
}  
// Same as: handlers = { 'onclick': true }
```

See: [Object literal enhancements](#)

## Extract values



```
const fatherJS = { age: 57, name: "Brendan Eich" }
```

```
Object.values(fatherJS)  
// [57, "Brendan Eich"]  
Object.entries(fatherJS)  
// [["age", 57], ["name", "Brendan Eich"]]
```

# Modules

## Imports

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

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

```
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('...').indentSpaces
```

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

## Exports

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

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

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

export is the new `module.exports`. See: [Module exports](#)

# Generators

## Generators

```
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 iteration

```
for (let i of iterable) {  
  ...  
}
```

For iterating through generators and arrays. See: [For..of iteration](#)