Índice

Sección III - Estructuras de Control y Funciones

Sección IV - Arrays

Sección V - Programación Orientada a Objetos en JavaScript (POO)

Sección VI - Asincronismo

Sección VIII - Complementos

Resumen clase por clase:

Sección II - Primeros Pasos

03 - Variables

Declaración:

Con el prefijo '*var*' podemos crear una variable.

ej:

```
var nombre
nombre = 'Aristóteles'
var apellido = 'De Atenas'      //se puede declarar en una sóla linea
```

Otra forma:

```
var nombre = 'Tiglath', apellido = 'Azur'      // en la misma línea

var nombre, apellido, edad

nombre = 'Sherezade'
apellido = 'Efendi'
edad = '24'
```

Javascript posee tipos de variables fácilmente intercambiables en la declaración de su valor. Esto significa que es un lenguaje debilmente tipeado. Esto quiere decir que las variables declaradas pueden ser de cualquier tipo; string, number, boolean, etc. No conocemos exactamente el tipo de variables que estamos usando.

ej:

```
var peso = 75          // number
var peso = '75Kgs'     // string
```

Encadenar variables:

Las variables en JS pueden encadenarse con otras variables y/o con texto con un signo '+'.

```
console.log('Mi nombre es ' + nombre + ' ' + apellido)
```

También podemos usar comillas invertidas y escapar las variables entre ${}.

```
console.log(`Mi nombre es ${nombre} ${apellido}`)
```

04 - Variables - Strings

Sobre una variable tipo string los métodos más básicos que podemos usar son:

```
.toUpperCase()  // convierte todo el string en mayúsculas
.toLowerCase()  // convierte todo el string en minúsculas
.charAt(n)      // nos dirige al caracter de la posición 'n'
.length         // arroja un número que equivale a la cantidad de caracteres del string
.substr(n, n1)  // arroja un string que comienza en la letra 'n' y termina en la letra
'n1' del string dado.
```

Ej:

```
var nombre = 'Sacha', apellido = 'Lifszyc'

var nombreEnMayusculas = nombre.toUpperCase()
// "SACHA"

var apellidoEnMinusculas = apellido.toLowerCase()
// "lifszyc"
```

```
var primeraLetraDelNombre = nombre.charAt(0)
// "S"

var cantidadDeLetrasDelNombre = nombre.length
// 5

var nombreCompleto = `${nombre} ${apellido.toUpperCase()}`
// también podemos acceder dentro de los escapes. Esta manera de concatenar se llama interp

var str = nombre.substr(1, 2)
// "ac"
```

05 - Variables - Numbers

Sobre una variable tipo *number* los métodos básicos son:

```
var += n        // suma el valor agregado después de += a la variable
var -= n        // resta el valor agregado después de -= a la variable
Math.round      // redondea un floating-point number al integer más cercano
toFixed(n)      // devuelve el valor mismo de la variable con 'n' dígitos después del
'.' . La variable pasa de ser número a string
parseFloat(var) // convierte un string en un floating-point number
```

Ej:

```
var a = 10
var b = 5

a += 1 // a = a + 1
// a = 11

a -= 1  // a = a - 1
// a = 10

var c = a + b
// (Suma) c = 15

c = a - b
// (Resta) c = 5

var d = 200.3

var e = a / b
// (División) e = 2
```

```
d = d * 3
// (Multiplicación) d = 600.9000000000001
//javascript arroja siempre esta cantidad de números después de la ',' para números

d = d * 100 * 3 / 100
// d = 600.9

d = Math.round(d * 100 * 3 / 100)
// igual que sentencia anterior pero más preciso

d = d.toFixed(2)
// d = "600.90" // Muestra 2 dígitos después de la coma.

d = parseFloat(d)
// d = 600.9
```

06 - Funciones

Las funciones son pedazos de código reutilizable. Se declaran precedidas de la palabra clave 'function', van seguidas de un nombre para la función y lo que queremos que haga dentro de '{}'.

ej: function myFunction(){ console.log('Hola Mundo!!!') } myFunction() // Hola Mundo!!!

Puedo agregar parámetros dentro de los paréntesis para hacer mi función dinámica.

ej: function myFunction(nombre) { console.log(`Hola ${nombre}.`) } myFunction('Walter') // Hola Walter.

Código de la clase:

```
var nombre = 'Sacha', edad = 28

function imprimirEdad(n, e) {
  console.log(`${n} tiene ${e} años`)
}

imprimirEdad(nombre, edad)
imprimirEdad('Vicky', 28)
imprimirEdad('Eric', 24)
imprimirEdad('Darío', 27)
imprimirEdad(25, 'Carlos')
imprimirEdad('Juan')
```

07 - Alcance de las Funciones

A qué variables puede acceder una función y qué valores van a tener esas variables al momento de invocar una función.

Si una variable no está definida dentro de una función es de alcance global; es decir, se puede acceder a esta desde cualquier función.

```
var nombre = 'Sacha'// global

functionnombreMayuscula(){
        nombre = nombre.toUpperCase()
        console.log(nombre)
}

nombreMayuscula()
// Esta función modofica la variable.
```

Al definir una variable de forma global, se la asigna al objeto global. En un servidor el contexto es por ejemplo node, en el browser el objeto global es 'window'. La variable 'nombre'en el ejemplo anterior queda asignada al objeto global 'window'.

```
window.nombre
// "SACHA"
```

Es buena práctica que las funciones no modifiquen variables que no están dentro de ellas misma. Para esto podemos usar los parámetros en las funciones.

```
functionnombreMayuscula(n){
        n = n.toUpperCase()
        console.log(n)
}

nombreMayuscula('Juan')
// "JUAN"
```

También podemos usar específicamente la variable 'nombre' aunque la hayamos ya definido anteriormente, es decir que las variables como parámetros dentro de una función no modificarán las variables definidas por fuera de ellas con el mismo nombre.

```
functionnombreMayuscula( nombre ){
        nombre = nombre.toUpperCase()
        console.log(nombre)
}

nombreMayuscula()
```

08 - Objetos

Cómo declararlos, cuáles son sus ventajas, cómo asignarles atributos y cómo trabajar con ellos dentro de las funciones.

Los objetos se definen delimitados mediante llaves {}

```
var objeto = {}
```

Un atributo se compone de una clave (key) y un valor (value), que se separan entre sí por dos puntos “”:””.

```
var yigit = {
        nombre: 'Yigit'
}
```

Los valores pueden ser de tipo string, número, booleano, etc. Cada atributo está separado del siguiente por una coma. Un objeto puede tener todos los atributos que sean necesarios.

```
var yigit = {
        nombre: 'Yigit',
        apellido: 'Sultan'
        edad: 13
        ...
}
```

Escribir el nombre de un objeto separado por un punto del nombre de un atributo, nos permite acceder al valor de dicho atributo para ese objeto. Un objeto también se puede pasar como atributo en una función.

```
yigit.nombre
// "Yigit"

function imprimirMayuscula(persona){
        console.log(persona.nombre.toUpperCase())
}

imprimirMayuscula(yigit)
// YIGIT
```

Las últimas versiones de JavaScript nos permiten desglosar el objeto dentro de los parámetros de la función. Lo hacemos pasando el *key* del atributo entre "{}" dentro de los parámetros de la declaración de la función.

```
function imprimirMayuscula({ nombre }){
        console.log(nombre.toUpperCase())
}

imprimirMayusculas(yigit)
// YIGIT
```

09 - Desestructurar Objetos

Otra forma de acceder a los atributos de los objetos es la desestructurización de los mismos. Para no duplicar las variables, introducir como nombre de la variable el parámetro de la segunda variable entre '{}'.

Ej:

```
var yigit = {
        nombre: 'Yigit',
        apellido: 'Sultan'
}

function imprimirMayuscula(persona){
        var { nombre } = persona  // var nombre = persona.nombre
        console.log(nombre.toUpperCase())
}

imprimirMayuscula(yigit)
// YIGIT
```

10 - Parámetros como referencia o como valor

Javascript se comporta de manera distinta cuando le pasamos un objeto como parámetro.

Cuando los objetos se pasan como una referencia, estos serán modificados dentro y fuera de la función.

```
var sacha = {
        nombre: 'Sacha',
        apellido: 'Lifszyc',
        edad: 28
}

functioncumpleanos(persona) {
        persona.edad += 1
} // esta función modifica el objeto

sacha
// { nombre: 'Sacha', apellido: 'Lifszyc', edad: 29 }
```

Para solucionar esto se puede crear un objeto diferente. Esto lo podemos hacer colocando tres puntos antes del nombre del parámetro al declarar la función.

La siguiente función copia el objeto (en la línea ...persona) y genera uno nuevo:

```
function cumpleanosOtro(persona) {
        return {
                ...persona,
                edad: persona.edad + 1
        }
}

cumpleanosOtro(sacha)
// { nombre: 'Sacha', apellido: 'Lifszyc', edad: 29 }

sacha
// { nombre: 'Sacha', apellido: 'Lifszyc', edad: 28 }
```

11 - Comparaciones en JavaScript

Existen varias maneras de comparar variables u objetos dentro de javascript.

Existen cinco tipos de datos que son primitivos y es necesario comprender al momento de hacer comparaciones:

- `Boolean`
- `Null`
- `Undefined`
- `Number`
- `String`

Variables

En el primer ejemplo le asignamos a '*x*' un valor numérico y a '*y*' un string. Para poder compararlos debemos agregar dos signos de igual "==". Esto los *convierte al mismo tipo de valor* y permite que se puedan comparar.

```
var x = 4
var y = '4'

x == y
// true
```

Cuando realizamos operaciones es recomendable usar tres símbolos de igual (===). Esto permite que JavasScript no iguale las variables que son de distinto tipo.

```
                                    *Sacha recomienda usa siempre el triple igual,
pero hay controversia...
```

```
x === y
// false
```

Objetos

Al comparar objetos JS tiene en cuenta también el nombre del objeto, por lo tanto se remite a comparar el nombre de las variables a demás del valor de los atributos. Con objetos literales desglosados (*otroMas* en este caso), pasa lo mismo y la comparación da false ya que lo que se genera es un nuevo objeto a partir del desglosado.

```
var sacha = {
        nombre: 'Sacha'
```

```
}
var otro = {
        nombre: 'Sacha'
}
var otroMas = {
        ...sacha
}

sacha == otro
// false

sacha === otro
// false

sacha == otroMas
// false

sacha === otroMas
// false
```

Si asignamos el valor del objeto a una variable y los comparamos, el doble y el triple igual darán como resultado 'true' ya que en este caso las dos variables estarían refiriendo al mismo espacio en la memoria RAM.

```
var otroMasTodavia = sacha

sacha == otroMasTodavia
// True

sacha === otroMasTodavia
// True
```

Otra cosa a tener en cuenta es que si cambiamos el valor del atributo en la variable, automáticamente cambia el valor del objeto también, por el mismo motivo que los operadores dan 'true', ambos refieren al mismo espacio en la memoria RAM.

```
otroMasTodavia.nombre = "Pepe"

otroMasTodavia.nombre
// "Pepe"

sacha.nombre
// "Pepe"
```

Sección III - Estructuras de Control y Funciones

12 - Condicionales

Las estructuras de control nos permiten controlar el flujo de nuestro código. Algunas de ellas son:

• if...else • switch • while • for • for...in • try

Los condicionales (*if...else*) nos permiten decidir si un código se ejecuta o no. Mediante un condicional (if) decidiremos si se ejecuta una parte de nuestro código cuando se cumpla o no cierta condición.

```
var sacha = {
        nombre: 'Sacha',
        apellido: 'Lifszyc',
        edad: 28,
        ingeniero: true,
        cocinero: false,
        cantante: false,
        dj: false,
        guitarrista: false,
        drone: true
}

function imprimirProfesion(persona) {
        console.log(`${persona.nombre} ${persona.apellido} es: `)
        if(persona.ingeniero) {
                console.log('Ingeniero')
        } else {
                console.log('No es Ingeniero')
        }
        if(persona.cocinero) {
                console.log('Cocinero')
        }
        if(persona.cantante) {
                console.log('Cantante')
        }
        if(persona.dj) {
                console.log('dj')
        }
        if(persona.guitarrista) {
                console.log('Guitarrista')
        }
        if(persona.drone) {
                console.log('Piloto de Drone')
        }
}
```

13 - Funciones que retornan valores

Usamos condicionales para desglosar las funciones en funciones más pequeñas que retornen un valor.

Return detiene la ejecución de una función y devuelve el valor de esa función.

Las variables definidas con '*const*' se comportan como las variables '*var*', excepto que no pueden ser reasignadas. Las constantes pueden ser declaradas en mayúsculas o minúsculas. Pero por convención, para distinguirlas del resto de variables, se escribe todo en mayusculas.

```
var sacha = {
        nombre: 'Sacha',
        apellido: 'Lifszyc',
        edad: 28,
        ingeniero: true,
        cocinero: false,
        cantante: false,
        dj: false,
        guitarrista: false,
        drone: true
}

const MAYORIA_DE_EDAD = 18

functionesMayorDeEdad(persona) {
        return persona.edad >= MAYORIA_DE_EDAD
}

functionimprimirSiEsMayorDeEdad(persona) {
        if(esMayorDeEdad(persona)) {
                console.log(`${persona.nombre}${persona.apellido} es mayor de edad.
        } else {
                console.log(`${persona.nombre}${persona.apellido} no es mayor de ed
        }
}
```

14 - Arrow Functions

Loas Arrow Functions permiten una nomenclatura más corta para escribir expresiones de funciones.

Este tipo de funciones deben definirse antes de ser utilizadas.

Al escribir las Arrow Functions no es necesario escribir la palabra function, la palabra return, ni las llaves.

JS permite asignar una función a una variable. Se llama función anónima. Y se puede escribir de varias maneras:

```
const MAYORIA_DE_EDAD = 18

var esMayorDeEdad =function(persona){
        return persona.edad >= MAYORIA_DE_EDAD
}
```

Sacha prefiere declararla como *'const'* y no como *'var'* para definir que es una función y no una variable:

```
const esMayorDeEdad =function(persona){
        return persona.edad >= MAYORIA_DE_EDAD
}
```

La palabra clave *'function'* puede reemplazarse por un '=>' después de persona y se convierte en un *arrow function*:

```
const esMayorDeEdad = (persona) => {
        return persona.edad >= MAYORIA_DE_EDAD
}
```

Se pueden seguir quitando caracteres. Cuando hay un sólo parámetro se pueden quitar los paréntesis.

```
const esMayorDeEdad = persona => {
        return persona.edad >= MAYORIA_DE_EDAD
}
```

Si una función sólo retorna un valor se puede quitar el keyword *'return'* y las llaves '*{}*'.

```
const esMayorDeEdad = persona => persona.edad >= MAYORIA_DE_EDAD
```

También se puede desestructurar el parámetro ya que sólo nos interesa la edad. Hay que agregar paréntesis:

```
const esMayorDeEdad =  ({edad}) => persona >= MAYORIA_DE_EDAD
```

15 - Estructuras repetitivas: for ...

El bucle '*for*', se utiliza para repetir una o más instrucciones un determinado número de veces.

Para escribir un bucle '*for*' se coloca la palabra '*for*' seguida de paréntesis y llaves.

Ej.

```
for(){ }
```

Dentro de los paréntesis irán las condiciones para ejecutar el bucle, y dentro las llaves irán las instrucciones que se deben repetir.

Las condiciones del '*for*' son 3 comandos separados por ';'.

• El primero es la declaración del contador, desde dónde comienza a contar; 'let i = 1' en este caso. • El segundo es hasta dónde cuenta; 'i <= 365' en este caso. • El tercero es incrementar en 1 la variable contador; 'i++'.

En este ejemplo la variable i la utilizamos como contador.

```
for(let i=1; i <= 10; i++) {
        console.log(i)
}
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
// 10
```

En el siguiente ejemplo se imprime el peso de la persona (objeto) al iniciar el año y luego del for (365 días) se imprime el peso de esa misma persona después del año.

```
var sacha = {
        nombre: 'Sacha',
        apellido: 'Lifszyc',
        edad: 28,
        peso: 75
}

console.log(`Al inicio del año ${sacha.nombre} pesa ${sacha.peso}kg`);

const VARIACION_DE_PESO = 0.2
const aumentoDePeso = persona => persona.peso += VARIACION_DE_PESO
const bajaDePeso = persona => persona.peso -= VARIACION_DE_PESO
```

```
for (let i = 1; i <= 365; i++) {
        var random = Math.random();
        if(random < 0.25) {
                aumentoDePeso(sacha)
        } else if (random < 0.50) {
                bajaDePeso(sacha)
        }
}

console.log(`Al final del año ${sacha.nombre} pesa ${sacha.peso.toFixed(1)}kg`);
```

16 - Estructuras repetitivas: while ...

La estructura repetitiva '*while*' nos permite repetir un loop hasta que se cumple una condición; hasta que la condición entre paréntesis de como resultado 'true'.

```
while(sacha.peso > META) {
        do next...
}
```

A demás aprendimos a usar el keyword 'debugger' que se usa cuándo nuestro código falla o no se ejecuta. Cada vez que el código lea esta palabra detiene su ejecución.

En este caso lo insertamos dentro del while, en la primera línea; después, en la consola, en la pestaña 'sources' nos muestra dónde se detiene. Con los botones superiores lo vamos haciendo avanzar hasta que muestra el error.

```
while(sacha.peso > META) {
        debugger
        if(comeMucho()){
```

17 - Estructuras repetitivas: do-while ...

Otra estructura repetitiva es el '*do-while*'. A diferencia de la instrucción '*while*', un bucle '*do...while*' se ejecuta una vez antes de que se evalúe la expresión condicional.

El ciclo *'do-while'* va a ejecutar el o los comandos dentro de las llaves del *'do'* al menos una vez hasta que la función declarada dentro de los paréntesis del while se cumpla. El *"al menos una vez"* es porque el flujo de ejecución comienza con *'do'* (hacer) y luego verifica por primera vez la condición.

```
var contador = 0
const llueve = () => Math.random() < 0.25

do {
        contador++
} while(!llueve())

console.log(`Fui a ver si llueve ${contador} veces.`)
```

18 - Estructuras repetitivas: switch ...

Switch se utiliza para realizar diferentes acciones basadas en múltiples condiciones.

Prompt, muestra un cuadro de mensaje que le pide al usuario que ingrese algúna información.

Break, sirve para que el browser se salte un bucle.

En este ejemplo le vamos a devolver el horozcopo del día al usuario de acuerdo al signo que ingrese.

```
while (window) {
        let signo = prompt('Cuál es tu signo?')

        switch (signo) {
                case 'aries':
                        alert('Aries empezará un año con...')
                        break
                case 'tauro':
                        alert('Tanta carga de trabajo que Tauro... ')
                        break
                case 'géminis':
                case 'geminis':
                        alert('Durante el mes de enero...')
                        break
                case 'cáncer':
                case 'cancer':
                        alert('La situación económica de Cáncer podría tener...')
                        break
                case 'leo':
                        alert('A pesar de las prisas por terminar un proyecto...')
                        break
                case 'virgo':
```

```
                alert('Durante el mes de enero se presentan...')
                break
            case 'libra':
                alert('La situación económica de Libra... ')
                break
            case 'escorpio':
                alert('Enero, será un mes en el cual Escorpio deberá...')
                break
            case 'sagitario':
                alert('Sagitario empezará un poco tenso ... ')
                break
            case 'capricornio':
                alert('El fuerte carácter de Capricornio...')
                break
            case 'acuario':
                alert('Los astros se han alineado estos primeros días... ')
                break
            case 'piscis':
                alert('Después de tantas distracciones...')
                break
            default:
                alert('No es un signo válido.')
                break
        }
}
```

Sección IV - Arrays

19 - Introducción a los arrays

Los arrays son estructuras de datos que nos permiten organizar elementos dentro de una collección. Estos elementos pueden ser números, strings, booleanos, objetos, etc. Luego es posible realizar operaciones sobre esa colección.

Según la documentación de developer.mozilla.org; ...'es un objeto global que es usado en la construcción de arrays, que son objetos tipo lista de alto nivel'.

Para indicar que una variable es un array se usan los corchetes rectos:

```
var array = []
```

Y para mostrar el contenido:

```
var sacha = {
    nombre: 'Sacha',
```

```
    apellido: 'Lifszyc',
    altura: 1.72
}

var alan = {
    nombre: 'Alan',
    apellido: 'Perez',
    altura: 1.86
}

var martin = {
    nombre: 'Martin',
    apellido: 'Gomez',
    altura: 1.85
}

var dario = {
    nombre: 'Dario',
    apellido: 'Juarez',
    altura: 1.71
}

var vicky = {
    nombre: 'Vicky',
    apellido: 'Zapata',
    altura: 1.56
}

var paula = {
    nombre: 'Paula',
    apellido: 'Barros',
    altura: 1.76
}

var personas = [sacha,alan,martin,dario,vicky,paula];

personas[0]
//{nombre: "Sacha", apellido: "Lifszyc", altura: 1.72}

personas[1]
{nombre: "Alan", apellido: "Perez", altura: 1.86}

// El 0 y el 1 representan respectivamente el subíndice del elemento que se va a mo

personas[0].nombre
// "Sacha"

personas[3].altura
//1.71

//también se puede usar
personas[0].['nombre']
// "Sacha"

personas[3].['altura']
//1.71
```

También se puede recorrer un array con un for loop e ir imprimiendo las propiedades deseadas del mismo:

```
for(var i=0; i<personas.length; i++){
  var persona=personas[i];
  console.log(`${persona.nombre} mide ${persona.altura}.`);
}

// Sacha mide 1.72.
// Alan mide 1.86.
// Martin mide 1.85.
// Dario mide 1.71.
// Vicky mide 1.56.
// Paula mide 1.76.
```

20 - Filtrar un array

Para filtrar necesitamos dos cosas: una función y una condición. Usamos la función nativa de javascript 'filter()' que recibe una condición como parámetro.

En este ejemplo nuestra condición es que la estatura de las personas sea mayor de 1.80mts.

```
var sacha = {
    nombre: 'Sacha',
    apellido: 'Lifszyc',
    altura: 1.72
}

var alan = {
    nombre: 'Alan',
    apellido: 'Perez',
    altura: 1.86
}

var martin = {
    nombre: 'Martin',
    apellido: 'Gomez',
    altura: 1.85
}

var dario = {
    nombre: 'Dario',
    apellido: 'Juarez',
    altura: 1.71
}
```

```
var vicky = {
    nombre: 'Vicky',
    apellido: 'Zapata',
    altura: 1.56
}

var paula = {
    nombre: 'Paula',
    apellido: 'Barros',
    altura: 1.76
}

var personas = [sacha, alan, martin, dario, vicky, paula]

const esAlta = ({ altura }) => altura > 1.8
var personasAltas = personas.filter(esAlta)

console.log(personasAltas)

// También puede escribirse así:

var personasAltas = personas.filter(function(){
        return personas.altura > 1.8
})
```

El método '*filter( )*' crea una nueva matriz con todos los elementos que pasan la prueba implementada por la función proporcionada. Recuerda que si no hay elementos que pasen la prueba, '*filter*' devuelve un array vacío.

21 - Transformar un array

El método *map()* itera sobre los elementos de un array en el orden de inserción y devuelve array nuevo con los elementos modificados. *map()* siempre nos devuelve un nuevo array. En este ejemplo lo usamos para cambiar la unidad de medida de la altura.

```
var personas = [sacha, alan, martin, dario, vicky, paula]

const pasarAlturaACmts = persona => ({
        ...persona,
        altura: persona.altura * 100
})
var alturaEnCmts = personas.map(pasarAlturaACmts)
```

Otra manera que se mostró en la clase. Pero esta función modifica también el array ingresado, así que paso a ser la forma correcta la que está en el ejemplo anterior.

```
const pasarAlturaACmts = persona => {
        persona.altura *= 100// Es lo mismo que persona.altura = persona.altura * 1
        return persona
}
```

22 - Reducir un array a un valor

Función Reduce

Es otra de las funciones más usadas con arrays en JS. Reduce un array a un valor único.

Si lo que queremos es calcular la cantidad total de libros de las personas definidas en los siguientes objetos es posible usar un for e ir incrementando un acumulador. Pero se ajusta mucho más para tal fin el método .reduce()

```
var sacha = {
    nombre: 'Sacha',
    cantidadDeLibros: 111
}
var alan = {
    nombre: 'Alan',
    cantidadDeLibros: 78
}
var martin = {
    nombre: 'Martin',
    cantidadDeLibros: 132
}
var dario = {
    nombre: 'Dario',
    cantidadDeLibros: 90
}
var vicky = {
    nombre: 'Vicky',
    cantidadDeLibros: 91
}
var paula = {
    nombre: 'Paula',
    cantidadDeLibros: 182
}

var personas = [sacha, alan, martin, dario, vicky, paula]
```

Con un loop *for*...

```
var acum = 0

for (var i = 0; i < personas.length; i++) {
        acum = acum + personas[i].cantidadDeLibros
}

console.log(`Entotal todos tienen ${acum} libros.`)
```

Con el método .reduce():

```
const reducer = (acum, persona) => acum + persona.cantidadDeLibros
var totalDeLibros = personas.reduce(reducer, 0)
console.log(`Entotal todos tienen ${totalDeLibros} libros.`)
```

Sección V - Programación Orientada a Objetos en JavaScript (POO)

23 - Como funcionan las clases en JavaScript

En JavaScript hablar de *objetos* es más bien referirse a ***Prototipos*** y no tanto a *clases*. Si bien en las nuevas versiones de JavaScript existen las *clases* no son clases como tales, como las pordríamos conocer en cualquier otro lenguaje de programación; no existe la *herencia* como tal. Pero sí existen los ***Prototipos*** y vamos a ver que esas llamadas *clases* terminan siendo ***Prototipos***.

Qué son los *prototipos*:

> ..."comenzaremos diciendo que (en JavaScript) todos los objetos dependen de un prototipo y que ***los prototipos son objetos***, es más cualquier objeto puede ser un prototipo"... ..."un prototipo es un objeto del que otros objetos heredan propiedades. Los objetos siempre heredan propiedades de algún objeto anterior, de este modo solo el objeto original y primigenio de javascript es el único que no hereda de nadie...

```
**Referencia:** http://mialtoweb.es/prototipos-en-javascript/
```

Objetos => Prototipos

Crear un prototipo es muy similar a crear una variable:

• se antepone el keyword *function*; • la primer letra del nombre va en mayúscula; • para invocar un nuevo objeto a partir de este ***prototipo*** se usa el keyword *'new'*.

```
function Persona(){
        console.log('Hola, soy un nuevo objeto.')
}
var pablo = new Persona()
```

• se le pueden pasar parámetros; • para generar nuevos parámetros o atributos dentro de la declaración del objeto se usa el keyword 'this' • es implícito en javaScript el retornar el objeto que se está creando

```
function Persona(nombre, apellido){
        this.nombre = nombre
        this.apellido = apellido
}
var pablo = new Persona('Pablo', 'Andrés')
```

• es posible anexar funciones al prototipo usando el apéndice *.prototype* precedido de el nombre que le asignamos a nuestro nuevo ***prototipo*** y sucedido del nombre de nuestra nueva función encadenados. Luego este se iguala a una función anónima.

• se pueden usar los mismos atributos que en el objeto (*this.xxx*)

```
function Persona(nombre, apellido){
        this.nombre = nombre
        this.apellido = apellido
}

Persona.prototype.saludar = function(){
        console.log(`Hola me llamo ${this.nombre} ${this.apellido}`)
}

var pablo = new Persona('Pablo', 'Andrés')
var joaquin = new Persona('Joaquín', 'Perez')
var rosa = new Persona('Rosa', 'Mosqueta')

rosa.saludar()
// Hola me llamo Rosa Mosqueta
```

24 - Modificando un prototipo

El prototipo es un objeto de javascript, si lo modificamos en un cierto lugar del código, a una cierta altura, a partir de ahí va a quedar modificado. Por lo que una buena práctica, o más bien la manera de declarar un prototipo es al inicio del código y en bloque, es decir todo el código junto (propiedades y funciones del prototipo). De otra manera, si queremos correr una función antes de declararla JS arrojará un error y quedará interrumpido el flujo de ejecución.

Ej:

```
function Persona(nombre, apellido, altura) {
  this.nombre = nombre
  this.apellido = apellido
  this.altura = altura
}
Persona.prototype.saludar = () => {
  console.log(`Hola, me llamo ${this.nombre} ${this.apellido}`)
}

Persona.prototype.soyAlto = function() {
  return this.altura > 1.8
}

var sacha = new Persona('Sacha', 'Lifszyc', 1.72)
var erika = new Persona('Erika', 'Luna', 1.65)
var arturo = new Persona('Arturo', 'Martinez', 1.89)

sacha.soyAlto()
// false

erika.soyAlto()
// false

arturo.soyAlto()
// true
```

Si declaro la función al final del código, después de llamar a la función; retorna un error *'Uncaught TypeError: sacha.soyAlto is not a function'*

```
function Persona(nombre, apellido, altura) {
  this.nombre = nombre
  this.apellido = apellido
  this.altura = altura
}
Persona.prototype.saludar = () => {
  console.log(`Hola, me llamo ${this.nombre} ${this.apellido}`)
}
//Persona.prototype.soyAlto = () => this.altura > 1.8
var sacha = new Persona('Sacha', 'Lifszyc', 1.72)
var erika = new Persona('Erika', 'Luna', 1.65)
var arturo = new Persona('Arturo', 'Martinez', 1.89)

sacha.soyAlto()
```

```
erika.soyAlto()
arturo.soyAlto()

Persona.prototype.soyAlto = function() {
  return this.altura > 1.8
}
// Uncaught TypeError: sacha.soyAlto is not a function
```

Si convierto la función en '*arrow function*' también tendré conflictos pero esta vez con el '*this*'; siempre obtendremos *false* como respuesta a la función ya que la propiedad '*this.altura*' es '*undefined*'.

```
function Persona(nombre, apellido, altura) {
  this.nombre = nombre
  this.apellido = apellido
  this.altura = altura
}
Persona.prototype.saludar = () => {
  console.log(`Hola, me llamo ${this.nombre} ${this.apellido}`)
}
Persona.prototype.soyAlto = () => this.altura > 1.8

var sacha = new Persona('Sacha', 'Lifszyc', 1.72)
var erika = new Persona('Erika', 'Luna', 1.65)
var arturo = new Persona('Arturo', 'Martinez', 1.89)

sacha.soyAlto()
// false

erika.soyAlto()
// false

arturo.soyAlto()
// false
```

25 - El contexto de las funciones: Quién es *this*?

Al cambiar una función por un arrow function en el código nos comenzó a arrojar valores 'undefined'. La propiedad '*this.altura*' es '*undefined*' debido a que '*this*' no refiere al prototipo que nosotros creamos.

Al debbuguear nos muestra que *this* === *window*.

• ***'this' en el espacio global refiere al objeto 'window***'; 'this' **es** el objeto 'window', ***el mismo campo en memoria***.

• Los arrow function refieren siempre al *'this'* del *'window'* **y no del prototipo.**

26 - La verdad oculta sobre las clases en JavaScript

Cómo hago para que un prototipo herede de otro?

> ..."JS no soporta la herencia porque no soporta las clases, no hay clases, hay prototipos a los que les vamos agregando métodos que reciben funciones, saben quien es 'this' y saben como ejecutarlas. Pero no existe un sistema como tal donde yo diga: _ "este prototipo va a heredar de otro"... Lo que sí existe es la 'herencia prototipal'

Cómo funciona? Es posible crear *'prototipoHijo'* que van a ser un subtipo del *'prototipoPadre'* (*persona* en el ejemplo que venimos usando), podemos crear por ejemplo un sub-tipo de *'persona'*, un 'desarrollador'.

Este *'prototiposHijo'*, cada vez que sea requerido buscará los métodos en sí mismo, luego si no los encuentra, buscará en su *'prototipoPadre'*, *'prototipoAbuelo'* ... y así hasta llegar al *'prototipo'* base de todos los objetos que en JavaScript es *'object'*. Si *'object'* no conoce ese mensaje, recién ahí es que javaScript lanzará el error de que ese método no puede ejecutarse.

Para crear nuestro 'desarrollador' **generamos este *'prototipoHijo'* inmediatamente después del código del *'prototipoPadre'***, *'Persona'* en este caso. Y agregamos un método a este *'prototiposHijo'* para que devuelva un saludo particular de desarrollador.

```
function Desarrollador(nombre, apellido){
        this.nombre = nombre
        this.apellido = apellido
}
Desarrollador.prototype.saludar = function(){
        console.log(`Hola, me llamo ${this.nombre}${this.apellido} y soy desarrolla
}
```

Pero cómo hacemos para que este *'prototipoHijo'* herede los métodos del *'prototipoPadre'*?

Para esto tenemos que generar una nueva función que le diga al *'prototipoHijo'* quién va a ser su *'prototipoPadre'* y la llamamos inmediatamente después de la declaración del *'prototipoHijo'* y antes de declarar las funciones de este mismo, ya que si queremos "pisar" algún método del *'prototipoPadre'*, esta llamada va a volver a sentenciar los métodos del *'prototipoPadre'*.

(Hasta las regulaciones del ECMAScript esta era la única manera de generar herencia en JS)

```
// función enlace
function heredaDe(prototipoHijo, prototipoPadre) {
  ...
}

// función padre
function Persona(nombre, apellido, altura) {
  ...
}
Persona.prototype.saludar = function () {
  ...
}
Persona.prototype.soyAlto = function () {
  ...
}

// función hijo
function Desarrollador(nombre, apellido) {
  this.nombre = nombre
  this.apellido = apellido
}

// invocación de función enlace
heredaDe(Desarrollador, Persona)

//método de función hijo
Desarrollador.prototype.saludar = function () {
  console.log(`Hola, me llamo ${this.nombre} ${this.apellido} y soy desarrollador/a
}
```

La funcionalidad de la función que va a generar el enlace de herencia es la siguiente.

• Primero generamos una variable a la que le cargamos una función vacía:

```
function heredaDe(prototipoHijo, prototipoPadre) {
    var fn = function () {}
}
```

• Luego asignamos el prototipo del '*prototipoPadre*' a esta misma:

```
function heredaDe(prototipoHijo, prototipoPadre) {
    var fn = function () {}
    fn.prototype = prototipoPadre.prototype
}
```

• Y entonces ahora asignamos el '*prototipoPadre*' al prototipo del *'prototipoHijo'*. Lo hacemos de esta manera así, al estar generando un clon del '*prototipoPadre*' no "pisamos" o modificamos nada en este.

```
function heredaDe(prototipoHijo, prototipoPadre) {
    var fn = function () {}
    fn.prototype = prototipoPadre.prototype
    prototipoHijo.prototype = new fn
}
```

• Y finalmente llamamos al '*constructor*' del mismo *'prototipoHijo'* para que se refiera a sí mismo y no al constructor del '*prototipoPadre*'

```
function heredaDe(prototipoHijo, prototipoPadre) {
  var fn = function () {}
  fn.prototype = prototipoPadre.prototype
  prototipoHijo.prototype = new fn
  prototipoHijo.prototype.constructor = prototipoHijo
}
```

Este es un método muy complejo, ya obsoleto. Hoy por hoy lo hacemos de otra manera.

El código completo queda así:

```
function heredaDe(prototipoHijo, prototipoPadre) {
  var fn = function () {}
  fn.prototype = prototipoPadre.prototype
  prototipoHijo.prototype = new fn
  prototipoHijo.prototype.constructor = prototipoHijo
}

function Persona(nombre, apellido, altura) {
  this.nombre = nombre
  this.apellido = apellido
  this.altura = altura
}

Persona.prototype.saludar = function () {
  console.log(`Hola, me llamo ${this.nombre} ${this.apellido}`)
}

Persona.prototype.soyAlto = function () {
  return this.altura > 1.8
}

function Desarrollador(nombre, apellido) {
  this.nombre = nombre
  this.apellido = apellido
}

heredaDe(Desarrollador, Persona)

Desarrollador.prototype.saludar = function () {
  console.log(`Hola, me llamo ${this.nombre} ${this.apellido} y soy desarrollador/a
}
```

Creamos unos objetos y analizamos como se comportan:

```
var sacha = new Persona('Sacha', 'Lifszyc', 1.72)
var erika = new Persona('Erika', 'Luna', 1.65)
var arturo = new Desarrollador('Arturo', 'Martinez', 1.89)

sacha.saludar()
// Hola, me llamo Sacha Lifszyc

erika.saludar()
// Hola, me llamo Erika Luna

arturo.saludar()
// Hola me llamo Arturo Martinez y soy desarrollador/a.
// este nos arroja el saludo de _Desarrollador_ ya que es una instancia del mismo
```

Ahora analicemos al atributo *.prototipe* en consola:

```
Persona.prototype
// {saludar: ƒ, soyAlto: ƒ, constructor: ƒ}
//      saludar: ƒ ()
//      soyAlto: ƒ ()
//      constructor: ƒ Persona(nombre, apellido, altura)
//      __proto__: Object
```

Vemos que *.prototype* es un atributo que tiene todas las funciones de Persona. Es un **objeto** que tiene métodos y su propio *constructor* a demás de un atributo **proto** que es el que hace referencia al '*prototipoPadre*'; en este caso '*Object*'

```
__proto__: Object
```

Veamos a quién hace referencia el *'prototipoHijo'*

```
Desarrollador.prototype
//Persona {constructor: ƒ, saludar: ƒ}
//      constructor: ƒ Desarrollador(nombre, apellido)
//      saludar: ƒ ()
//      __proto__:
//              saludar: ƒ ()
//              soyAlto: ƒ ()
//              constructor: ƒ Persona(nombre, apellido, altura)
//              __proto__: Object
```

Y aquí vemos que hace referencia al prototipo *Persona* aunque en principio nos muestra su propio '*constructor*' y su propio método '*saludar*'. Dentro del '*proto*' nos muestra los métodos del '*prototipoPadre*' y a su vez el '*proto*' de este hace referencia a su '*prototipoPadre*' que es nuevamente '*Object*' Es desde este encadenamiento que es posible que hagamos la invocación de los métodos del '*prototipoPadre*' en el *'prototipoHijo'*.

27 - Clases en JavaScript

Como ya se mencionó en esta referencia, a partir del standar de código de JavaScript, ECMAScript 2015 y sus subsiguientes versiones se modificó la forma vista de generar herencia prototipal. Estos standars no agregaron funcionalidad nueva a este tema pero sí lo ha facilitado y sintetizado mucho.

> Dato extra: "***sugar syntax***; este es un término usado para describir una característica en un lenguaje que te permite hacer algo más facilmente con menos escritura o código, pero en realidad no agrega ninguna funcionalidad nueva al lenguaje".

Definimos la variable persona de acuerdo a ECMA-Script 2015 entonces:

```
class Persona {
        constructor(nombre, apellido, altura, genero){
                this.nombre = nombre
                this.apellido = apellido
                this.altura = altura
                this.genero = genero
        }
        saludar(){
                console.log(`Hola me llamo ${this.nombre}${this.apellido}`)
        }
        soyAltX(){
                var altX = this.genero == 'masculino' ? 'alto' : 'alta'
                var string = this.altura >= 1.8         ? `Soy ${this.nombre}${this
                                                        : `Soy ${this.nombre}${this.apellid
                console.log(string)
        }
}
```

Vemos que se utiliza el keyword *class* para definir el prototipo (aunque se llame 'class' sigue siendo un prototipo). Y que agregamos una definición de *constructor* como la función que recibe los parámetros. También podemos, a continuación de este y dentro del contexto de la clase y no por fuera definir los métodos que esta *clase* va a utilizar.

Cómo logramos que una clase herede de otra?

Simplemente definimos una nueva clase y agregamos a esta el keyword *extends* seguido de la clase de la que va a Heredar. A demás, en el constructor de la misma debemos invocar los parámetros de la función de la que hereda con el keyword *super*.

```
class Desarrollador extends Persona{
        constructor(nombre, apellido, altura, genero){
                super(nombre, apellido, altura, genero) // super llama los atributo
        }
        saludar(){
                console.log(`Hola, me llamo ${this.nombre} ${this.apellido} y soy d
        }
}
```

Notemos que *Persona.prototype* en consola seguirá arrojando el objeto *prototype* exactamente igual que con la sintaxis referida exclusivamente a *prototype*.

Sección VI - Asincronismo

28 - Funciones como parámetros

Antes de comenzar a hablar de asincronismo tenemos que comprender la mecánica de las funciones como parámetro. Es posible pasar funciones como parámetro, como si fuera cualquier otro tipo de variable.

Cómo se declara? Como cualquier función.

La siguiente función es una respuesta a la función saludar del código que traemos de las clases anteriores:

```
function responderSaludo(){
        console.log(`Buen día`)
}
```

Cómo se implementa al prototipo? Se agrega un parámetro a la función que la va a disparar dentro del prototipo ('saludar(fn)', en este caso) y un if que evalúe si es llamada.

```
saludar(fn){
    console.log(`Hola me llamo ${this.nombre} ${this.apellido}`)
    if(fn){
        fn()
    }
}
```

Cómo se ejecuta? Se ejecuta invocando la función sin paréntesis dentro de los paréntesis de la función saludar ya que es una respuesta al saludo:

```
sacha.saludar(responderSaludo)
```

Si quiero pasar parámetros?

Los agrego al declarar la función y luego los agrego entre paréntesis a la invocación de la misma dentro de la función que la dispara en el prototipo. Funcionan implícitamente, es decir, no se agregan cuando invoco la función dentro del 'saludar()'.

En la declaración:

```
function responderSaludo(nombre, apellido, esDev){
        console.log(`Buen día ${nombre} ${apellido}.`)
        if (esDev) {
                console.log(`Ah mirá, no sabía que eras dev.`)
        }
}
```

En la función nativa:

```
saludar(fn){
        console.log(`Hola me llamo ${this.nombre} ${this.apellido}`)
        if(fn){
                fn(this.nombre, this.apellido)
        }
}
```

Se invoca:

```
sacha.saludar(responderSaludo)
```

Para despejar la lectura del código y escribir menos:

```
saludar(fn){
        var {nombre, apellido} = this
        console.log(`Hola me llamo ${nombre}${apellido}`)
        if(fn){
                fn(nombre, apellido)
```

```
        }
}
```

Hay valores que al ser evaluados dentro de un if dan verdadero y otros falso, en este caso fn dentro del if evalúa si existe la función en la invocación de saludar().

El código completo quedaría así:

```
class Persona{
        constructor(nombre, apellido, altura, genero){
                this.nombre = nombre
                this.apellido = apellido
                this.altura = altura
                this.genero = genero
        }
        saludar(fn){
                var {nombre, apellido} = this
                console.log(`Hola me llamo ${nombre} ${apellido}`)
                if(fn){
                        fn(nombre, apellido)
                }
        }
        soyAltX(){
                var altX = this.genero == 'masculino' ? 'alto' : 'alta'
                var string = this.altura >= 1.8 ? `Soy ${this.nombre} ${this.apelli
                                                : `Soy ${this.nombre} ${this.apelli
                console.log(string)
        }
}

class Desarrollador extends Persona{
        constructor(nombre, apellido, altura){
                super(nombre, apellido, altura)
        }
        saludar(fn){
                var {nombre, apellido} = this
                console.log(`Hola, me llamo ${this.nombre} ${this.apellido} y soy d
                if(fn){
                        fn(nombre, apellido, true)
                }
        }
}

function responderSaludo(nombre, apellido, esDev){
        console.log(`Buen día ${nombre} ${apellido}.`)
        if (esDev) {
                console.log(`Ah mirá, no sabía que eras dev.`)
        }
}

var pablo = new Persona('Pablo', 'Andrés', 1.78, 'masculino')
var joaquin = new Desarrollador('Joaquín', 'Perez', 1.91, 'masculino')
var rosa = new Persona('Rosa', 'Mosqueta', 1.81, 'femenino')
```

```
var elis = new Persona('Elis', 'Detta', 1.73, 'femenino')

pablo.saludar()
joaquin.saludar(responderSaludo)
rosa.saludar(responderSaludo)
elis.saludar(responderSaludo)
```

29 - Cómo funciona el asincronismo en JavaScript

Transcripción del guión del video... No tan exacta.

Asincronismo

Javascript sólo puede hacer una cosa a la vez ... pero puede delegar la ejecución de ciertas funciones a otros procesos. Este modelo de concurrencia se llama EventLoop.

JavaScript tiene algo llamado pila de ejecución o callStack donde va poniendo las llamadas a las funciones según el orden de ejecución de nuestro programa; si una función llama a otra, entonces esta se agrega a la pila. Cuando termina de ejecutar una función la saca de la pila y la desecha.

En algún momento dado nuestro programa necesita algún dato de otro sitio en la web y le pide al navegador que cuándo obtenga los datos ejecute cierta función. Esta tarea que se lleva el navegador se llama 'callBack'. Mientras tanto js sigue ejecutando nuestro programa principal y cuando la respuesta llega va a parar a la 'cola de tareas'. Aquí las tareas se ordenan una detrás de la otra a medida de que van llegando.

Qué tareas van a parar a esta cola? • las peticiones a servidores • las interacciones visuales • la navegación clayInside (dice esto?) • los eventos que se realizan cada cierto tiempo

Recién cuando el programa se queda sin funciones en la pila de ejecución es que va a ir a buscar las funciones en la 'cola de tareas'; por eso es que hay que tener cuidado de no generar un 'cuello de botella' en la pila de ejecución.

Si javaScript se queda ejecutando tareas muy pesadas las funciones de la 'cola de tareas' van a tardar mucho tiempo en ejecutarse.

Por eso recuerda estas palabras y repitelas todas las noche antes de irte a dormir: No Voy A Bloquear el EventLoop!

30 - Cómo funciona el tiempo en JavaScript

Tiempos y prioridades de ejecución en JS:

Dados los siguientes console.logs, 'a', 'b' y 'c', los resultados son en orden e inmediatos; estarán en orden en el EventLoop.

```
console.log(`a`)
console.log(`b`)
console.log(`c`)

// a
// b
// c
```

Si genero un callBack a partir de un setTimeout(), el browser toma la petición y después del tiempo estipulado se dispara el console.log.

```
console.log(`a`)
setTimeoun(()=> console.log(`b`), 2000)
console.log(`c`)

// a
// c

// b [2 segundos después]
```

Si genero otro setTimeout, pero con un tiempo = 0, el callBack será generado de todas maneras por la función setTimeout y el console.log será disparado al finalizar el EventLoop de nuestro programa, cuando está listo para revisar la cola de tareas.

```
console.log(`a`)
setTimeout(()=> console.log(`b`), 0)
console.log(`c`)

// a
// c

// b [después de finalizado el EventLoop]
```

Esto puede verse más claramente si generamos un loop for que demore un tiempo evidente y mayor al seteado en el callBack. El tiempo configurado para el setTimeout es de 2000 milisegundos, como mínimo 2000 milisegundos, ya que el console.log funcionará después de que el for termine su proceso.

```
setTimeout(() =>console.log(`b`), 2000)
for(var i = 0; i < 10000000000; i++){}
```

31 - Callbacks

Qué son? Cómo son? Cómo los utilizo?

Utilizaremos una librería externa, *jQuery*, con un fin específico que es el de realizar un request y obtener datos de una API externa. Utilizamos la versión CDN de JQuery. *Nota: un CDN es un Content Delivery Nerwork. Un servidor en el planeta que nos va a conectar con la versión de jQuery más cercana a nuestra locación.*

La API que usaremos es la de Star Wars. Implementamos la llamada a la librería jquery.minified en el html antes de llamar nuestro archivo de funciones. Este lo usaremos para hacer requests a la api de 'swapi.co'.

Realizamos el request. En este caso:

```
const API_URL = 'https://swapi.co/api/'
const PEOPLE_URL = 'people/:id'

const URL = `${API_URL}${PEOPLE_URL)}`

$.get(URL, {crossDomain: true}, function(){ })
```

Donde:

- API_URL es la URL de la api
- PEOPLE_URL es un folder interno de la API, people/ en este caso y '*:id*' se escribe para después hacer un .replace() por cada id
- $.get es el método de jQuery para realizar el callback. (*Referencia http://api.jquery.com/jQuery.get/*)

Los parámetros del $.get son 3 separados por ',':

- URL: es el URL completo, en este caso tenemos que encadenar API_URL con PEOPLE_URL y a esta segunda remmplazarle el :id por el id de cada personaje.

  const URL = `${API_URL}${PEOPLE_URL.replace(':id', 1)}` // 1 es id de Luke

- Este segundo parámetro es un objeto que le indica al método si el callback es local o remoto.

  const opts = {crossDomain: true}

- Y por último el callback. Es una función que será invocada en algún futuro por el método $.get cuando termine de establecer la conexión remota, el request a la URL. Es una función anónima nativa de jQuery. *Referencia: http://api.jquery.com/jQuery.get/*

jQuery.get(url [, data] [, success] [, dataType])

Los parámetros, a parte de la referencia de jQuery podemos verlos haciendo un console.log de *arguments*:

```
$.get( URL, opts, function(){
        console.log(arguments)
})
```

arguments es una variable que nos va a dar un array con los parámetros que recibe la función. En este caso:

```
Arguments(3) [{…}, "success", {…}, callee: ƒ, Symbol(Symbol.iterator): ƒ]
0: {name: "Luke Skywalker", height: "172", mass: "77", hair_color: "blond", skin_color:
"fair", …}
1: "success"
2: {readyState: 4, getResponseHeader: ƒ, getAllResponseHeaders: ƒ, setRequestHeader: ƒ,
overrideMimeType: ƒ, …}
callee: ƒ (persona)
length: 3
Symbol(Symbol.iterator): ƒ values()
__proto__: Object
```

Vemos que el parámetro que nos devuelve los resultados de la API es el primero [,data]. Entonces el callback quedaría así:

```
$.get( URL, opts, function(data){
        console.log(data.name)
})
// Luke Skywalker
```

Donde 'data' puede ser reemplazado por el argumento que querramos

```
$.get( URL, opts, function(personaje){
        console.log(personaje.name)
})
// Luke Skywalker
```

Por último generamos una constante a partir de la función:

```
const API_URL = 'https://swapi.co/api/'
const PEOPLE_URL = 'people/:id'

const URL = `${API_URL}${PEOPLE_URL.replace(':id', 1)}`
```

```
const opts = { crossDomain: true }
const onPeopleResponse = function (personaje) {
        console.log(personaje.name)
}

$.get(URL, opts, onPeopleResponse)
```

32 - Haciendo múltiples requests

Requests en Paralelo

Creamos una nueva función y modificamos levemente el código para hacer el callback ingresando solamente el id:

```
const API_URL = 'https://swapi.co/api/'
const PEOPLE_URL = 'people/:id'

const opts = { crossDomain: true}
const onPeopleResponse = function(person){
        console.log(person.name)
}

function obtenerPersonaje(id){
        consturl = `${API_URL}${PEOPLE_URL.replace(':id', id)}`
        $.get(url, opts, onPeopleResponse)
}
```

Dado este código. En qué orden nos llegarán las respuestas a varios request al mismo tiempo?

```
obtenerPersonaje(1)
obtenerPersonaje(2)
obtenerPersonaje(3)

//  3
//  2
//  1
```

En este request el resultado llegó en el orden inverso en el que los pedimos.

Por qué sucede esto?

Por el asincronismo de JS. No sabemos en qué orden nos llegarán las respuestas, esto depende del servidor y de cada uno de los requests. Iniciamos los requests en un determinado orden pero no sabemos en qué orden van a llegar.

33 - Manejando el Orden y el Asincronismo en JavaScript

Una manera de asegurar que se respete la secuencia en que hemos realizado múltiples tareas es utilizando callbacks, con lo que se ejecutará luego, en cada llamada. Lo importante es que el llamado al callback se haga a través de una función anónima.

Para esto, agregamos primero otro parámetro a la función obtenerPersonaje(), verificamos si existe con un if y lo ejecutamos si así es. También reemplazamos la constante en el tercer parámetro del $.get por la función.

```
function obtenerPersonaje(id, callback) {
        consturl = `${API_URL}${PEOPLE_URL.replace(':id', id)}`
        $.get(url, opts, function(people){
                console.log(people.name)
        })
        if(callback) {
                callback()
        }
}
```

Así podemos invocar la función del callback de la siguiente manera:

```
obtenerPersonaje(1, function(){
        obtenerPersonaje(2, function(){
                obtenerPersonaje(3, function(){
                        obtenerPersonaje(4, function(){
                                obtenerPersonaje(5, function(){
                                        obtenerPersonaje(6, function(){
                                                obtenerPersonaje(7)
                                        })
                                })
                        })
                })
        })
})
```

Pero esto, como podemos ya ver, trae la problemática del anidamiento infinito llamado CallbackHell.

34 - Manejo de errores con callbacks

Cómo solucionar o prever el que el programa se quede sin conexión u algo parecido?

Primero modificamos la función y quitamos el if:

```
function obtenerPersonaje(id, callback) {
        consturl = `${API_URL}${PEOPLE_URL.replace(':id', id)}`
        $.get(url, opts, callback)
}
```

El callback ahora lo llamaremos desde la invocación de la función:

```
obtenerPersonaje(1, function(person){
        console.log(person.name)
})
```

A demás del método get() podemos encadenar otro llamado al método fail() que va a recibir un callback y se va a disparar si hay algún error.

```
$.get(url, opts, callback).fail(() => { console.log(`EROR! La conexiónse ha interrumpido y
```

Probamos en consola, pestaña 'Network' deshabilitamos la cache y después de recargar nos pusimos en modo offline y se reprodujo el error para disparar el fail().

El código completo queda así:

```
const API_URL = 'https://swapi.co/api/'
const PEOPLE_URL = 'people/:id'
const opts = { crossDomain: true}

functionobtenerPersonaje(id, callback) {
        consturl = `${API_URL}${PEOPLE_URL.replace(':id', id)}`
        $
                .get(url, opts, callback)
                .fail(() => { console.log(`ERORRRRRRR!!!!!!!!!!! No se pudo obtener
}

obtenerPersonaje(1, function(character) {
        console.log(character.name)

        obtenerPersonaje(2, function(character) {
                console.log(character.name)

                obtenerPersonaje(3, function(character) {
                        console.log(character.name)

                        obtenerPersonaje(4, function(character){
                                console.log(character.name)

                                obtenerPersonaje(5, function(character){
```

```
                                                console.log(character.name)

                                                obtenerPersonaje(6, function(character){
                                                        console.log(character.name)

                                                        obtenerPersonaje(7, function(charac
                                                                console.log(character.name)
                                                        } )
                                                } )
                                        } )
                                } )
                        })
                })
        })
```

35 - Promesas

Con los callBacks teníamos un problema al anidarlos. Para este problema existen las 'promesas'.

Antes era necesario usar librerías externas pero ahora la mayoría de los browsers soportan las promesas. Si queremos verificar si las promesas son soportadas por el usuario se podría usar lo que se llama un 'polifield'. Este detecta si el navegador donde está corriendo nuestro código no soporta las promesas, y si así es, crea las clases de las promesas por nosotros y así podrían ser utilizadas por nosotros de manera transparente para nuestro código.

Qué son las promesas? Tenemos que pensar las promesas como valores que aún no conocemos. Es la promesa de que ahí va a haber un valor cuando una acción asíncrona suceda y se devuelva.

Las promesas tienen 3 estados y son como cualquier otro objeto de javascript.

El primero de los estados es 'pending'. Es el estado cuando las creamos. Si se resuelve exitosamente pasa al estado 'fulfilled'. Si ocurre algún error y no se resuelve pasa al estado de 'rejected'.

Las promesas pueden no ser asíncronas también.

Para obtener el valor de la resolución de la promesa llamamos a la función _.then(val =>) _a la que le vamos a pasar como parámetro otra función en la que el primer parámetro será el valor que esperábamos.

Si sucede algún error agregamos el método .catch(err=>) al que se le asigna una función también como parámetro que va a recibir el error.

Las promesas se declaran de la siguiente manera:

```
new Promise( function( resolve, reject ) {
        ...
}).then( valor => {
        ...
}).catch( err => {
        ...
})
```

Se crea el nuevo objeto y se le asigna una función con dos parámetros 'resolve' y 'reject'. Estas son dos funciones que debemos llamar si la promesa se resuelve o no. Si se resuelve exitosamente llamamos a '.then(valor =>' para obtener el valor del promise dentro del arrow function (valor=>). Si sucede algún error podemos llamar al '.catch(err =>' para obtener el tipo de error que sucedió y actuar en consecuencia.

Otra cosa más a cerca de las promesas es que luego de llegar al estado de 'fulfilled' podemos retornar otra promesa dentro del .then y de esa manera ir encadenándolas en sucesivas acciones asíncronas. Cada una de ellas puede ser resuelta o rechazada en una nueva promesa que terminará en el estado de 'fulfilled'.

Entonces en nuestro código borramos las invocaciones anidadas y volvemos a modificar obtenerPersonaje().

function obtenerPersonaje() ya no recibirá un callback, directamente va a retornar una promesa.

```
function obtenerPersonaje(id) {
        return new Promise( function(resolve, reject){
                ...
        })
}
```

Como arrow function:

```
function obtenerPersonaje(id) {
        return new Promise((resolve, reject) => {
                ...  //Aquí dentro se genera el llamado asíncrono   ...
        })
}
```

Dentro de esta función se va a generar el llamado asíncrono. Devolvemos la generación de url y el $.get con el parámetro 'callback' reemplazado por una nueva función a modo de callback que se va a ejecutar recién cuando el get haya sido exitoso resolviendo la promesa. Por lo que le pasamos el parámetro 'data', a través del cual van a llegar los valores de nuestro personaje, y dentro de la función invocamos, a su vez, a la función resolve. También vamos a vover a insertar el método .fail() invocando el parámetro/función 'reject' con parámetro 'id'.

```
function obtenerPersonaje(id) {
        return new Promise((resolve, reject) => {
```

```
                consturl = `${API_URL}${PEOPLE_URL.replace(':id', id)}`
                $
                        .get(url, opts, function(data){
                                resolve(data)
                        })
                        .fail(() => reject(id))
        })
}
```

La función se invocaría entonces solamente con el parámetro id.

```
obtenerPersonaje(id)
```

La forma de obtener el valor es llamando al .then() con su respectiva función como parámetro que va a estar trayendo a nuestro personaje a través del parámetro 'data' que está en la función invocada en los parámetros del .get si este es exitoso.

```
obtenerPersonaje(1)
.then(function(personaje){
        console.log(personaje.name)
})
```

Y si sucede algún error en nuestro callback lo vamos a obtener con el método .catch() que va a recibir el id que viene a través del .fail() de la función.

```
function onError(id){
        console.log(`ERORRRRRRR!!!!!!!!!!! No se pudo obtener el personaje con id =
}

obtenerPersonaje(1)
        .then(function(personaje){
                console.log(personaje.name)
        })
        .catch(function(id){
                onError(id)
        })
```

O directamente invocamos la función desde el parámetro del .catch:

```
function onError(id){
        console.log(`ERORRRRRRR!!!!!!!!!!! No se pudo obtener el personaje con id =
}

obtenerPersonaje(1)
        .then(function(personaje){
                console.log(personaje.name)
```

```
        })
        .catch(onError)
```

Código final completo:

```
const API_URL = 'https://swapi.co/api/'
const PEOPLE_URL = 'people/:id'
const opts = { crossDomain: true}

function obtenerPersonaje(id) {
        return new Promise((resolve, reject) => {
                consturl = `${API_URL}${PEOPLE_URL.replace(':id', id)}`
                $
                        .get(url, opts, function(data){
                                resolve(data)
                        })
                        .fail(() => reject(id))
        })
}

function onError(id){
        console.log(`ERORRRRRRR!!!!!!!!!!! No se pudo obtener el personaje con id =
}

obtenerPersonaje(1)
        .then(function(personaje){
                console.log(personaje.name)
        })
        .catch(onError)
```

36 - Promesas Encadenadas

Encadenar promesas es mucho más limpio que con el método anterior. Primero escribimos la invocación de la promesa con un arrow function:

```
obtenerPersonaje(1)
    .then( personaje => {
        console.log(personaje.name)
    })
    .catch(onError)
```

Al resolver esta promesa vamos a retornar otra promesa invocando dentro del .then nuevamente la función obtenerPersona() con el id del siguiente personaje:

```
obtenerPersonaje(1)
    .then( personaje => {
        console.log(personaje.name)
        return obtenerPersona(2)
    })
    .catch(onError)
```

Y para obtener los valores de esta promesa encadenamos otro .then y copiamos la función parámetro cambiando el valor del id.

```
obtenerPersonaje(1)
    .then( personaje1 => {
        console.log(personaje1.name)
        return obtenerPersona(2)
    })
    .then( personaje2 => {
        console.log(personaje2.name)
        return obtenerPersona(3)
    })
    .then( personaje3 => {
        console.log(personaje3.name)
        return obtenerPersona(4)
    })
    .
    .
    .
    .catch(onError)
```

Ahora es mucho más legible y si cualquiera de estas promesas da un error funciona el mismo .catch para todos.

37 - Múltiples promesas en paralelo

Con promises podemos hacer los requests en paralelo sin alterar el orden de los objetos, lo que mejoraría mucho nuestro código y performance.

Generamos un array con los ids de los personajes que queremos obtener. Y a partir de este vamos a generar otro array con múltiples promesas, donde cada elemento sea una promesa, la promesa de obtener un personaje con su id. Con el método map() vamos a recorrer el array ids y por cada elemento de este vamos a generar uno nuevo que va a ser una promesa. Estas promesas las guardamos en una variable 'promesas'. A partir de cada objeto del array ids (de cada id) obtenemos una nueva promesa con la función _obtenerPersonaje(id).

```
var ids = [1, 2, 3, 4, 5, 6, 7]
var promesas = ids.map(function(id){
    return obtenerPersonaje(id)
})
```

Expresado en arrow function

```
var ids = [1, 2, 3, 4, 5, 6, 7]
var promesas = ids.map( id => obtenerPersonaje(id) )

// (7) [Promise, Promise, Promise, Promise, Promise, Promise, Promise]
```

Cómo obtenemos los valores de estas promesas cuando se resuelvan?

Para esto podemos llamar a un método que tiene la clase de promesas llamado 'Promise.all()' A este le pasamos el array 'promesas', le encadenamos el .then()que nos entrega los objetos y depués encadenamos el .catch() que se va a ejecutar si cualquiera de las promesas que tenemos en el array falla.

```
Promise
    .all(promesas)
    .then( personajes => console.log(personajes))
    .catch(onError)

// (7) [{...}, {...}, {...}, {...}, {...}, {...}, {...}]
//Si lo desglosamos tenemos en orden las respuestas de cada una de las promesas.
```

Las promesas tienen un gran potencial por sobre los callBakc. El código queda mucho más prolijo y a demás podemos realizar promesas en paralelo.

Código completo:

```
const API_URL = 'https://swapi.co/api/'
const PEOPLE_URL = 'people/:id'
const opts = { crossDomain: true}

functionobtenerPersonaje(id) {
    returnnew Promise((resolve, reject) => {
        consturl = `${API_URL}${PEOPLE_URL.replace(':id', id)}`
        $
            .get(url, opts, function(data){
                resolve(data)
            })
            .fail(() => reject(id))
    })
}

functiononError(id){
    console.log(`ERORRRRRRR!!!!!!!!!! No se pudo obtener el personaje con id = ${i
}
```

```
var ids = []
for (let i = 1; i <= 10; i++) {
    ids.push(i)
}
console.log(ids.length)
var promesas = ids.map( id => obtenerPersonaje(id) )

Promise
    .all(promesas)
    .then(personajes => console.log(personajes))
    .catch(onError)
```

38 - Async-await: lo último en asincronismo

Async-await es la manera más sencilla y clara de realizar tareas asíncronas. Se parece mucho a la forma de escribir código hace unos años, de manera secuencial, desde arriba hacia abajo. Lo que sí, incluye algunas palabras clave.

Como primer cambio a nuestro código anterior vamos a crear una función en la que incorporamos desde la declaración de la variable 'ids' hasta incluso el .catch. A continuación llamamos a la función:

```
functionobtenerPersonajes(){
    var ids = []
    for (let i = 1; i <= 10; i++) {
        ids.push(i)
    }
    varpromesas = ids.map( id => obtenerPersonaje(id) )
    Promise
        .all(promesas)
        .then(personajes => console.log(personajes))
        .catch(onError)
}
obtenerPersonajes()
```

Sección VIII - Complementos

46 - var, let y const: las diferencias entre ellos

- Cuando declaramos variables con ‘var’ siempre conviene declararlas ‘arriba’ del código en el que sea claro cuáles van a ser las variables que se van a usar dentro de nuestra función o programa.
- Dentro de una función javascript detecta todas las variables declaradas con ‘var’ y las ‘declara’ por sí solo como si estuvieran ‘arriba’ en el código. Por lo que si se declara un var dentro de un bloque else y este no es accedido por el condicional, la variable ‘var’ declarada dentro de ese else existe de todas maneras.
- Si utilizamos ‘let’, el alcance de esa variable se ve reducido únicamente al bloque de código donde es utilizado.
- ‘const’ se comporta parecido a let sólo que no es posible reasignarlo.
- Es posible modificar una variable ‘const’ en el caso de un array[ ] con el método push() por ejemplo.
- Reducir siempre al mínimo el alcance de nuestras variables.
- Utilizar 'let’ si tenemos que reasignar una variable.
- Si nunca tenemos que reasignar una variable usamos ‘const’.

47 - Manejo de Fechas:

¿Hace cuántos días naciste?

Con variables de tipo Date, se pueden realizar operaciones de suma y resta similares a las que se realizan con números. El resultado que se obtiene está en milisegundos, por lo que luego hay que hacer algunas operaciones adicionales para llevarlos a días, meses o años según queramos. También aplica para Horas, Minutos, Segundos y Milisegundos.

Math.abs() nos permite poner cualquier fecha en el primer orden de tal manera que el resultado siempre sea positivo o ‘absoluto’’. La suma o resta nos da un número expresado en milisegundos.

```
functiondiasEntreFechas(fecha1, fecha2) {
        const unDia = 1000 * 60 *60 * 24        //1000ms x 60sec x 60min x 24hs
        const diferencia = Math.abs(fecha1 - fecha2)
```

```
        returnMath.floor(diferencia / unDia)
}

const hoy = newDate()   // fecha actual por defecto
const nacimiento = newDate(1981, 08, 12)   // el constructor de new Date() recibe a

diasEntreFechas(hoy, nacimiento)  // == 13638   si divido esta cantidad de días por
```

48 - Funciones Recursivas

La recursividad es un concepto muy importante en cualquier lenguaje de programación. Una función recursiva es básicamente aquella que se llama (o se ejecuta) a sí misma de forma controlada, hasta que sucede una condición base.

Para realizar recursividad necesitamos 2 cosa; 1 caso base y 1 caso recursivo. Usaremos una división para mostrar las funciones recursivas.

Algoritmo de nuestro ejercicio:

```
13 /4___

13 - 4 = 9              1       => caso recursivo
9 - 4 = 5               1       => caso recursivo
5 - 4 = 1               1       => caso recursivo
1 - 4 = -3              0       => caso base
```

La función:

```
function divisionEntera(dividendo, divisor) {
        if (dividendo < divisor) {
                return 0
        }
        return 1 + divisionEntera(dividendo - divisor, divisor)
}
```

'dividendo - divisor' en el primer parámetro de nuestra función recursiva invocada dentro de sí misma sería el 'caso recursivo' que se repetirá mientras la condición no se cumpla. Cuando la función se cumple, 'caso base' termina la recursividad, termina la operación y nos entrega el resultado de la división en valor entero.

49 - Memorización: ahorrando cómputo

Este proceso nos va a permitir ahorrar procesamiento, ahorrar cómputo, guardando ciertos resultados de algunas cuentas.

Usaremos otro ejemplo de recursividad:

Factoriales: !6 = 6* 5 * 4 * 3 * 2 * 1 = 720 !12 = 12 * 11 * 10 * 9 * ... * 1 == 12 * 11 * 10 * 9 * 8 * 7 * !6

Cómo guardamos los resultados para no tener que volver a realizar cuentas ya hechas.

```
function factorial(n) {
        if(n ===1) {
                return 1
        }
        returnn * factorial(n - 1)
}
```

Ahora guardamos en una cache los resultados de las operaciones ya hechas.

```
function factorial(n) {
        if(!this.cache) {
                this.cache = {}
        }
        if (this.cache[n]){
                return this.cache[n]
        }
        if(n === 1) {
                return 1
        }
        this.cache[n] = n * factorial(n - 1)
        return this.cache[n]
}
```

50 - Closures

Un closure es una función que recuerda el estado de las cosas cuando fue creada. Una función que devuelve otra función con parámetros invocados en dos veces; primero el de la función 'padre' y luego el de la función 'hijo'.

Para ejemplificar generemos una función que va a crear saludos. En este caso hagamos un saludo argentino, uno mexicano y otro para colombia.

```
function crearSaludo(finalDeFrase){
        return function(nombre){

        }
}
```

La función 'padre' es generadora o creadora de otras funciones y la función 'hijo' es anónima, es la que nos va a devolver el resultado. Vamos a llamar esta función para crear constantes.

```
const saludoArgentino = crearSaludo('che')
const saludoMexicano = crearSaludo('wey')
const saludoColombiano = crearSaludo('amigo')
```

Entonces ahora podemos llamar a la función nuevamente a través de cada variable constante y pasando el parámetro de la función 'hijo' esta vez, el parámetro 'nombre' para generar el saludo de la siguiente manera.

```
saludoArgentino('Pablo')
// Hola Pablo che

saludoMexicano('Pablo')
// Hola Pablo wey

saludoColombiano('Pablo')
// Hola Pablo amigo
```

Y agregamos la respuesta de la función (el console.log en este caso) en la que accedemos a la variable 'finalDeFrase' generada en la declaración de los diferentes saludos

```
function crearSaludo(finalDeFrase) {
        return function(nombre) {
                console.log(`Hola ${nombre}${finalDeFrase}`)
        }
}
```

La variable 'finalDeFrase' es la generada en las constantes con el nombre mismo de la función, a partir del parámetro que le pasamos; 'che', 'wey' o 'amigo' en este caso. La función 'hijo' recuerda cada una de las variables generadas que se usó para crear el saludo. Y la función 'hijo' va a ser cada una de las constantes creadas; saludoArgentino, saludoMexicano o saludoColombiano en este mismo caso.

```
const saludoArgentino = crearSaludo('che')
```

Al invocar la función 'hijo' luego, le pasamos el parámetro 'nombre' y así la función se completa y nos imprime el saludo 'Hola Pablo che' en este caso.

```
saludoArgentino('Pablo')
// Hola Pablo che
```

El código completo queda así:

```
function crearSaludo(finalDeFrase) {
        return function(nombre) {
                console.log(`Hola ${nombre} ${finalDeFrase}`)
        }
}

const saludoArgentino = crearSaludo('che')
const saludoMexicano = crearSaludo('wey')
const saludoColombiano = crearSaludo('amigo')

saludoArgentino('Pablo')
// Hola Pablo che

saludoMexicano('Pablo')
// Hola Pablo wey

saludoColombiano('Pablo')
// Hola Pablo amigo
```

51 - Estructuras de datos inmutables

Las estructuras de datos inmutables nos van a permitir deshacernos de los “efectos colaterales” cuando estamos desarrollando (side effects; efecto de lado según Sacha).

Dada el siguiente código:

```
const pablo = {
        nombre: 'Pablo',
        apellido: 'Andrés',
        edad: 30
}

const cumpleanos = persona => persona.edad++
```

La función modificará la edad en el objeto cada vez que se ejecute:

```
pablo
//{nombre: "Pablo", apellido: "Andrés", edad: 30}
```

```
cumpleanos(pablo)
//30

pablo
//{nombre: "Pablo", apellido: "Andrés", edad: 31}

cumpleanos(pablo)
//31

pablo
//{nombre: "Pablo", apellido: "Andrés", edad: 32}

cumpleanos(pablo)
//32

pablo
//{nombre: "Pablo", apellido: "Andrés", edad: 33}
```

Este es el llamado side effect (efecto de lado). La función puede modificar el objeto sin que nosotros así lo queramos. Para evitar este efecto colateral definimos una función inmutable.

```
const cumpleanosInmutable = persona => ({
        ...persona,
        edad: persona.edad + 1
})
```

Si le pasamos el objeto 'pablo' la función nos devolverá un nuevo objeto sin modificar el anterior.

```
pablo
// {nombre: "Pablo", apellido: "Andrés", edad: 33}

cumpleanosInmutable(pablo)
// {nombre: "Pablo", apellido: "Andrés", edad: 34}

cumpleanosInmutable(pablo)
// {nombre: "Pablo", apellido: "Andrés", edad: 34}

pablo
// {nombre: "Pablo", apellido: "Andrés", edad: 33}

cumpleanosInmutable(pablo)
// {nombre: "Pablo", apellido: "Andrés", edad: 34}

cumpleanosInmutable(pablo)
// {nombre: "Pablo", apellido: "Andrés", edad: 34}

pablo
// {nombre: "Pablo", apellido: "Andrés", edad: 33}
```

La "desventaja" que tendremos es que para guardar el valor de la función vamos a tener que generar una nueva variable.

```
const pabloViejo = cumpleanosInmutable(pablo)
const pabloMasViejo = cumpleanosInmutable(pabloViejo)
```

Utilizar estructuras de datos es parte de las buenas prácticas de javascript ya que nos permite deshacernos de los “efectos de lado” y no preocuparnos de modificar código inconscientemente y que se “rompa todo” en cualquier otro lado.

52 - Cambiando de contexto al llamar a una función

El contexto en javascript está definido por el objeto ‘this’ cuando se ejecuta un código. Es muy común el error: ‘No se puede ejecutar este método porque es indefinido’, esto sucede porque el ‘this’ no es quien esperamos que sea.

Dado el siguiente código:

```
const pablo = {
        nombre: 'Pablo',
        apellido: 'Andrés',
}
const mariela = {
        nombre: 'Mariela',
        apellido: 'Riesnik',
}

functionsaludar() {
        console.log(`Hola, mi nombre es ${this.nombre}`)
}

// Si ejecuto:

saludar()
// Hola, mi nombre es undefined

// Ya que tenemos la función definida dentro de un contexto global el 'this' en sal

window.saludar()
// Hola, mi nombre es undefined
```

Cómo hacemos para cambiar ese ‘this’ de la función? El método ‘.bind()’ se usa justamente para cambiar en contexto, el ‘this’, en una función.

```
const saludarAPablo = saludar.bind(pablo)
const saludarAMariela = saludar.bind(mariela)
```

‘.bind()’ nos devuelve una nueva función atando el parámetro,_ ‘(pablo)’_ en este caso, al ‘this’ dentro de esa función, saludar en este caso. Este nunca modifica a la función original.

```
saludarAPablo()
// Hola, mi nombre es Pablo

saludarAMariela()
// Hola, mi nombre es Mariela
```

Otra forma de usarlo:

```
setTimeout( saludar.bind(pablo), 1000) == setTimeout(saludarAPablo, 1000)
```

Y otra, agregado un parámetro a la función:

```
functionsaludar(saludo = 'Hola') {
        console.log(`${saludo}, mi nombre es ${this.nombre}`)
}
setTimeout( saludar.bind(pablo, 'Hola loco!'), 1000)
// Hola loco!, mi nombre es Pablo

//también se puede agragar en la declaración de la constante.
const saludarAPablo = saludar.bind(pablo, 'Hola loco!!')
// Hola loco!!, mi nombre es Pablo
```

El primer parámetro es el contexto y luego van los siguientes parámetros en el orden en el que aparezcan.

IMPORTANTE!!! La función .bind() no ejecuta la función a la que se agrega, sino que simplemente nos retorna una nueva función con ese contexto cambiado. Usando el método .bind, enviamos la referencia a la función sin ejecutarla, pasando el contexto como parámetro.

Otros dos métodos que nos sirven para cambiar el contexto son: .call y .apply.

Usando el método .call, ejecutamos inmediatamente la función con el contexto indicado.

```
saludar.call(pablo)
// Hola, mi nombre es Pablo  -  se ejecuta inmediatamente

saludar.bind(pablo)
// no produce ningún resultado, nola ejecuta.
```

Al .call le pasamos los parámetros separados por ‘,’ igual que en el .bind.

```
saludar.call(pablo, 'Hola cheeee!!')
// Hola cheeee!!, mi nombre es Pablo
```

Usando el método .apply, es similar a .call pero los parámetros adicionales se pasan como un arreglo de valores.

```
saludar.apply(pablo, ['Hola mi querido'])
// Hola mi querido, mi nombre es Pablo
```

Manejar a dónde refiere el '.this' es algo que tenemos que tener muy presente, sobretodo cuando escribimos en modo asíncrono, ya que siempre que ejecutemos una función de esta naturaleza el '.this' siempre cambia y es muy importante atarlo a nuestra clase, objeto o función.

53 - ¿Cuándo hace falta poner el punto y coma al final de la línea?

En general javascript no necesita el ";" para funcionar correctamente. "Es opcional".

Hay dos situaciones donde si no lo usamos, el código nos va a arrojar un error.

- Caso 1 - cuando comienzo una nueva línea escribiendo un array en la línea anterior va ";" si no da error

```
console.log('Lorem ipsum dolor...') ;
[1, 2, 3].forEach( n => {console.log(n * 2)} )
```

- Caso 2 - Si comienzo una línea con comillas invertidas "`", en la línea anterior o al principio de la misma va ";".

```
const pablo = 'Pablo'
console.log('Lorem ipsum dolor...') ;
`${nombre} es un desarrollador.`
```

Otra situación que puede dar problemas es el salto de línea con enter. Inmediatamente después de un 'return' un enter es interpretado como ";". Si le doy un enter a la llave que viene después de este da error.

```
// Esta manera funciona
function calcularDoble(numero) {
        return {
                original: numero, doble: numero * 2
        }
```

```
}

// ERROR
function calcularDoble(numero) {
        return // ...enter aquí...
        {
                original: numero, doble: numero * 2
        }
}
```

Y todavía otra (que me crucé al inicio del 'Curso de jQuery a JavaScript'); es al llamar una función que se auto-ejecute *(function myFunction(){})()* después de un string declarado con cualquier tipo de comillas; dobles, simples o invertidas; | " | ' | ` |.

```
const str = `lorem ipsum dolor`

( function myFunction(){
        alert('Si usas ";" sí funciono!')
})()

// Uncaught TypeError: "lorem ipsum dolor" is not a function at <anonymous>:4:2
```

La forma correcta sería:

```
const str = `lorem ipsum dolor` ;
( function myFunction(){
        alert('Si usas ";" sí funciono!')
})()

// ó
const str = `lorem ipsum dolor`
; ( function myFunction(){
        alert('Si usas ";" sí funciono!')
})()
```