

Preface / Introducción

el libro de mano sigue la regla 80/20: Aprender en 20% del tiempo el 80% de un tema.

en particular, el objetivo es darte la velocidad necesaria con javascript

este libro es escrito por Flavio, el publica programas y tutoriales en el blog flaviocopes.com, y organiza un Campamento Anual en bootcamp.dev
Disfruten,

Nota: yo Sabas camilo ramirez jimenez me encargue de traducir esta pieza al español esperando Ayudar y compartir el conocimiento.

4.1 White Space:

JavaScript no considera el "Espacio en blanco"
Con Significado al igual Que "Line Breaks"
"ruptura de linea o "Enter", pueden Ser
usados de cualquier manera, aunque esto
es pensado en la teoria.

en la practica te agradara Seguir un buen
estilo y adherirte a lo que las personas
común mente usan, y fortalecer esto usando
un "linter" o una herramienta de estilo como lo
es "prettier" (Son Extensiones)

For Example, I like to always use 2 Characters
to indent

2 Case Sensitive

JavaScript es sensible a las Mayúsculas,
una variable llamada "Algo" es diferente a "algo"
dósmismo va para cualquier identificador o función

4.5 Literales

Nosotros describimos como literales un valor que es escrito en la fuente del Código, por Ejemplo un numero, una cadena, un booleano, o un constructo mas Avanzado com "Objeto literal" ó "Arrays literales"

5

'Test'

true

['a', 'b']

{ color: 'red', shape: 'Rectangle' }

4.6 Identificadores

un identificador es una Secuencia de caracteres que pueden ser usados para identificar una Variable, una funcion, o un objeto. este puede iniciar con una letra, es Signo dolar \$ ó un guion guion, y puede Contener digitos. Usando Unicode una letra puede identificarse como un Emojí 😊, Siendo esto un Caracter como Mayusculas ☺ o un Anoba.

Alt+123 , Alt+64

Test, TEST, Test1

test, _test, \$test

* El Símbolo dolar es Comúnmente Referenciado

Como DOM elements

* Algunos nombres son reservados por JavaScript para uso interno, y nosotros no podemos usarlos como identificadores.

4.5 Comentarios.

Los comentarios son una de las mas importantes partes de cualquier programa. Son importantes porque nos dejan Anotar el código y Agregar importante información que de otra manera no estaría disponible para otros personas (o nosotros mismos) al leer el Código.

en JavaScript, podemos escribir un comentario en una sola linea usando "://"

Todo después de "://" no es considerado como código por el interpretador

// un comentario

true // otro comentario

Otro tipo de comentario es multi-lines comentario que empieza en "/*" y termina en "*/"

/* Algun tipo de Comentario
una linea xd

*/

5. Semicolons.

Toda linea en javascript es opcionalmente terminada usando Semicolons
al decir opcionalmente, es porque el interprete es lo suficientemente inteligente para introducir los Semicolons por ti.

en la mayoria de casos, tu puedes omitir los Semicolons altogether from your programs.

este hecho es muy controversial, y tu podrias siempre encontrar codigo que lo usa y codigo que no.

Mi opcion personal en preferencia es para siempre evitar Semicolons al menos que sea estrictamente necesarios.

6. Valores

un "hola" como cadena de texto o string es un valor, un numero como "12" es un valor.
"hola" y "12" son valores, string y number,

Cadena de texto y numero son tipos de cierto valor.

el tipo es el modo de identificar en un conjunto de valor. es categorizar. nosotros tenemos muchos tipos en javascript, y nosotros debemos hablar de ello en detalle despues, cada tipo tiene sus propias caracteristicas.

Cuando necesitamos de dar Referencia a un valor, nosotros Asignamos estos a una variable. la variable podria tener un nombre, y el valor es lo que Almacenamos en una variable, entonces podemos acceder luego a dicho valor atravez del nombre de la variable.

7. Variables.

Una variable es un valor asignado a un identificador, entonces tu puedes referenciarlo y luego usarlo en el programa/codigo.

esto es porque esta Flojamente tipado un Concepto del Cual tu escucharas.

Una variable debe ser declarada antes de que tu la puedas usar.

Hay 2 maneras principales, el primero es usar const: (Que Referencia a una variable) (no cambia)

const a = 0

La Segunda manera de usar es Let:

let a = 0

¿Cuál es la diferencia?

const define una constante referencia a un valor
esto significa que la referencia no puede ser cambiada,
no puedes reasignar un valor a ella.

Usando `let` puedes Asignarle un nuevo valor a el.

`const` no Significa "Constante" en la manera que otros idiomas lo hacen, en particular, tampoco es que no sea un valor que no puede cambiar - el `const` Significa que no puede ser reasignada, si la variable apunta a un objeto o un `Array` (veremos mas sobre objetos y Arrays mas Adelante) el Contenido de el objeto o el `Array` puede libremente cambiar.

```
/* Const variables must be initialized  
at the declaration time: */
```

Const $a=0$

but `Let` values can be initialized later:

1º. Const a=1, / 2º Let c=1 pero no puedes declarar
b=2. / d=2 la misma variable varias veces.

Cannot redeclare a variable twice or more times:

Example: Let a=1

Let a=2

You will receive an error of "declaration duplicated".

My advice is always use `const` and only use `let`. When you know you need to reassign a value to that variable. [why]? Because the less power our code has, the better. If we know that a value cannot be reassigned, then it is a source of fewer errors or bugs.

Now that we see how to work with `Const` and `Let`, I want to mention about `Var`.

Until 2015, `Var` was the only way to declare a variable in JavaScript. To date, a modern codebase/base of code uses mainly `const` and `let`. Where some fundamental differences are visible, which are detailed in another way, but if you are just starting, you should not worry about them.

Only use `const` and `let`.

8. Tips

The variables in javascript do not have any

Tipo declarado es decir: son no tipadas.

Una vez le asignas un valor con algún tipo a una variable, tu puedes después reasignar la variable a almacenar un valor de otro tipo, sin ningún problema.

en JavaScript tenemos 2 principales Categorías de Tipado: Primitive types y Object types.

8.1. Tipos Primitivos

Son:

- o Números
- o Cadenas
- o Booleanos
- o Símbolos
- o NULL
- o Undefined

8.2 Object types

Cualquier valor que no es un tipo de dato primitivo es un objeto.

Los tipos de objetos tienen propiedades y también tienen métodos que pueden actuar en esas propiedades

Hablaremos sobre objetos más tarde y más a fondo.

9. Expresiones

Una Expresion es una sola unidad de javascript code que Engine de javascript. puede evaluar, y retornar un valor.

Expresiones Pueden Variar en su complejidad, Iniciaremos desde las que son muy Simples llamadas Expresiones primarias

2.

0.02

; ('Something')

true

false

this // the current scope

undefined

i //where i is a variable or a constant.

Arithmetic expressions son expresiones que toman una variable y un operador (mas de operadores pronto), y resultar en un numero:

1/2

i++

i = 2

i * 2

Expresiones en Cadena son Expresiones que resultan dentro de una cadena

'A' + 'String'

Logical expressions / expresiones lógicas hacen uso de operadores lógicos y resuelven a un valor booleano

a && b

Las Expresiones mas avanzadas incluyen objetos, funciones, y arrays de los cuales hablaremos despues.

a || b

!a

10. Operators / Operadores

te permiten obtener dos expresiones simples y combinarlas entre ellas para formar una Expresion mas compleja.

We can classify operators / podemos clasificar operadores basados en los operandos, con lo que ellos trabajan. Algunos con 1 operando, muchos con 2 operandos. Solo un operador trabaja con 3 operandos.

en esta primera introducción a los operadores debemos introducir los operadores con los que tu deberas ser mayormente familiar: operadores binarios.

yo ya he introducido uno cuando al hablar acerca de las variables: El Operador de Asignacion = . Es usado = para Asignar un valor a una variable:

Let b=2.

10.1. el operador de Adicion (+)

const three = 1 + 2

three + 1 // 4

'three' + 1 // three + 1

10.2. el operador de Sustraccion (-)

const two = 4 - 2

10.3 el operador de division (/)

retorna el cociente de el primer operador y el Segundo

const result = 20 / 5 // result === 4

const result = 20 / 7 // result === 2.857142857

Si tu divides por zero, javascript no lanza ningun error pero retorna el valor **Infinity** (o el valor **-Infinity**, si el valor es negativo)

1 / 0 // Infinity

1 / 0 // -Infinity

10.4. el operador resto o remanente (%)

the remainder by zero is always NaN, a Special Value that means "not a Number":

1 % 0 // NaN, es muy util en varios casos de usos

const result = 20 % 5 // result === 0

Const result = 20 % 7 // Resultado == 6

10.5 el factor de Multiplicación, operador multiplicación (*)

Tan sencillo como suena multiplica 2 números
(lo cual es una suma consecutiva)

1 * 2 // 2

1 * 2 // 2

10.6 el operador de Exponenciación (**)

levanta el primer operador hacia el poder del
segundo operando

1 ** 2 // 1

2 ** 1 // 2

2 ** 2 // 4

2 ** 8 // 256

8 ** 2 // 64

11. Reglas Precedentes

todo Complemento Estamento con múltiples operadores
en la misma línea con múltiples operadores en la
misma línea tendrá problemas precedentes

Toma este Ejemplo

Let a = 1 * 2 + ((5/2) % 2)

El resultado es 2.5 pero por?

¿Qué operaciones son ejecutadas primero y cuáles necesitan esperar?

Algunas operaciones son ejecutadas primero, debido a los precedentes que tienen las reglas de precedentes están listadas en esta tabla:

Operadores	Descripción
* / %	Multiplicación / División
+ / -	Adición / Sustracción
=	Asignación

Operadores en el mismo nivel (como + and -) son ejecutados en el orden que ellos son encontrados, desde izquierda hacia la derecha.

Siguiendo estas reglas, la operación anterior puede ser resuelta en esta manera

$$\text{Let } a = 1 * 2 / ((5/2) \% 2)$$

$$\text{Let } a = 2 + ((5/2) \% 2)$$

$$\text{Let } a = 2 + (2.5 \% 2)$$

$$\text{Let } a = 2 + 0.5$$

$$\text{Let } a = 2.5$$

12. Comparison operators / Operadores de Comparación

A continuación

después de Asignadores y operadores matemáticos, el tercer set de operadores que yo quiero introducir es operadores de comparación.

Tú puedes usar el siguiente operador para comparar dos números, o dos cadenas.

Los operadores de comparación siempre regresan un booleano, un valor que es cierto o falso.

Operadores para comparar desigualdad

< menor que, <= menor o igual que, > Mayor que,
>= Mayor o igual que

Ejemplo:

```
let a=2  
a>=1 // true
```

en adición a ello, tenemos 4 operadores de equivalencia, ellos accept two values y regresar un booleano

- Checks for equality == // igual
- checks for inequality != // desigual

Nota: en javascript nosotros tambien tenemos
== y != en javascript, pero yo solo uso
== y != porque esto puede prevenirmos de algunos
problemas menores

13. Condicionales

Con la comparacion operadores en sitio,
nosotros podemos hablar acerca de condicionales

un estamento de un **if** es usado para hacer
el programa tomar una ruta, o otra, dependiendo
en el resultado de una Expresión Evaluando.

este es el Ejemplo Simple que siempre ejecuta

```
if(true){ //do something;  
    // "Haga algo";
```

Por el contrario, este es nunca ejecutado

```
if(false){ // do something; // never xd?
```

el condicional revisa la Expresion y al pasar por
ella encontraras un valor verdadero o falso,

1. Si es un numero, lo valdra como un True/
verdadero a menos que sea cero

Si pasas por un `String`, el valor sera `True` al menos que sea un valor vacio, estas son reglas generales de invocacion de ciertos tipos de booleanos.

Notaste las llaves `{ } []` eso se llama Bloque, y se usa para agrupar una lista de diferentes declaraciones.

Se puede colocar un bloque donde pueda tener una sola declaracion y si tienes una sola declaracion para ejecutar despues de los condicionales, puede omitir el bloque, y solo escribe la declaracion:

```
if (true) doSomething()
```

Pero siempre querras usar las llaves para ser mas claro.

13.1. Else (sino)

Tu puedes proveer una segunda parte del si estamentando un `else`

Tu puedes colocar un estamento a lo que vayas a ejecutar si la condicion `Si` es falsa

```
if (true) { // do Something }  
else { do Something else }
```

Sin { // do Something }

desde que el else acepta estamentos,
tu puedes anidar otro if/else dentro de la
declaración

```
if (a === true) { // do something } else if (b ===  
else if (b === true) { // do something else }  
else {  
    // fallback  
}
```

14. Arrays

un Array es una Colección de elementos.

un Array en javascript no es un tipo/type
por si mismo.

un Array es un objeto

nosotros podremos inicializar un objeto
vacío o un Array vacío, en estas 2 diferentes
maneras

```
const a = []
```

```
const a = Array()
```

La primera es usando la sintaxis literal del Array, el segundo usa el Array en función

tambien tu puedes rellenar el Array usando esta sintaxis.

```
const a = [1, 2, 3]
```

```
const a = Array.of(1, 2, 3)
```

Un Array puede contener cualquier valor, incluso diferentes tipos de valor

```
const a = [1, 'SABAS', [a, b, d]]
```

desde que podemos agregar un Array dentro de un Array, nosotros podremos crear Arrays multidimensionales, los cuales seran Aplicaciones muy utiles (e.g a matrix)

```
const matrix = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9],  
]
```

```
matrix[0][0] // 1  
matrix[2][0] // 7
```

tu puedes Acceder a cualquier elemento de el Array por referencia de su index el cual inicia desde zero 0:

```
a[0] // 1
```

```
a[1] // 2
```

```
a[2] // 3
```

Tú puedes inicializar un nuevo Array con un Set de valores usando esta sintaxis, la cual primero inicia un Array de 12 elementos, y rellena cada elemento con el numero 0

Array(12).fill(0)

Tú puedes tener el numero de elementos en el Array revisando su propiedad length

const a = [1, 2, 3]
a.length // 3

Nota: tú puedes colocar el largo de el Array. Si tú Asignas un mayor numero que la capacidad actual del Array, nada pasa. Si tú Asignas un menor numero, el Array es cortado en esa posicion.

const a = [1, 2, 3]
a // [1, 2, 3]
a.length = 2
a // [1, 2]

14.1. Cómo Agregar un objeto a un Array

Nosotros podemos Agregar un elemento

al final de el Array Usando el push() metodo:

a. push(4)

podemos Agregar un elemento al inicio de un Array usando el unshift() method:

a. unshift(0)

a. Unshift(-2,-1)

14.2 como remover un item desde un array

podemos remover un item desde el final de un Array usando el pop() metodo

a. pop()

Nosotros podemos remover un item desde el inicio de un Array usando el shift() method:

a. shift()

14.3 como unir 2 o mas Arrays

Tu puedes unir multiples Arrays usando concat()

Const a = [1,2] Const C = a.concat(b) // [1,2,3,4]

Const b = [3,4] a // [1,2]

b // [3,4]

Tu tambien puedes usar el spread operator (...) en esta manera;

const a = [1, 2]

const b = [3, 4]

const c = [...a, ...b]

c // [1, 2, 3, 4]

14.4. Como poder encontrar un específico item en el Array.

Tu puedes usar el find() metodo de un Array:

a.find(element, index, array) => ↴

// regresar true or false

})

Regresa el primer item que retorna verdad, regresa Undefined si el elemento no es encontrado

Un comando usado Syntax es:

a.find(x) => x.id === my_id)

The above line will return the first element in the Array that has id === my_id.

findIndex() works similarly to find(), but

returns the index of the first item that returns true, y si no es encontrado, el regresa undefined:

a. findIndex (element, index, array) \Rightarrow {
 // return true or false
}

Otro metodo es includes():

a. includes (value)

returns true / regresa verdadero if a contiene un valor

a. includes (value, i)

retorna verdad if a contains value after the position i.

15. Strings

Una string es una secuencia de caracteres.

el tambien puede ser definido como una literal cadena, la cual es encerrada en Apostrofe o doble frases:

'A string'

'Another string'

y usar double frases solo en html para definir Attributos.

Yo personalmente prefiero

solo Apostrofes todo el tiempo,

solo en html para definir Attributos.

Tu asignas un cadena valor a una variable como esto

```
const name = 'flavio'
```

Tu puedes determinar el largo de una cadena usando el length como propiedad de el:

```
'Hola mundo' // 10
```

```
const name = 'Hola mundo'
```

```
name.length // 6
```

este es un cadena vacia: '' Its length propiedad es 0:

```
'', length // 0
```

las dos cadenas pueden ser unidas usando el operador +:

```
'A' + 'String'
```

Tu puedes usar el + operador para interpolar variables:

```
const name = 'Sabas'
```

```
'my name is' + name // My name is Sabas
```

Otra manera para definir cadenas es el usar plantillas literales, definidas dentro

Comillas

backticks. ellas son especialmente útiles para hacer multilíneas de cadenas lo cual es más simple.

Con comillas simples o comillas dobles frases que no pueden definir una cadena multilinea esta cadena es fácilmente usadas por cadenas multilineas se necesita usar caracteres de escape.

Entonces una plantilla literal es abierta con la comilla , tu solo presiona enter para crear una nueva linea sin ningún carácter especial, y esto es tomado tal cual es:

```
const string = 'hey this string is Awesome!'
```

Las plantillas literales también son geniales porque ellas proveen una fácil manera de interponer variables y expresiones dentro de cadenas

Lo puedes hacer usando \${...} Syntax:

```
const var = 'test'
```

```
const string = 'Something ${var}'
```

```
// Something test
```

Dentro del \${ } tu puedes agregar cualquier de estas Expresiones

```
const string = 'something ${1+2+3}'  
const String2 = 'something  
$ ${foo() ? 'x' : 'y'}'
```

16. loops

Loops estos son una de las principales estructuras de control de javascript.

Con un loop podemos automatizar y repetir indefinidamente un bloque de código por cuantas veces nosotros queramos que corra / ejecute.

javascript provee muchas maneras para repetir Atravez de loops

Yo quiero enfocarme en 3 maneras

- While Loops
- for Loops
- for...of Loops

16.1. While

The while loop es el mas simplest looping structure that javascript provide us //
el loop Mientras es el mas simple en estructura que java nos provee

al agregar una condición después del `while` (palabra clave) y nosotros proveer un bloque que corra hasta que la condición evalúe en verdadero.

Ejemplo: `const list = ['a', 'b', 'c']`

`let i = 0`

`while (i < list.length) {`

`console.log(list[i]) // value`

`console.log(i) // index`

`i = i + 1`

`}`

Tú puedes interrumpir un loop `while` usando el `break` (palabra clave) como este ejemplo:

`while (true) {`

`if (somethingIsTrue) break`

`}`

y si tú decides que en el medio de un loop
quieres saltar a la siguiente iteración
usando continuar `continue`

`while (true) {`

`if (somethingIsTrue) continue`

`// do something else`

Nota: la traducción continua Mas Adelante

Muy similar al while, y tenemos q do..while como ciclo repetitivo, este basicamente es el mismo que un ciclo while, excepto que la condición es Evaluada despues de que el bloque de código es Ejecutado.

esto significa q el bloque de código es siempre ejecutado al menos una vez.

Ejemplo

```
Const list = ['a', 'b', 'c']
Let i = 0
do {
    console.log(list[i]) // Value
    console.log(i) // index
    i = i + 1
} while (i < list.length)
```

16.2. for

la Segunda Mas importante Estructura de ciclo repetitivo en javascript es el **for loop**

Nosotros usamos el **For** como palabra clave y pasamos un conjunto de 3 instrucciones la inicialización, la condición y la parte que se incrementa.

Ejemplo:

```
const list = ['a', 'b', 'c']
```

```
for (let i = 0; i < list.length; i++) {  
    console.log(list[i] // value  
    console.log(i) // index
```

Justo como con los loops **while**, tu puedes interrumpir un **for** loop usando un **break** y puedes rápidamente adelantar a la siguiente iteración de un **for** loop usando **continue**

16.3. for...of

Este loop es relativamente reciente introducido en 2015 y es una versión simplificada del ciclo **for**

```
const list = ['a', 'b', 'c']
```

```
for (const value of list) {  
    console.log(value) // value
```

17. Funciones

en cualquier moderadamente complejo programa de javascript, todo sucede dentro de sus funciones

las funciones son como un Nucleo y parte esencial de JavaScript.

{Que es una función?}

Una función es un bloque de código, autocontenido
esta es una función declarada

```
function getData() {  
    // Do something  
}
```

Una función puede correr en cualquier momento que tu quieras invocarla, como la siguiente

```
getData()
```

Una función puede tener uno o más argumentos

```
function getData() {  
    // do something  
}
```

```
function getData(color) {  
    // do something  
}
```

```
function getData(color, age) {  
    // do something  
}
```

When we can pass an argument, we invoke.

Cuando nosotros pasamos un argumento,
nosotros invocamos la función pasandole parámetros

```
function getData(color, age) {  
    // do something  
}
```

```
getData('green', 24)  
getData('black')
```

Notese que en la segunda invocación yo pase la
cadena **black** como un parámetro de argumento
de color, pero no la edad, en este caso **age**
dentro de la función es indefinida

tambien podemos revisar si el valor es indefinido
undefined usando este condicional.

```
function getData(color, age) {  
    // do something  
    if (typeof age !== 'undefined') {  
        // ...  
    }  
}
```

typeof is a unary operator that allows us to check the type of a variable

//
typeof es un operador unario que nos permite revisar el tipo de una variable

tu tambien puedes revisar en esta manera

```
function getData(color, age) {  
    // do something  
    if (age) { // ... }  
}
```

Aunque en este caso el condicional podria ser falso si la edad es nula, o una cadena vacia

Tu puedes tener valores por defecto para los parametros, en caso de que no hayan pasado

```
function getData (color = 'black', age = 25) {  
    //do something  
}
```

Tu puedes pasar cualquier valor como un parametro: numeros, cadenas, booleanos, Arrays, objetos y tambien funciones

Una Funcion tiene como retorno un valor, por defecto una funcion regresa undefined, a menos que agregues una palabra clave **return** con un valor

```
funcion getData() { //... do something
```

```
    return 'hi'
```

```
}
```

```
let result = getData()
```

La funcion **result** ahora sostiene una Cadena de texto con el valor **'hi'**

Tu solo puedes retornar un valor

Para retornar multiples valores

tu puedes retornar un objeto, o un Array, como este

```
function getData() {  
    return ['flavio', 37]  
}
```

```
let [name, Age] = getData
```

Las funciones pueden ser definidas dentro de otras funciones:

```
const getData = () => {  
    const doSomething = () => {}  
    doSomething()  
    return 'test'  
}
```

la función anidada no puede ser llamada desde afuera de la función encerrada

tambien,
tu puedes regresar una función de una función,

18. Funciones Arrow

las funciones Arrow son una reciente introducción a javascript

estas son bastante usadas por el contrario de las funciones regulares, la uno que yo describo en el capítulo previo, tu encontraras ambas formas usadas en muchas partes

visualmente, estos permiten que tu escribas las funciones con cortas sintaxis desde

```
function getData() {  
    //...  
}
```

9

```
;() => {  
    //...  
}
```

pero notamos que no tenemos nombre aquí las funciones Arrow Son Anónimas, nosotros debemos asignarles a ellas una variable, nosotros podemos asignar UNA UNA función regular a una variable, como esta:

```
let getData = function getData() {  
    //...  
}
```

When we do so,

Cuando nosotros lo hacemos, nosotros podemos
remover el nombre desde la función

```
Let getData = function () {  
    //...
```

}

y llamar la función usando el nombre de la
variable

```
Let getData = function () {  
    //...  
}  
getData()
```

esta es la misma cosa que nosotros hacemos con
las funciones Flecha

```
Let getData = () => {  
    //...  
}  
getData()
```

si el cuerpo de la función contiene justamente un solo
estamento, tu puedes omitir el parentesis y escribirlo
todo en una sola linea, ejemplo

```
Const getData = () => console.log ('hi')
```

los parámetros son pasados en el parentesis

const getData = (param1, param2) => console.log(
- param1, param2)

si tu tienes uno y (solo un parámetro), tu puedes omitir el parentesis completamente

Const getData = (parámetro) => console.log(parámetro)

Las funciones Arrow (flecha) te permiten tener un retorno implícito: los valores son retornados sin tener la necesidad de usar la palabra clave return

esto funciona cuando hay una declaración en linea en el cuerpo de la función

const getData = () => 'test'

getData() // 'test'

Como con las funciones regulares, nosotros podemos tener parámetros por defecto:

tu puedes tener valores principales por parámetros, en caso de que ellos no hayan pasado

const getData = (color = 'black', Age = 2) => {
// ... do something }

y nosotros solo podemos retornar un valor
las funciones Arrow pueden contener otra
función Arrow, o también funciones regulares.
ellas son bastante similares, entonces te podrás
preguntar por que ellas son presentadas?
la gran diferencia con las funciones regulares es cuando
ellas son usadas como métodos de objeto
esto es algo que muy pronto veremos

19. Objetos

Cualquier valor que no sea de tipo primitivo (cadena de
texto, un numero, un booleano, un simbolo, nulo o
indefinido) es un objeto

aquí vemos como se define un objeto

```
const car = {}
```

esta es la sintaxis literal del objeto, la cual es
una de las cosas mas buenas de javascript

tú tambien puedes usar el `new Object` syntax

```
const car = new Object()
```

Otra sintaxis es el usar `Object.create()`:

```
const car = Object.create()
```

Tú también puedes inicializar un objeto usando el `new` palabra clave antes de una función con una letra capital. Esta función sirve como un constructor para ese objeto. En eso, nosotros podemos inicializar el argumento que recibimos como parámetros, para configurar el estado inicial de el objeto.

```
function Car(brand, model) {
```

```
    this.brand = brand
```

```
    this.model = model
```

Nosotros inicializamos un nuevo objeto usando

```
const myCar = new Car('Ford', 'Fiesta')
```

```
myCar.brand // 'Ford'
```

```
myCar.model // 'Fiesta'
```

Los objetos son siempre pasados por referencia. Si tú asignas una variable el mismo valor de otro, si este es de tipo primitivo como un número o una cadena de texto, ellos son pasados como un valor.

Toma este ejemplo

```
let age = 36  
let myAge = Age  
myAge = 37  
age // 36
```

```
const car = {  
    color: blue,  
}
```

```
const anotherCar = car  
anotherCar.color = yellow  
car.color // 'yellow'
```

Incluso los Arrays o funciones son bajo la cubierta objetos entonces es muy importante el entender como ello funciona

19.1 Propiedades de Objeto

los objetos tienen propiedades, las cuales estan compuestas por una etiqueta asociada con un valor.

el valor de una propiedad puede ser de cualquier tipo, lo que significa que puede ser un Array, una función, y esto incluso puede ser un objeto, Así como los objetos pueden Anidar otros objetos

esta es la sintaxis literal que nosotros vimos en el capítulo previo

```
const car = {}
```

nosotros podemos definir una propiedad de color en esta manera,

```
const car = { color: 'blue' }
```

Aquí nosotros tenemos un carro `car` objeto con una propiedad color con el valor `blue`

las etiquetas pueden ser cualquier cadena de texto, pero cuidado con los caracteres especiales: si yo quisiera incluir un carácter no válido como el nombre de una variable en el nombre de la propiedad, habría que poner comillas o Apostrofes alrededor de ella:

```
const car = { color: 'blue', "the color": 'blue' }
```

los caracteres de nombre de variables invalidos incluyen espacios, guiones y otros caracteres especiales.

Así como ves, cuando nosotros tenemos múltiples propiedades, nosotros Separamos cada propiedad con una coma.

nosotros podemos recuperar el valor de una propiedad usando 2 diferentes sintaxis

- el primero es notación de punto.

```
[car.color // 'blue']
```

- el Segundo (el cual es solo uno que nosotros podemos usar para las propiedades con nombres invalidos), es el uso de corchetes cuadrados

```
[car['the color'] // blue]
```

Si tu accedes a una propiedad inexistente, tu obtendras el valor undefined:

```
[car.brand // undefined]
```

como se dijo, los objetos pueden tener dentro de si otros objetos como propiedades:

```
const car = { brand: { name: 'Ford' }, color: 'blue' }
```

en este ejemplo, tu puedes acceder al nombre de la marca usando

```
[car.brand.name]
```

```
[car[brand][name]]
```

Tu puedes poner el valor de una propiedad cuando tu definas el objeto, pero tu podras siempre actualizarlo despues

```
const car = { color: 'blue', };
```

```
car.color = 'yellow'
```

```
car['color'] = 'red'
```

y tu tambien puedes agregar nuevas propiedades a un objeto

```
car.model = 'Fiesta'
```

```
car.model // 'Fiesta'
```

Dado el objeto

```
const car = { color: 'blue', brand: 'ford', }
```

ademas puedes eliminar una propiedad de este objeto usando

Delete car.brand

19.2 Metodos de objeto

Ya habiamos hablado acerca de las funciones en un capitulo anterior,

las funciones pueden ser asignadas a las propiedades de una funcion y en este caso estas seran llamadas metodos.

en este ejemplo, la propiedad `start` la cual tiene una funcion asignada y nosotros podemos invocarla

a ella usando la sintaxis de puntos que usamos para las propiedades, con el parentesis al final

```
const car = {  
    brand: 'Ford'  
    model: 'Fiesta'  
    start: function() { console.log('started') }  
}  
car.start()
```

dentro de un metodo definido usando una function () {} nosotros tenemos acceso a la syntaxis del objeto instanciando la referencia como this

en el siguiente ejemplo nosotros tenemos acceso a los valores de propiedad de marca y modelo (brand, model) Usando this.brand y this.model

```
const car = {  
    brand: 'Ford'  
    model: 'Fiesta'  
    start: function() { console.log('started')  
        ${this.brand} ${this.model}  
    }  
}  
car.start()
```

esto es porque las funciones Arrow (Flecha) no
están atadas a el objeto.
esta es la razón por la que funciones regulares
a menudo son usados como método de objeto,
los métodos pueden aceptar parámetros, como las
funciones regulares

```
const car = {  
    brand: 'Ford'  
    model: 'Fiesta'  
    goTo: function(destination){console.log(`Going to  
        ${destination}`)}  
}  
  
car.goTo('Rome')
```

20. Classes - Clases

Hemos hablado acerca de objetos, los cuales
son uno de los mas interesantes partes de JavaScript
en este capítulo nosotros subiremos un nivel,
introduciendo clases.

Que son las clases? Son las maneras de definir un
patrón común para multiples objetos

```
const person = { name: 'Flavio' }
```

nosotros podemos crear una clase llamada Person
(notar la P mayúscula, una costumbre cuando usamos
clases) estas tienen **name** como propiedad

class Person { name }

Ahora desde esta clase, nosotros inicializamos un
flavio como un objeto

const flavio = new Person()

flavio es llamado una instancia de la clase
Person

nosotros podemos poner el valor de la propiedad
name

flavio.name = 'Flavio'

y podemos acceder a él usando

flavio.name

Como nosotros lo hacemos por propiedades de objeto.
las clases pueden sostener propiedades como **name**
y métodos.

Los métodos son definidos en esta manera

```
class Person {  
    hello() {  
        return 'Hello, I am Flavio'  
    }  
}
```

y nosotros podemos invocar métodos en una instancia de la clase

```
class Person {  
    Hello() {  
        return 'Hello, I am Flavio'  
    }  
}
```

```
const flavio = new Person()  
flavio.hello()
```

Existe un método especial llamado constructor() el cual podemos usar para inicializar la clase propiedades cuando nosotros creamos una nueva instancia de objeto.

el funciona así:

```
Class Person {  
    constructor(name) { this.name = name }  
    hello() { return 'Hello, I am' + this.name }  
}
```

nota como usamos [this] para acceder a la instancia del objeto

ahora podemos instanciar un nuevo objeto desde la clase, pasando una cadena `String`, y cuando nosotros llamemos `hello`, nosotros obtendremos un mensaje personalizado

`Const flavio = new Person ('flavio')`

`flavio.hello() // 'Hello, I am flavio.'`

Cuando el objeto es inicializado, el `constructor` metodo es llamado con ningun parametro pasado.

normalmente los objetos son definidos como instancias de objetos, no sobre la clase.

tu puedes definir un metodo como estatico, `static` para permitirle ser ejecutado sobre la clase

```
class Person {
```

```
    static genericHello() {
```

```
        return 'Hello'
```

```
}
```

```
}
```

```
Person.genericHello() // Hello
```

esto es muy util, en algunos momentos

21. Inheritance - Herencia

una clase puede Extender a otra clase y objetos inicializados usando esta clase herencia a todos los metodos de Ambas clases.

Supongamos que nosotros tenemos una clase Person

```
Class Person {
```

```
    hello() { return 'Hello, I am a person' }
```

nosotros podemos definir una nueva clase Programmer que se extiende hasta Person

```
Class Programmer extends Person {
```

ahora si nosotros instanciamos un nuevo objeto con clase programmer, el tendra acceso al metodo

```
const flavio = new Programmer()
```

```
flavio.hello() // 'Hello, I am a person'
```

Dentro de una clase hija, tu puedes referenciar a la clase padre llamandola super()

```
class Programmer extends Person { hello() {
    return super.hello() + ' I am also a programmer.'
}
}

const flavio = new Programmer()
flavio.hello()
```

el programa de Arriba imprime 'Hello, I am a Person,
I am also a programmer.'

22. Programación asincrona y llamadas de vuelta

La mayoría del tiempo el código de javascript corre de manera sincronizada. esto significa que cuando una linea de código es ejecutada, la siguiente es llamada y ejecutada y así en adelante.

Todo es como tu esperas y como esto funciona en la mayoría de lenguajes de programación. como siempre habrá momentos cuando tu no puedes esperar por una linea de código para ser ejecutada tu no puedes esperar 2 segundos por un gran archivo para cargar y detener el programa completamente tu no puedes solamente esperar por un recurso de internet de ser descargado, antes de hacer algo mas

JavaScript resuelve este problema usando el método devolver llamada

Uno de los simples ejemplos del retorno de llamada son los tiempos o temporizadores que no son parte de javascript pero ellos son proveidos por el buscador y node.js, hablaremos acerca de uno de los temporizadores que tenemos: setTimeout()

el temporizador setTimeout() como función acepta 2 argumentos: una función y un número. el número es en milisegundos y debe pasar antes de que la función se ejecute.

ejemplo:

```
setTimeout(() => { // runs after 2 seconds
    console.log('inside the function')
}, 2000)
```

la función conteniendo el `console.log('inside the function')` linea podrá ser ejecutada después de 2 segundos si tu agregas un `console.log('before')` antes de la función, y `console.log('after')` después de ella

```
console.log('before')
```

```
setTimeout(() => { // corre después de 2 segundos
    console.log('inside the function')
}, 2000)
```

```
console.log('after')
```

Tu verás esto pasando en tu consola

before

after

inside the function

La llamada de vuelta función es ejecutada de manera asincrónica, esto es un patrón muy común cuando trabajamos con archivos de sistema, la red, eventos o el DOM en el buscador.

Todas las cosas ya mencionadas no son "principales" en javascript y no son explicadas en este libro pero podrás encontrarlas en los libros de flaskicopes.co ahora veremos como podemos implementar llamadas de retorno en nuestro código.

Nosotros definimos una función que acepta los parámetros de llamada de vuelta [callbacks], el cual es una función.

Cuando el código está listo para invocar la llamada de vuelta, la invocamos pasando el resultado.

```
const doSomething = (callback) => {
```

```
    // do things
```

```
    // do things
```

```
    const result = /* */ callback(result)
```

```
}
```

El código que usa esta función será usado de esta manera.

```
doSomething(result) => { console.log(result); }
```

23. Promesas

Las promesas son las alternativas para lidiar con código asíncrono.

Así como vimos en el capítulo previo, con las llamadas de vuelta nosotros pasamos de una función a otra llamado de función, esta podría ser llamada

Cuando la función haya finalizado el procesamiento.
como esto:

```
doSomething((result) => { console.log(result) })
```

Cuando el doSomething() código termine, el llamará a la función recibida como un parámetro

```
const doSomething = (callback) => { // do things  
  const result = /*..*/ callback(result)
```

El principal problema con este acercamiento es que si necesitamos el resultado de esta función está en el resto de nuestro código, todo nuestro código debe estar anidado dentro del callback y si nosotros tenemos que hacer 2-3 callbacks entrariamos en lo que usualmente es definido como "infierno de callback" con muchos niveles de Sangría o espacioado dentro de las funciones.

```
doSomething((result) => {  
  doSomethingElse((anotherResult) => {  
    doSomethingElseAgain((yetAnotherResult) => {  
      console.log(result)  
    })  
  })  
})
```

las promesas no son la manera de lidiar con esto
en vez de hacer:

```
doSomething((result)) => {  
    console.log(result)  
}
```

nosotros llamamos a una función promise-based
en esta manera

```
doSomething().then((result) => {  
    console.log(result)  
})
```

nosotros primero llamamos la función, entonces nosotros
llamamos un "then()" método que es llamado cuando
la función finalice.

La indentación o sangría no importa, pero tú deberías
usar este estilo para claridad, eso es común para
detectar errores usando un "catch()" método.

```
doSomething()  
.then((result) => { console.log(result) })  
.catch((error) => { console.log(error) })
```

Ahora, seremos capaces de usar esta syntax, el
doSomething() función cuya implementación, debería
ser un poco especial, esto debería usar las promesas
API.

en vez de declararse como una función normal
const doSomething = () => {}

nosotros la declaramos como un objeto
promesa

(const doSomething = new Promise()

y nosotros pasamos una función en el
constructor de la promesa

const doSomething = new Promise(() => {})

esta función recibe 2 parámetros, el primero
es una función que nosotros llamamos para
resolver la promesa, el segundo es una función
que nosotros llamamos rechazar la promesa

const doSomething = new Promise((resolve, reject) =>
{}
})

Resolver una promesa significa complementarlo
exitosamente (lo cual resulta en llamar el método
then() en quien lo usa)

Rechazar una promesa significa terminarla con un
error (lo cual resulta en un llamado del método catch()
de quien lo usa).

```
const doSomething = new Promise(  
  [reslove, reject] => {  
    //somecode  
    const success = /*...*/  
    if (success) {  
      reslove('OK')  
    } else {  
      reject('this error occurred')  
    }  
  }  
)
```

nosotros podemos pasar un parametro para resolver y rechazar funciones de cualquier tipo que queramos

24. Asyncrono y espera

las funciones Asincronas tienen un nivel mas Alto de Abstraccion sobre las promesas.
una funcion asincrona retorna una promesa, como en este ejemplo.

```
const getData = () => {  
  return new Promise([reslove, reject]) => {  
    setTimeout(() => reslove('some data'), 2000)  
  }  
}
```

Cualquier código que quiera usar esta función deberá usar la palabra clave await justo antes de la función.

```
const data = await getData()
```

y haciendo entonces, cualquier dato devuelto por la promesa va a ser asignado al data variable.

en nuestro caso, el dato es el "algún dato" cadena

con una particular advertencia cual sea nosotros usamos la await Keyword, nosotros debemos de hacer dentro una función definida como `async`

como esto:

```
const doSomething = async () => {  
  const data = await getData  
  console.log(data)
```

el `Async/await` duo permite que nosotros tengamos un código limpio y un simple modelo mental para trabajar con código Asincrono.

como tu puedes ver en el ejemplo encima, nuestro código luce muy simple, comparado al código usando promesas, o funciones de llamadas de retorno.

y este es un muy simple ejemplo, el mayor beneficios resurgirán cuando el código es mucho mas complejo.

como un ejemplo, aquí es como tu obtendrás un json recurso usando el fetch API, y parcharlo usando promesas

```
Const getFirstUserData = () => {
  // get users list
  return (
    fetch('/users.json')
    // Parse JSON
    , then((response) => response.json())
    // pick first user
    , then((users) => users[0])
    // get user data
    , then((user) => fetch(`/users/${user.name}`))
    // parse JSON
    , then((userResponse) => userResponse.json())
  )
  getFirstUserData()
```

23. Alcance de variables

Cuando yo introduzco variables, yo hablo acerca de usar la const, let, y var.

el alcance es el conjunto de variables esta es visible a una parte de el programa en java script nosotros tenemos un alcance global, lo cual significa esto esta disponible en cualquier parte de un programa, lo cual significa esto

Ahi es un muy importante diferencia entre var, let, y const declarations

una variable como const o let, y const declaraciones

Una Variable es var dentro una función es solo visible dentro de esa función. Similarmente a funciones argumentos

Una Variable definida como const or let en otra mano es solo visible dentro del bloque cual este es definido.

un bloque es un conjunto de instrucciones Agrupadas dentro de un par de llaves

Como las que podemos encontrar dentro de un if, establecido o un loop, y una función, también esto es importante para entender que un bloque no define un nuevo alcance para var (variables), pero lo hace para let y const.

Esto tiene muchas prácticas implicaciones. Supongamos tu defines una var, variable dentro un if, condicional en una función.

```
function getData() {  
  if(true) {  
    var data = 'Some data'  
    console.log(data)  
  }  
}
```

Pero si tu cambias var data, a let data:

```
function getData() {  
  if(true) {  
    let data = 'Some data'  
  }  
  console.log(data)  
}
```

Tu tendrás un error: ReferenceError: data is not defined

esto es porque var es una función de Alcance, y aquí esta es una cosa especial pasando aquí llamado hospedaje. en corto, la var declaración es movida hasta el inicio de la función más cercana por JavaScript, Antes de que esto corra el código mas o menos esto es como la función luce para JS, internamente

```
function getData() {  
    var data  
    if (true) {  
        data = 'some data'  
    }  
    console.log(data)
```

esto es porque tu puedes también console.log(data) al inicio de una función, incluso antes esta es declarada, y tu podrías obtener undefined, como un valor por esa variable:

```
function getData() {  
    console.log(data)  
    if (true) {  
        var data = 'some data'  
    }  
}
```

Pero si tu intercambias a let, tu obtendras un error Reference Error: data is not defined, porque huespedes no pasan las declaraciones let.

const, sigue las mismas reglas que let: esta es bloqueada en alcance

esto puede ser engañoso al inicio, pero una vez tu realces esta diferencia, entonces tu podras ver por que var, es considerado una mala practica hoy en dia comparado a let: esto hacen tener fibras moviéndose partes, y su alcance es limitado a el bloque, el cual tambien hace ello muy bueno como una variable cida, porque esta cesa de existir despues ^{que} un loop ha finalizado

function doLoop() {

for (var i = 0; i < 10; i++) console.log(i)}

doLoop()

Cuando tu sales del loop i podra ser una valida variable con valor 10

Si tu cambias a let, si tu lo intentas console.log(i) podria resultar en un error ReferenceError: i is not defined.

Conclusion

Muchas gracias por leer este libro

Para mas, empieza sobre flavioeopes.com