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

Block scoping

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 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

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

Methods

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

Calling superclass methods

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

Static methods

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

Syntactic sugar for prototypes. See: Classes

Exponent operator

```
const byte = 2 ** 8
// Same as: Math.pow(2, 8)
```

Promises

Making promises

```
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 with finally

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

{: data-line="4"}

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

`async` functions are another way of using functions.

See: async function

Destructuring

Destructuring assignment

Arrays

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

Objects

```
let {title, author} = {
  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

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

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

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

Default values

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

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

Reassigning 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.

Loops

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

The assignment expressions work in loops, too.

Object destructuring

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

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

Spread

Object spread

with Object spread

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

{: data-line="2"}

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 = [
  ...admins,
  ...editors,
  '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

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

Rest arguments

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

Spread

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

Default, rest, spread. See: Function arguments

Fat arrows

Fat arrows

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

With arguments

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

Implicit return

```
numbers.map(n => n * 2)
// No curly braces = implicit return
// Same as: numbers.map(function (n) { return n * 2 })
numbers.map(n => ({
  result: n * 2
}))
// Implicitly returning objects requires parentheses around the object
```

{: data-line="1,4,5,6"}

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

{: data-line="2"}

See: Object literal enhancements

Getters and setters

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

See: Object literal enhancements

Computed property names

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