* **Variables**
  + Se declaran usando la palabra let o var, pero no es aconsejable utilizar var porque afecta al alcance de la misma dentro de una función.
* **Constantes**
  + Se declaran utilizando la palabra const.
* **Variables globales**
  + Si no se declara una variable con var, let o const entonces será una variable global, para accesar a ella será por medio de window.variable.
* **Strings**
  + Se puede utilizar “ (comillas doble), ‘ (comilla simple) o ` (comilla invertida). Si se utiliza las comillas invertidas se pueden poner operaciones dentro de las comillas con ${}.

Ejemplo: console.log(`El área del triángulo es ${5 \* 7 / 2}`)

* + Funciones:
    - str.toLowerCase() - Retorna string en minúsculas.
    - str.endsWith(“ar”) - Informa si una palabra termina con el string mandado como parámetro.
    - str.slice(0, -2) - Permite cortar caracteres (desde 0 hasta 2 caracteres antes de que termine el string).
    - str.startsWith(“z”) - Informa si una palabra comienza con el string mandado como parámetro.
    - str.length - Retorna el tamaño del string.
    - str.split(‘’).reverse().join(‘’) - Convierte cadena a array, invierte los índices y convierte de nuevo a string.
    - str.charAt(3) - Retorna el carácter en el índice mandado como parámetro.
* **Math**
  + Math.random() - Retorna un valor entre 0 y 1
  + Math.floor(3.5) - Redondea el valor del parámetro hacia abajo.
  + Math.round(3.5) - Redondea el valor del parámetro.
  + Math.sqrt(10) - Retorna la raíz cuadrada del parámetro.
* **Decimales**
  + dec.toFixed(2) - Redondea decimal al número de decimales que se mandan como parámetro.
* **Fechas**
  + new Date(1981, 1, 19) - Retorna fecha 19/02/1981
  + new Date() - Retorna la fecha de hoy
  + date.getFullYear() - Retorna el año de la variable date.
  + date.getMonth() - Retorna el mes de la variable date.
  + date.getDate() - Retorna el día del mes de la variable date.
  + date.getDay() - Retorna el día de la semana donde 0 es Domingo y 6 es Sábado.
* **Colecciones**
  + Arrays
    - Ejemplo:

const días = [

“lunes”,

“martes”,

“miercoles”,

“jueves”,

“viernes”

]

* + - Operaciones con arrays:
      * reduce(function, iniAcum) - Esta función permite acumular todos los elementos del array. Hacerlo en un for no es la manera correcta de utilizar las propiedades del lenguaje.

Ejemplo:

function suma(...numeros) {

return numeros.reduce(function (acum, numero) {

aum += numero

return acum

}, 0)

}

o

const suma = (...numeros) => numeros.reduce((acum, numero) => acum += numero, 0)

* + - * push() - Permite agregar un elemento en el arreglo.

Ejemplo:

const resultado = []

resultado.push(24 \* 2)

* + - * map(function) - Esta función permite obtener un nuevo arreglo pero pudiendo hacer operaciones en cada índice antes de copiarlo.

Ejemplo:

//Los returns en las arrow functions no son necesarios.

function dobles(...numeros) {

return numeros.map(function (numero) {

return numero \* 2

})

}

o

const dobles = (...numeros) => numeros.map(numero => numero \* 2)

* + - * filter(function) - Recibe función que retorna un boolean. Sirve para regresar cuales elementos de un arreglo cumplen con una condición.

Ejemplo:

function pares(...numeros) {

return numeros.filter(function (numero) {

return numero % 2 == 0

})

}

o

const pares = (...numeros) => numeros.filter(numero => numero % 2 == 0)

* + - * forEach(function) - Recibe una función que es lo que hará por cada uno de los elementos del array.
* **Funciones**
  + Se pueden escribir de diferentes formas:
    - Forma tradicional:

Ejemplo:

function triangleArea(base, height) {

return base \* height / 2

}

* + - Variable tipo function:

Ejemplo:

let triangleArea = function (base, height) {

return base \* height / 2

}

* + - Arrow functions:

Ejemplo:

let triangleArea = (base, height) => {

return base \* height / 2

}

o

let triangleArea = (base, height) => base \* height / 2

* + Se pueden asignar valores por defecto a los parámetros de una función para que si no se manda valor siempre tenga uno.

Ejemplo:

const canWatchMovie = (name, age, isWithAdult = false) => {

if (age >= 13) {

alert (`${name} can watch the movie`)

} else if (age < 13 && isWithAdult) {

alert (`${name} has ${age} but is with an adult, so can watch the movie`)

} else {

alert (`Cannot watch the movie because ${name} has ${age}`)

}

}

* + Se pueden mandar llamar funciones con parámetros sin declarar.

Ejemplo:

const p1 = {

x: 5,

y: 0

}

distancia(p1, {x: 20, y: -7})

* + Se pueden realizar funciones con cantidad indefinida de parámetros.

Ejemplo:

function suma(…numeros) {

//Números será un arreglo

}

//Usar la función

suma(1, 8, 9)

suma(5, 7)

* + Closures - Una de las funcionalidades principales de Javascript es que se pueden declarar variables de tipo funciones, esto facilita a la hora de querer hacer código reutilizable.

Ejemplo:

function saludarFamilia(apellido) {

return function saludarMiembroDeFamilia(nombre) {

console.log(‘Hola ${nombre} ${apellido}’)

}

}

const saludarGomez = saludarFamilia(“Gomez”)

saludarGomez(“Pedro”)

const saludarPerez = saludarFamilia(“Perez”)

saludarPerez(“Eduardo”)

* + Las funciones tienen métodos que se pueden utilizar:
    - bind()

1.- Sirve para mandar valores a los parámetros para que la función las pueda utilizar, el primer parámetro es obligatorio y es donde se manda el contexto que se utilizará como this. Retorna una función por lo que se tendrá que asignar a otra variable.

Ejemplo 1:

index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<button id="boton">

Off

</button>

<script src="index.js"></script>

</body>

</html>

index.js

class Toggable {

constructor(el) {

this.el = el

this.el.innerHTML = 'Off'

this.activated = false

this.el.addEventListener('click', this.onClick.bind(this))

}

onClick(ev) {

console.log(`Esto es lo que tiene this: ${this}`)

this.activated = !this.activated

this.toggableText()

}

toggableText() {

this.el.innerHTML = this.activated ? 'On' : 'Off'

}

}

const button = document.getElementById('boton')

const miBoton = new Toggable(boton)

Ejemplo 2:

function saludar(nombre, apellido) {

console.log(`Hola ${nombre} ${apellido}!`)

}

const saludarJahazieles = saludar.bind(null, “Jahaziel”)

saludarJahazieles(“Martinez”)

saludarJahazieles(“Gomez”)

* + - call() - Manda llamar la función con los parámetros que se envían. El primer parámetro es el contexto.

Ejemplo:

const jahaziel = {

nombre: 'Jahaziel',

apellido: 'Martínez'

}

function saludar(veces) {

for (let i = 0 ; i < veces; i++) {

console.log(`Hola ${this.nombre} ${this.apellido}`)

}

}

//Si no se utiliza el call la función mandará undefined porque this hace referencia a

//window y no hay variables con esos nombres, por lo que se tiene que utilizar el call.

saludar.call(jahaziel, 3)

* + - apply() - Manda llamar la función con los parámetros que se envían. El primer parámetro es el contexto y los parámetros que utilizará la función se tienen que mandar en un array.

Ejemplo:

const jahaziel = {

nombre: 'Jahaziel',

apellido: 'Martínez'

}

function saludar(veces, uppercase) {

let str = `Hola ${this.nombre} ${this.apellido}`

str = uppercase ? str.toUpperCase() : str.toLowerCase()

for (let i = 0 ; i < veces; i++) {

console.log(str)

}

}

saludar.apply(jahaziel, [3, true])

* **Objetos**
  + Forma tradicional
    - Ejemplo:

const p1 = {

x: 5,

y: 0,

moverEnX: function moverEnX(x) {this.x += x},

moverEnY: function moverEnY(y) {this.y += y}

}

//Usar el objeto

p1.moverEnX(10)

* + - **NOTA:**

Es buena práctica que cuando se agregan funciones en un objeto, siempre declararlas con nombre, no dejarlas anónimas para que se facilite la identificación de un error en caso de haberlo.

* + **Prototipos de objetos**
    - Ejemplo:

function Punto(x, y) {

this.x: x

this.y: y

}

Punto.prototype.moverEnX = function moverEnX(x) {

this.x += x

}

Punto.prototype.moverEnY = function moverEnY(y) {

this.y += y

}

//Crear objetos

const p1 = new Punto(2, 0)

const p2 = new Punto(4, -5)

//Usar el objeto

p1.moverEnX(10)

//Modificar el prototipo de un objeto

p1.\_\_proto\_\_.moverEnX = function() { alert(‘Hola’) }

* + - **NOTA:**

A diferencia de JAVA, **si modificamos el prototipo de p1, se modificará también el p2** porque comparten el mismo prototipo.

* + **Objetos con Object.create**
    - Ejemplo:

const Punto = {

init: function init(x, y) {

this.x = x

this.y = y

},

moverEnX: function moverEnX(x) {

this.x += x

},

moverEnY: function moverEnY(y) {

this.y += y

}

}

//Crear objetos

const p1 = Object.create(Punto)

const p2 = Objets.create(Punto)

//Inicializar objetos, es necesario

p1.init(2, 0)

p2.init(4 -5)

* + - **NOTA**

A diferencia de JAVA, **si modificamos el prototipo de p1, se modificará también el p2** porque comparten el mismo prototipo.

* + **Objetos con class (nueva forma de crear objetos en Javascript)**
    - Ejemplo:

class Punto {

constructor(x, y) {

this.x = x

this.y = y

}

moverEnX(x) {

this.x += x

}

moverEnY(y) {

this.y += y

}

}

//Crear objetos

const p1 = new Punto(2, 0)

const p2 = new Punto(4, -5)

//Usar el objeto

p1.moverEnX(10)

//Modificar el prototipo de un objeto

p1.\_\_proto\_\_.moverEnX = function() { alert(‘Hola’) }

* + - **NOTA**

A diferencia de JAVA, **si modificamos el prototipo de p1, se modificará también el p2** porque comparten el mismo prototipo.

* Alcance (Scope)
  + Se debe evitar declarar variables con la palabra reservada var porque esto altera el alcance de esta variable que se está declarando. Si se declara una variable con var en un for por ejemplo, esa variable estará disponible en toda la función, no solo en el bucle; sin embargo, si utilizamos la palabra reservada let se respetara el alcance de esa variable en su ámbito, que en el ejemplo anterior será solo en el bucle.
* This
  + Nos podemos referir al objeto actual utilizando la palabra reservada this, pero si el objeto se utiliza alguna función fuera de la case no se podrá utilizar la palabra reservada this, pero hay 3 soluciones para poder utilizar this.

Ejemplo:

//Manda error, no se puede utilizar this en este ámbito.

class Persona {

constructor(nombre, amigos = []) {

this.nombre = nombre

this.amigos = amigos

}

listarAmigos() {

this.amigos.forEach(function (amigo) {

console.log(`Hola, mi nombre es ${this.nombre} y son amigo de ${amigo}`)

})

}

}

const jazzo = new Persona("Jahaziel", ["Pedro", "Julieta"])

jazzo.listarAmigos()

//Solución 1: Utilizar variable copia de this.

class Persona {

constructor(nombre, amigos = []) {

this.nombre = nombre

this.amigos = amigos

}

listarAmigos() {

let \_this = this

this.amigos.forEach(function (amigo) {

console.log(`Hola, mi nombre es ${\_this.nombre} y son amigo de ${amigo}`)

})

}

}

const jazzo = new Persona("Jahaziel", ["Pedro", "Julieta"])

jazzo.listarAmigos()

//Solución 2: Indicarle a la función que puede utilizar this.

class Persona {

constructor(nombre, amigos = []) {

this.nombre = nombre

this.amigos = amigos

}

listarAmigos() {

this.amigos.forEach(function (amigo) {

console.log(`Hola, mi nombre es ${this.nombre} y son amigo de ${amigo}`)

}.bind(this))

}

}

const jazzo = new Persona("Jahaziel", ["Pedro", "Julieta"])

jazzo.listarAmigos()

//Solución 3: Utilizar arrow functions.

class Persona {

constructor(nombre, amigos = []) {

this.nombre = nombre

this.amigos = amigos

}

listarAmigos() {

this.amigos.forEach((amigo) => {

console.log(`Hola, mi nombre es ${this.nombre} y son amigo de ${amigo}`)

})

}

}

const jazzo = new Persona("Jahaziel", ["Pedro", "Julieta"])

jazzo.listarAmigos()

* Compatibilidad
  + La herramienta más utilizada para traducir lenguaje ES6 o ES7 a ES5 y hacerlo compatible con todos los navegadores es Babel. Es necesario tener instalado NodeJS.
* Módulos
  + Un módulo es un pedazo de código que se encarga de resolver un problema en específico. Se utilizan módulos para encapsular funcionalidad y utilizar dependencias en lugar de repetir código.
* Callbacks
  + En Javascript primero se realiza el código que está en la pila de tareas del hilo principal y posteriormente realiza lo que esté en el callback (WEB API o C++ Lib).

Ejemplo:

console.log('A')

setTimeout(function () {

console.log('B')

})

for(let x = 0; x < 999999999; x++) ;

console.log('C')

* setTimeout
  + Permite realizar un callback pasado algún tiempo definido.

Ejemplo:

setTimeout(() => {

//Código a realizar pasado el tiempo del 2do parámetro.

}, <tiempo en milisegundos>)

* setInterval
  + Permite realizar algo cada cierto tiempo.

Ejemplo:

setInterval(() => {

//Código que se realizará cada que pase el tiempo del 2do parámetro.

}, <tiempo en milisegundos>)

* Cancelar timers
  + Para cancelar timeout - clearTimeout(<timeout>)

Ejemplo:

const $cancelButton = document.getElementById(‘boton’);

const timeout = setTimeout(() => {

alert(‘Han pasado 5 segundos’);

}, 5000);

$cancelButton.addEventListener(‘click’, () => {

clearTimeout(timeout);

});

* + Para cancelar interval – clearInterval(<interval>)

Ejemplo:

const $cancelButton = document.getElementById(‘boton’);

let counter = 0;

const interval = setInterval(() => {

counter++;

console.log(´Han pasado ${counter} segundos´);

}, 1000);

$cancelButton.addEventListener(‘click’, () => {

clearInterval(interval);

});

* Promesas
  + Las promesas son objetos y están asociadas con alguna tarea asíncrona. Son muy útiles para realizar callbacks a servidores externos permitiéndonos fácilmente anidar llamados.

Ejemplo:

function get(URL) {

return new Promise((resolve, reject) => {

const xhr = new XMLHttpRequest()

xhr.onreadystatechange = function () {

const DONE = 4

const OK = 200

if (this.readyState === DONE) {

if (this.status === OK) {

resolve(JSON.parse(this.responseText))

} else {

reject(new Error(`Error al realizar request: ${this.status}`))

}

}

}

xhr.open('GET', URL)

xhr.send(null)

})

}

let objLuck

get('https://swapi.co/api/people/1/')

.then(luck => {

objLuck = luck

return get(luck.homeworld)

})

.then(homeword => console.log(`${objLuck.name} nacio en ${homeword.name}`))

.catch(err =>console.log(`${err}`))

Ejemplo con fetch:

let objLuck

fetch('https://swapi.co/api/people/1/')

.then(response => response.json())

.then(luck => {

objLuck = luck

return fetch(luck.homeworld)

})

.then(response => response.json())

.then(homeworld => {

console.log(`${objLuck.name} nacio en ${homeworld.name}`)

})

.catch(err =>console.log(`${err}`))

* Async-await
  + Nuevo uso de código javascript con las palabras async y await. Lo que hace es que javascript detiene su ejecución cuando ve await y cuando termina continua con la siguiente línea.

Ejemplo:

async function getLuck() {

try {

const response = await fetch('https://swapi.co/api/people/1/')

const luck = await response.json()

const responseHomeworld = await fetch(luck.homeworld)

luck.homeworld = await responseHomeworld.json()

console.log(`${luck.name} nacio en ${luck.homeworld.name}`)

} catch (err) {

console.log(`${err}`)

}

}

* Iteradores
  + Los iteradores permiten hacer listas infinitas de elementos.

Ejemplo 1:

//El algoritmo retorna la suma de los 2 valores anteriores

function fibonacci() {

let a = 0, b = 1

return {

next: (reset) => {

if(reset) a = 0, b = 1

let f = a

a = b

b = f + a

return {

value: f,

done: false

}

}

}

}

const fibo = fibonacci()

fibo.next().value

fibo.next().value

fibo.next().value

fibo.next(true).value

Ejemplo 2:

function fibonacci() {

let a = 0, b = 1

return {

next: function () {

f = a

a = b

b = f + a

return {

value: f,

done: false

}

}

}

}

const fibo = {}

fibo[Symbol.iterator] = fibonacci

let i = 1

for(let value of fibo) {

console.log(value)

i++

if(i > 20) break;

}

* Generadores
  + Una función de tipo generador podrá ser detenida o continuada y nosotros tenemos el control de ello.

Ejemplo:

function\* fibonacci() {

let a = 0, b = 1

while(true) {

let f = a

a = b

b = f + a

let reset = yield f

if(reset) a = 0, b = 1

}

}

const fibo = fibonacci()

fibo.next()

fibo.next()

fibo.next()

fibo.next(true)

* Datos inmutables
  + En Javascript al igual que en Java, cuando asignamos un objeto a otro nuevo en realidad estamos creando un objeto que apunta a la misma dirección de memoria que el copiado, por lo que si se modifica uno, el otro también será afectado. Para poder hacer otro objeto independiente apuntando a otra dirección de memoria se utiliza Object.assign.

Ejemplo:

function (persona) {

const clone = Object.assing({}, persona)

clone.edad++

return clone

}