

JS - 103

- 0. Intro to ECMA versions (5 to 6)
- 1. Let & const
- 2. Template strings
- 3. Arrow functions
- 4. Destructuring
- 5. Spread operator
- 6. ES5 prototypes and ES6 classes

0. ES5 & ES6



****EcmaScript is a scripting-language specification standarized by Ecma International created in order to standarize Javascript.****

Created by Brendan Eich of NetScape.

****First version from 1997!****

**2015 -> ES6 with new features!* 🎉🎉*

1. let and const



*// Let and const are never globally registered, which means they don't create
// properties of the window object when declared globally.*

```
var human = 'Jon'  
let dog = 'Odie'  
const cat = 'Garfield'  
  
console.log(this.human) // Jon  
console.log(this.dog) // undefined  
console.log(this.cat) // undefined
```



```
// let is blocked scoped (that means that let doesn't care about local or  
// global, it just cares about the {block} it is declared within). They are  
// never globally registered.
```

```
function letScoping () {  
    let cat = 'Garfield'  
    if (true) {  
        let cat = 'Snowball II'  
        console.log(cat) // Snowball II  
    }  
    console.log(cat) // Garfield  
}
```

// VS.

```
function varScoping () {  
    var cat = 'Garfield'  
    if (true) {  
        var cat = 'Snowball II' // Same variable!  
        console.log(cat) // Snowball II  
    }  
    console.log(cat) 🤯 // Snowball II  
}
```



// Still we can access let variables that exist on a higher block.

```
function letScoping () {  
  let cat = 'Garfield'  
  if (true) {  
    console.log(cat) // Garfield  
  }  
  console.log(cat) // Garfield  
}
```



🤔 // Will this work?

```
function getHuman (petName) {  
  switch (petName) {  
    case 'Garfield':  
      let human = 'Jon'  
      break  
    case 'Cheshire':  
      let human = 'Alice'  
      break  
    default  
      let human = ''  
      break  
  }  
  return human  
}
```



🎉 // We can fix it by declaring new scopes inside the cases.

```
function getHuman (petName) {  
  switch (petName) {  
    case 'Garfield': {  
      let human = 'Jon'  
      break  
    }  
    case 'Cheshire': {  
      let human = 'Alice'  
      break  
    }  
    default {  
      let human = ''  
      break  
    }  
  }  
  return human  
}
```



🤔 // What does this print and why?

```
for (var number = 0; number < 10, number++) {  
    setTimeout(function (number) {  
        console.log(number)  
    }, 1000) // Run the function after one second  
}
```



🎉 // We can fix it by using `let`, which is block scoped and doesn't get
// overwritten by latter declarations, as for each iteration the for loop
// creates a new scope.

```
for (let number = 0; number < 10, number++) {  
    setTimeout(function (number) {  
        console.log(number)  
    }, 1000) // Run the function after one second  
}
```



```
// const is used to create variables that cannot be reassigned. It's used  
// for constants or any variable that we want to protect, so it doesn't get  
// assigned any other unexpected value.
```

```
const API_ENDPOINT = 'https://www.onesait-api-services.com/api/v1/'  
const API_KEY = '2ejovPwSdj0HCnG7TuS0Y0KSU1kQRoEO'  
  
function double (numbers) {  
    const two = 2  
    const doubledNumbers = numbers.map(function (number) {  
        return number * two  
    })  
    return doubledNumbers  
}
```



*// const is block scoped (so is let), so in different scopes we can declare
// variables using let and const using the same name without issues.*

```
function blockScope () {  
  const human = 'Jon'  
  let dog = 'Oddie'  
  const cat = 'Garfield'  
  if (true) {  
    const human = 'Alice'  
    let cat = 'Cheshire'  
    console.log(human) // 'Alice'  
    console.log(dog) // 'Oddie'  
  }  
  console.log(cat) // Garfield  
}
```



*// const must be initialised (unlike let or var), because we cannot assing a
// value to a const by definition.*

```
const iHaveNoValueYet // Throws an error
iHaveNoValueYet = "I'm a const variable"
console.log(iHaveNoValueYet)
```

```
let iHaveNoValueYet
iHaveNoValueYet = "I'm a let variable"
console.log(iHaveNoValueYet)
```



```
// const also works on objects and arrays, so they are protected against re  
// assignation, but object key-value pairs and array values are not, and can  
// change. Remember, const only protects against reassignment
```

```
const house = {  
    human: 'Alice',  
    cat: 'Cheshire'  
}
```

// Attempting to reassign the object throws an error

```
house = {  
    human: 'Jon',  
    cat: 'Garfield'  
}
```

// However we can change the values

```
house.human = 'Jon'  
house.cat = 'Garfield'  
house.dog = 'Oddie' // Even assign new key-value pairs
```



// It works the same way with arrays

```
const cats = ['Cheshire', 'Garfield']
```

// Attempting to reassign the array throws an error

```
cats = ['Snowball II', 'Cheshire', 'Garfield']
```

// However we can add new values

```
cats.push('Snowball II')
```

2. template strings



```
// Template strings are strings that allow embedded expressions, multi line
// text and string interpolation. Template strings are enclosed by the back
// tick `` and can contain placeholders indicated by a dollar sign ($) and
// curly braces

const string = 'normal string'
const templateString = `I'm more than a ${ string }` // I'm more than a normal string
```



```
// Template strings can be used to write multiline text. While using normal  
// strings the way to achieve this was by using the backspace character (\n),  
// with template strings we can write multiline text by writing a multiline  
// template string
```

```
// String  
const multiLineString = 'I need more than \n one line'
```

```
// Template string  
const multiLineTemplateString = `I need more than  
one line`
```



```
// Embedding expressions with normal strings requires concatenating the result  
// of the expression with our string
```

```
const resultAsString = '2 + 4 = ' + (2 + 4)
```

```
// With template strings we can compute the result inside the placeholder
```

```
const resultAsTemplateString = `2 + 4 = ${2 + 4}`
```



```
// A more advanced type of template strings is tagged templates, which allow us
// to parse template strings with a function. This function takes as the first
// argument a list of the normal strings contained in the template and the
// remaining arguments are the result of the expressions. They can return anything.

const house = {
  human: 'Jon',
  pets: ['Garfield', 'Oddie', 'Bird']
}

function pets (strings, name, pets) {

  let petCount = pets.length

  if (petCount > 2) {
    petCount = 'too many'
  }

  return `${name}${strings[1]}${petCount}${strings[2]}`
}

// This yields: Jon has too many pets
const result = pets`${house.human} has ${house.pets} pets`
```

3. arrow functions



*// Arrow functions are an alternative to regular functions, written in a more
// syntactically compact way. They don't have their own bindings to the this and
// arguments keywords, but this is an intended feature.*

```
(a, b) => { return a + b }
```



*// The basic syntax for an arrow function takes arguments within parentheses,
// and expressions contained in it's body, surrounded by curly braces.*

(/ arguments */) => { /* expressions */ }*



// We can write arrow functions in different ways

// When just one argument is passed we don't need the parentheses

```
a => { return a * 2 }
```

// When the expression in the body is just used to return a value we don't need

// the curly braces

```
(a, b) => a + b
```

// If the expression in the body returns an object literal, we need parentheses

// within the body

```
(key, value) => ({ key: value })
```

// If the function takes no arguments, we need to use empty parentheses

```
() => console.log('🍕')
```



```
// We can store arrow functions in a variable so we can use them anywhere within  
// our code
```

```
const capitalize = (string) => string.charAt(0).toUpperCase() + string.slice(1)  
  
console.log(capitalize('garfield')) // 'Garfield'
```



🤔 // Are these arrow functions well written? What do they return?

// 1 (a === 4, b === 3)

```
const sum = (a, b) => a + b
```

// 2 (array === ['a', 'b', 'c'])

```
const firstElement = array => array[0]
```

// 3 (object === { a: 5, b: 2 })

```
const concat = (object) => { return `${object.a} - ${object.b}` }
```

// 4 (human === 'Jon', cat === 'Garfield')

```
const makeObject = human, cat => ({ human: human, cat: cat })
```



```
// The most common use for these functions is within methods that take a function
// as an argument, such as map(), reduce(), find(), filter()...

const animals = [
  {
    name: 'cat',
    legs: 4
  }, {
    name: 'dog',
    legs: 4
  }, {
    name: 'human',
    legs: 2
  }
]

const twoLeggedAnimals = animals.filter(animal => animal.legs === 2)

const dog = animals.find(animal => animal.name === 'dog')

const animalNames = animals.map(animal => animal.name)

const totalLegs = animals.reduce((total, current) => total += current.legs, 0)
```



```
// Arrow functions don't create context, which means they don't have their own  
// 'this' object. The expression within the arrow function will look up through  
// the contexts until it finds a valid 'this' and use it.
```

```
function updateThis () {  
  
    this.value = 0  
  
    const addOneToThis = () => {  
        this.value ++  
    }  
  
    console.log(this.value) // 0  
    addOneToThis()  
    console.log(this.value) // 1  
  
}
```

4. destructuring



*// We can assign default values to the variables in case the position of the
// array they try to acces doesn't exist or is undefined*

```
const animals = [ 'Garfield', undefined ]
```

```
const [ cat, dog = 'Oddie', horse = 'Bojack' ] = animals
```




```
// With objects works similarly, but with basic destructuring variables need  
// to be named with the object's keys. This allows us to declare them in any  
// order.
```

```
const house = { human: 'Jon', cat: 'Garfield', dog: 'Oddie' }
```

```
const { human, dog, cat } = house
```

```
console.log(human, dog, cat) // Jon Oddie Garfield
```




```
// For functions that take objects or arrays as arguments we can use destructuring  
// to access each of the values within these data structures directly in the  
// arguments.
```

```
const person = { name: 'Jon', surname: 'Q. Arbuckle' }
```

```
function concatenateName ({ name, surname }) {  
    return name + surname  
}
```

```
const fullName = concatenateName(person)
```

```
console.log(fullName) // Jon Q. Arbuckle
```



🎉 // This is how things we have learned help us write better and cleaner code!

// From this:

```
const person = { name: 'Jon', surname: 'Q. Arbuckle' }
```

```
function concatenateName (person) {
  let name = person.name
  let surname = person.surname
  return name + surname
}
```

```
const fullName = concatenateName(person)
```

// To this!

```
const person = { name: 'Jon', surname: 'Q. Arbuckle' }
```

```
const fullName = ({ name, surname }) => `${name} ${surname}`
```


5. spread operator



*// The spread operator allows iterables to be expanded. We can think of it as
// spreading butter over bread.*

```
const array = [ 1, 3, 4 ]
```

```
function sum (x, y, z) {  
    return x + y + z  
}
```

// Applying the function to the array old-school style

```
console.log(sum(array[0], array[1], array[2])) // 8
```

// Using the spread operator!

```
console.log(sum(...array)) // 8
```



```
// We can make use of this operator for merging and concatenating arrays in a  
// simpler way
```

```
const a = [ 1, 2 ]  
const b = [ 3, 4 ]  
const c = [ 5, 6, 7 ]
```

```
// old-school style  
const concatenated = a.concat(b).concat(c)
```

```
// using the spread operator  
const concatenated = [ ...a, ...b, ...c ] // 1, 2, 3, 4, 5, 6, 7
```

```
// We can also merge an array into an arbitrary position of another array  
const merged = [ 1, 2, ...b, 5, 6 ] // 1, 2, 3, 4, 5, 6
```



```
// It can also be used to clone arrays to another variable, passing the values  
// as values instead of as a reference to the original array, so mutations to the  
// cloned array don't affect the original one
```

```
const a = [ 1, 2, 3 ]  
const b = [ ...a ] // b is now a clone of a
```

```
b.push(4) // Mutation being made to b
```

```
console.log(...b) // 1, 2, 3, 4  
console.log(...a) // 1, 2, 3 -> Still remains the same
```



```
// The rest operator works as the spread operator, but used when destructuring,  
// so it holds the remaining values that where not destructured  
  
const user = [ 'Jon', 'Q. Arbuckle', '28000', 'Madrid', 'Spain' ]  
  
function getFullNameAndAddress ([ name, surname, ...rest ]) {  
    // rest is now [ '28000', 'Madrid', 'Spain' ]  
    return `${name} ${surname}, ${rest.join(' ')}`  
}  
  
console.log(getFullNameAndAddress(user)) // Jon Q. Arbuckle, 28000 Madrid Spain
```



```
// Spread also works with objects, spreading the key/value pairs.

const user = { name: 'Jon', surname: 'Q. Arbuckle' }
const address = { zipCode: '28000', city: 'Madrid', country: 'Spain' }

const userInfo = { ...user, ...address }
// { name: 'Jon', surname: 'Q. Arbuckle', zipCode: '28000', city: 'Madrid', ... }
```



```
// When merging two objects with keys that are the same, the values from the latter  
// prevail over the previous values. This is useful for declaring objects with  
// default values.
```

```
const defaultConfiguration = { method: 'get', protocol: 'http', port: '8080' }
```

```
const customConfiguration = { ...defaultConfiguration, protocol: 'https' }
```

```
// { method: 'get', protocol: 'https', port: '8080' } -> http changed to https!
```

6. prototypes and classes



```
// Everything in javascript ends up being an object under the hood. This means  
// that every data structure has a similar structure to that of an object. It  
// has properties that can be values or even methods that interact with the data.
```

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

```
console.log(sum.name) 🤔
```

```
console.log(sum.length) 🤔
```



```
// Javascript is often described as a prototype-based language. Objects can have  
// a prototype object which acts as a template or model for every object we create,  
// which will inherit methods and properties from this prototype.  
  
const array = [ 1, 2, 3 ]  
  
console.log(array.length) // This property is inherited from the prototype  
console.log(array.map(...)) // This method is inherited from the prototype too!
```

```
// We can declare a new data structure of our own, with it's own properties and
// methods

function Cat (name, color, age) {

    this.name = name
    this.color = color
    this.age = 0
    this.lifes = 7

    this.isAlive = function () {
        return this.lifes > 0
    }

    this.growOld = function () {
        this.age ++
        if (this.age > 15) {
            this.lifes --
            this.age = 0
        }
    }
}

const cat = new Cat('Garfield', 'orange', 5)

cat.growOld()

console.log(cat.color) // orange
console.log(cat.age) // 1
console.log(cat.isAlive()) // true
```



```
// New structures inherit properties and methods from their prototype. That's  
// why our Cat has methods and properties that we didn't define, but the chain  
// of prototypes allows us to access properties and methods from the Object  
// prototype!
```

```
cat.valueOf() // Try it on the console!
```



```
// We can modify existing prototypes for data structures that are 'native' data
// structures in javascript. For instance, we can customize the Array prototype
// and add new methods:

Array.prototype.whatAmI = 'An array, duh...' // Array is the 'native' array!
Array.prototype.rightShift = function () {
  this.unshift(this.pop()) // Shift array
  return this // Return shifted array
}

// Now all arrays will have these methods and properties!

const week = [ 'MONDAY', 'TUESDAY', 'WEDNESDAY', 'THURSDAY', 'SATURDAY', 'SUNDAY' ]

console.log(week.whatAmI) // An array, duh...
console.log(week.rightShift()) // [ 'SUNDAY', 'MONDAY', 'TUESDAY', 'WEDNESDAY', 'THURSDAY',
'SATURDAY' ]
```



```
// Classes were introduced in ECMAScript 2015 and are basically syntactical  
// sugar over the existing prototype inheritance with some differences we will  
// take a look at.
```

```
function Cat () {  
    /* Old school 'classes' */  
}
```

```
class Cat {  
    /* We use the class keyword followed by the name of the class */  
}
```



```
// The constructor is just a method, but one that is special and used for
// creating and initializing an object with a class. Is the method that gets
// called when we use the 'new' keyword for creating an object of said class.

class Cat {

    constructor (name, color, age) {
        this.name = name
        this.color = color
        this.age = age
    }

}

const cat = new Cat('Garfield', 'orange', 5) // constructor('Garfield', 'orange', 5)
```

```
// As with prototypes we can declare methods within the class body, but we can use
// computed properties that return data based on the instance properties.

class Cat {

    constructor (name, color, age) {
        this.name = name
        this.color = color
        this.age = age
    }

    get description () {
        return `${this.name} is an ${this.color} cat, aged ${this.age}`
    }

    growOld () {
        this.age ++
    }

}

const cat = new Cat('Garfield', 'orange', 5)

cat.growOld

console.log(cat.description) // Garfield is an orange cat, aged 6
```



```
// Classes can extend other classes and inherit their methods and properties by
// using the keyword extends followed by the 'parent' class.
```

```
class Kitty extends Cat {

    constructor (name, color, age) {
        super(name, color)
        if (age > 3) throw Error('This is a fully grown cat!')
        this.age = age
    }
}
```

```
// We can still use growOld and get description because we inherited them!
```

```
}
```



```
// While we can overwrite the parent's class methods or properties, we can  
// always access them within the class using super.
```

```
class Kitty extends Cat {  
  
    constructor (name, color, age) {  
        super(name, color)  
        if (age > 3) throw Error('This is a fully grown cat!')  
        this.age = age  
    }  
}
```

```
// We can overwrite growOld but still inherit some functionallity  
growOld () {  
    super.growOld() // Invoke the growOld method from the parent class  
    if (this.age > 3) throw Error('This kitty is now a full grown cat :( ')  
}  
}
```



References

MDN web docs:

let

<https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/let>

const

<https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/const>

template strings

https://developer.mozilla.org/docs/Web/JavaScript/Reference/Template_literals

arrow functions

https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/Arrow_functions

destructuring

https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

spread operator

https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Spread_syntax

rest operator

https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/rest_parameters

prototypes

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

classes

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

Github

<https://github.com/pedro-rodalia/JS-103>

Repl.it

https://repl.it/@pedro_rodalia