# 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 acceder 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}`)

## 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 1:**

const días = [

“lunes”,

“martes”,

“miercoles”,

“jueves”,

“viernes”

]

**Ejemplo 2:**

let arreglo = new Array(5).fill(0);

### 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)

#### fill(valor)

Esta función permite inicializar todos los elementos con un valor.

#### 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)

#### 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)

#### forEach(function)

Recibe una función que es lo que hará por cada uno de los elementos del array.

# Funciones

## Formas de escribirlas

### 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

## Parámetros

* 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”)

## Métodos de las funciones:

### bind()

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])

# Obtener elementos de DOM

## document.getElementById(‘id’)

Obtiene el elemento del HTML cuyo id sea igual al especificado en el parámetro.

## document.querySelector(`[atributo=”${valor\_buscado}”]`)

Obtiene el elemento del HTML cuyo atributo y valor corresponda con el especificado como parámetro. Útil cuando utilizamos atributos personalizados para nuestro uso particular.

**Ejemplo:**

function getElementByKeyCode(keyCode) {

return document.querySelector(`[data-key="${keyCode}"]`);

}

# Modificar atributos de un elemento

## elemento.classList.add(‘clase’)

Agrega un valor al atributo class.

## elemento.className = ‘clase’

Le asigna un nuevo valor al atributo class borrando cualquier valor que tuviera anteriormente.

# 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.

## Clone

* + - **Ejemplo:**

var clone = Object.assign({}, obj);

# 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()

# 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 1:**

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 2:**

const DESPERTAR = 1000;

const DUCHA = 3000;

const ARREGLARME = 3000;

const DESAYUNAR = 2000;

const MANEJAR = 5000;

const $msg = document.getElementById("msg");

const $botones = document.getElementById("div\_botones");

const $pregunta = document.getElementById("pregunta");

const $btn\_si = document.getElementById("si");

const $btn\_no = document.getElementById("no");

const mostrarPregunta = function(pregunta, resolve, reject) {

$pregunta.textContent = pregunta;

$botones.style.display = "block";

$btn\_si.addEventListener("click", () => {

resolve();

});

$btn\_no.addEventListener("click", () => {

reject("Tarea no realizada");

});

}

const ocultarPregunta = () => {

$botones.style.display = "none";

}

const showError = (message) => {

ocultarPregunta();

$msg.style.color = "red";

$msg.textContent = message;

}

const despertar = function() {

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

setTimeout(() => {

mostrarPregunta("¿Hay agua?", resolve, reject);

}, DESPERTAR);

});

}

const ducha = function() {

ocultarPregunta();

$msg.textContent = "Ahora me estoy bañando";

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

setTimeout(() => {

resolve();

}, DUCHA);

});

}

const vestir = function() {

ocultarPregunta();

$msg.textContent = "Ahora me estoy vistiendo";

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

setTimeout(() => {

mostrarPregunta("¿Hay comida?", resolve, reject);

}, ARREGLARME);

});

}

const comer = function() {

ocultarPregunta();

$msg.textContent = "Ahora estoy comiendo";

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

setTimeout(() => {

resolve();

}, DESPERTAR);

});

}

const manejar = function() {

ocultarPregunta();

$msg.textContent = "Voy rumbo al trabajo";

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

setTimeout(() => {

resolve();

}, MANEJAR);

});

}

despertar()

.then(ducha)

.then(vestir)

.then(comer)

.then(manejar)

.catch(showError)

**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

}

# 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.

Para publicarlo algún módulo a NPM es necesario los siguientes pasos:

1. Cuenta en NPM.
2. Instalar Node.js.
3. Situarnos en la carpeta donde estará nuestro módulo.
4. Configurara NPM.
   1. npm set init.author.name “Nombre”
   2. npm set init.author.email “Correo”
   3. npm set init.author.uri “URI de algún sitio o red social”
   4. npm login
5. Iniciar npm para generar archivo package.json
   1. npm init

Pedirá:

Nombre: “Aquí pones el nombre del paquete “

Version: “Aquí agregas la versión de tu librería, es recomendado que si es la primera iteración sea la versión 0.1.0, si el producto ya está completo, tienes test y de ahí no va a cambiar algo usamos 1.0.0, si son cambios más grandes que rompen lo que teníamos sería un cambio a la version 2.0.0, etc.

Description: “Aquí es donde vamos a explicar que hace nuestro modulo, puedes ser tan extenso como sea necesario, esto es lo que va a leer la comunidad”

Entry point: “Es el archivo principal de nuestra librería, por defecto es index.js”

Test command: “Si nosotros tenemos test, aquí declaramos el comando en el cual podemos ejecutar este test”

Git repository: “Aquí agregamos la URL del repositorio de GitHub en donde estará el código de nuestro proyecto, no es obligatorio pero es recomendado para que otras personas puedan colaborar en tu proyecto de ser necesario”

License: Una de las partes más importantes y que no deberías omitir, es básicamente donde indicamos que tenemos todos los derechos intelectuales del software y que pueden usar nuestra librería de acuerdo a ciertos términos y condiciones, acá la opción recomendada es MIT.

1. Creamos una carpeta src y dentro un archivo .js con nuestras funciones que queremos tener en nuestro módulo.
   1. Las funciones deberán empezar con **export function xxx() {…}**
   2. Si solo es una función en el archivo entonces lo podemos poner como default con **export default function xxx() {…}**
2. Segur instrucciones que tenemos en el título **Compatibilidad** de esta guía para convertir nuestro código y que sea compatible con todos los navegadores.
   1. Contemplar que en scripts del package.json tenemos que indicar que al hacer build, transpile nuestro archivo src/archivo.js hacia dist con la siguiente línea:

**“build”: babel scr/archivo.js -d dist**

Y cambiar el main de index.js al nombre que tenemos de nuestro archivo base ya transpilado de la carpeta dist.

1. Utilizar git, agregar a nuestro .gitignore la carpeta “node\_modules” donde están las dependencias ya que no forma parte de nuestro código, hacer add, commit y versionar nuestro commit con un tag.
2. Subir proyecto a GitHub.
   1. git push origin master
   2. git push origin master --tags
3. Correr comando **npm publish** para que se publique en NPM nuestra librería**.**

# 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.

1. Página home de babel (https://babeljs.io/), Docs🡪Setup, Babel build-ins🡪CLI
2. Seguir instrucciones para instalar babel-cli estando dentro de la carpeta de nuestro proyecto e instrucciones para configurar y poder transpilar nuestro código.

**NOTA**: En scripts del package.json tenemos que indicar que al hacer build, transpile nuestro archivo src/archivo.js hacia dist con la siguiente línea:

**“build”: babel scr/mi\_archivo.js -d dist**

Y cambiar el main de index.js al nombre que tenemos de nuestro archivo base ya transpilado de la carpeta dist.

# Mejores mensajes al usuario

Podemos utilizar librerías externas como sweet alert.

https://sweetalert.js.org/ 🡪 Get started

# Test

Se recomienda que los test se escriban antes de que se codifique la funcionalidad.

## Pasos

### 1.- Instalar librerías

**npm install --save-dev mocha chai**

### 2.- Crear una carpeta test

Esta carpeta deberá estar al nivel de nuestra carpeta src.

### 3.- Escribir test en ES5

Crear archivo .js donde escribiremos nuestros test.

**Ejemplo:**

const expect = require(‘chai’).expect

const ejemplo = require(‘..’).default

describe(‘#ejemplo’, function() {

it(‘descripción de mi escenario 1 o regla 1’, function () {

const resultado = ejemplo(<argumentos>);

expect(resultado).to.equal(<valor esperado que retorne la función>)

});

it(‘descripción de mi escenario 2 o regla 2’, function () {

})

it(‘descripción de mi escenario 3 o regla 3’, function () {

})

}

### 4.- Agregar script para correr test en package.json

En scripts agregar uno para test

**Ejemplo:**

//Agregar coma después del último script si existen más scripts

“test”: “mocha”

### 5.- Correr test

**npm test**