

Páginas utilizadas durante mi recorrido de estudio:

[Mozilla - Todo sobre la programación](#)

[Free CodeCamp - practicas y certificados](#)

[Visualizaciones interactivas de los temas de M1](#) === [VisualGo](#)

[La cocina del código - YT](#)

[Jon Mircha - YT](#)

[Soy Dalto - YT](#) y [Dalto Emprende - YT](#)

Recordar que los archivos README poseen más información que la dada en clase y puede resultar muy útil para ciertos problemas.

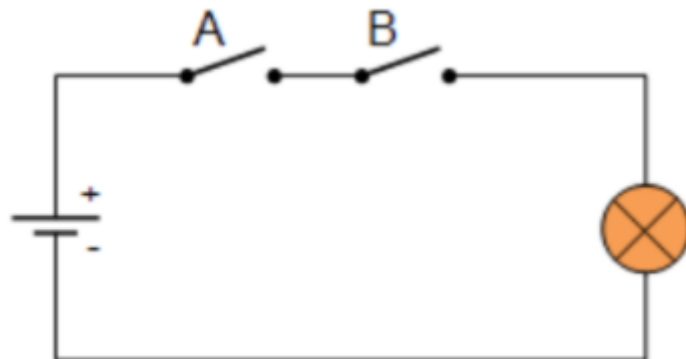
### INTRODUCCIÓN A CS

Es el estudio de procesos algorítmicos, sistemas computacionales y las computadoras per se.

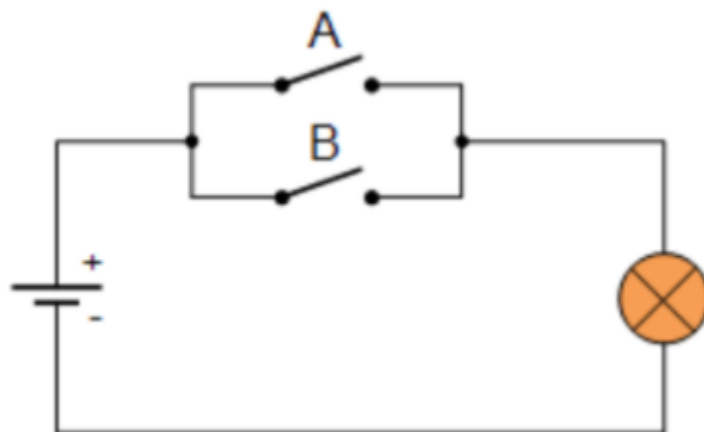
Esta se divide en tres partes igual de importantes:

- 1) **Teoría computacional** (aprendizaje y análisis) : en este mismo se encuentra la teoría de grafos, criptografía, hardware y estructuras de datos. Estos últimos se utilizan para organizar los datos de forma eficiente para su posterior uso.
- 2) **Aplicaciones** : donde se aplica la teoría, como por ejemplo la creación de lenguajes para la programación
- 3) **Ingeniería computacional** (donde nos encontramos nosotros): desarrollo del software

Para darle inicio a esta revolución todo comienza de un circuito eléctrico simple, el cual comparamos a `and(&&)`, `or(||)`, `xor` and `not`.



(&&)



(II)

(las cuatro posibilidades pensadas en interruptores de solo dos valores):

A	B	AND
1	1	1
1	0	0
0	0	0
0	1	0

A	B	OR
1	1	1
1	0	1
0	0	0
0	1	1

A	B	XOR
1	1	0
1	0	1
0	0	0
0	1	1

A	NOT
1	0
0	1

#### Sistema de numeración:

- 1) **Unario** : un único símbolo para representar los caracteres, con la desventaja de no poder simbolizar de manera cómoda y rápida un conjunto de elementos.
- 2) **Números romanos**: 7 símbolos, con la posibilidad de realizar dos operaciones entre ellos, suma y resta ( $XVI = 10 + 5 + 1$ ).
- 3) **Sistema posicional**: Dándole posición a los elementos y, dependiendo su posición, un valor asignado (0-9).

Dependiendo de los usos que le damos a los números y elementos, es la notación a la que recurriremos, es por eso que precisamos una manera sencilla de pasar de una notación a otra.

#### BINARIOS A DECIMALES:

Physical State	OFF	ON	ON	OFF	OFF	ON	ON	OFF
Binary Notation	0	1	1	0	0	1	1	0
Order of Magnitude	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
Decimal Value	128	64	32	16	8	4	2	1
Applicable Value	0	64	32	0	0	4	2	0
Total Decimal Value	$102 = 64 + 32 + 4 + 2$							

**DECIMALES A BINARIO:** lo divido por la base '2', el resto de cada división dara el número en binario.

Tener en cuenta que solo se toma la parte entera y que  $1\%2$  da resto 1.

EG:

$15 / 2 = 7$ , resto 1     $6 / 2 = 3$ , resto 0  
 $7 / 2 = 3$ , resto 1     $3 / 2 = 1$ , resto 1  
 $3 / 2 = 1$ , resto 1     $1 / 2 = 0$ , resto 1  
 $1 / 2 = 0$ , resto 1  
 $0 / 2$  invalido

Los restos en este caso son 1111, por lo que 15 en binario es 1111.

En el caso de 6 los restos serán 011, pero el binario el mismo al revés, es decir, 6 en binario es 110.

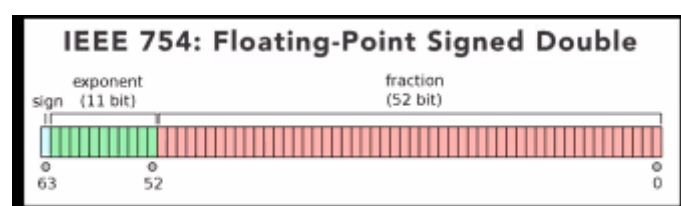
**BINARIOS A ASCII:** donde el código será compuesto por la posición de los bites, y formados por un único byte. Hay problemas con ascii ya que existen caracteres que no contempla.

ASCII					
Binary	Decimal	Glyph	Binary	Decimal	Glyph
0010 1110	46	.	0011 1010	58	:
0010 1111	47	/	0011 1011	59	;
0011 0000	48	0	0011 1100	60	<
0011 0001	49	1	0011 1101	61	=
0011 0010	50	2	0011 1110	62	>
0011 0011	51	3	0011 1111	63	?
0011 0100	52	4	0100 0000	64	@
0011 0101	53	5	0100 0001	65	A
0011 0110	54	6	0100 0010	66	B
0011 0111	55	7	0100 0011	67	C
0011 1000	56	8	0100 0100	68	D
0011 1001	57	9	0100 0101	69	E

**CODIGO UNICODE:** con la posibilidad de ordenar varios bytes, es el codigo que mas caracteres contempla incluyendo emojis.

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

**SISTEMA DE PUNTO FLOTANTE:** almacenamiento de números con decimales.



**LENGUAJE MAQUINA:** Es el lenguaje con el cual se le pasan instrucciones al procesador, esto se realiza en binario (impulsos eléctricos 0 1) y es la forma más básica de comunicación.

**-CUANTO MÁS BAJO ES EL NIVEL DEL LENGUAJE, SE ACERCA MÁS AL LENGUAJE MAQUINA**

**-CUANTO MÁS ALTO ES EL NIVEL DEL LENGUAJE, SE ACERCA MAS A NUESTRO LENGUAJE** (es más fácil para la interpretación humana)

A día de hoy los lenguajes de programación compilan lo que nosotros escribimos para transmitírselo a la máquina.

Para finalizar la introducción a CS, adjunto dos ejercicios, para convertir números binarios a decimales y viceversa:

```
function BinarioADecimal(num) {
  let valorposicion = num.split('').reverse().
  map(function(element,index){
    if(element == 1){
      return Math.pow(2,index);
    }else{
      return 0;
    }
  });
  return valorposicion.reduce((a,b)=>a+b);
}

function DecimalABinario(num) {
  let arr = [];
  while(num != 0){
    arr.unshift(num%2);
    num = Math.floor(num/2);
  }
  return arr.join('');
}
```

### CALL STACK O PILA DE EJECUCIÓN

Es una especie de mapa que utiliza el motor de JS para saber en que función esta ubicado y que funciones pasaron previamente para llegar hasta ahí.

Se llama pila por que funcionan apilados, poniendo elementos arriba de todo y sacando del mismo lugar (el ultimo en entrar es el ultimo en salir).

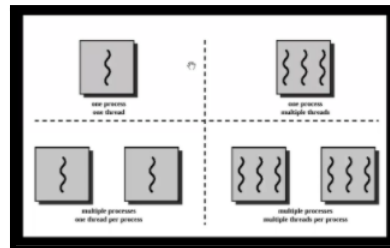
El motor de JS puede ejecutar una cosa a la vez, por eso, cuando termine de trabajar con una función, puede pasar a la que le sigue en la pila, en caso de ser necesario.

En la pila de ejecución se guardan frames o registros que contienen a la función, el contexto de ejecución(scope), el nombre del archivo al que pertenece y el numero de la próxima línea a ejecutar, que todo se va actualizando a medida que ejecutamos, apilando o desapilando hasta que llegamos al final de la función global, dando por terminado el programa.

Es muy importante tener en claro el funcionamiento de ejecución de JS, sobre todo a la hora de la recursión de las funciones.

Para comenzar, es importante tener en cuenta que hay dos formas de procesar tareas:

- 1) Single threaded : un único hilo de ejecución, es decir una acción por momento de ejecución (JS). No se pasa a la siguiente línea a menos que la anterior ya haya sido ejecutada con éxito.
- 2) Sincrónico : Cuando se realizan varias tareas al mismo tiempo, hay formas de emular este comportamiento en JS para ciertas ocasiones.



JS esta compuesto de diferentes mecanismos que permiten su correcto uso, uno de ellos es:

**Syntax Parser:** básicamente lo que hace es leer las líneas de código e interpretar lo que hacemos, y si se encuentra con un error de sintaxis nos avisa.

Generalmente nos determina el problema y la línea en donde se generó el problema, que puede no ser exacta.

Además, trabaja con el **lexical environment** el cual detecta dónde están declaradas las expresiones (variables, sentencias (sueltas)) o statements (if,while,for,etc (codigo dentro de sentencias)), por lo que se comportan y se reconocen de diferente manera.

Al ser invocadas, lexical environment sabrá si puede acceder o no a esas variables, reconociendo el entorno donde fueron expresadas.

(VAR = todos tienen acceso a ella a lo largo del código, LET = pertenece a un bloque de código Y CONST = se mantiene constante y no será modificada).

Otro trabajo de syntax parser es el **hoisting**, el cual lleva a la declaración de funciones y variables (pero no su valor), hacia arriba (guarda espacio de memoria para ellas al principio), por así decirlo, es por eso que podemos invocar funciones aún no declaradas, veamos este caso:

```
> hola();
console.log(name)

function hola(){
  console.log('Hola');
}

var name = 'agus';
Hola
< undefined
```

La función hola se invoca sin problemas, pero name imprime el valor undefined. Cada función hace a su vez su propio hoisting, pero para el let no hay hoisting.

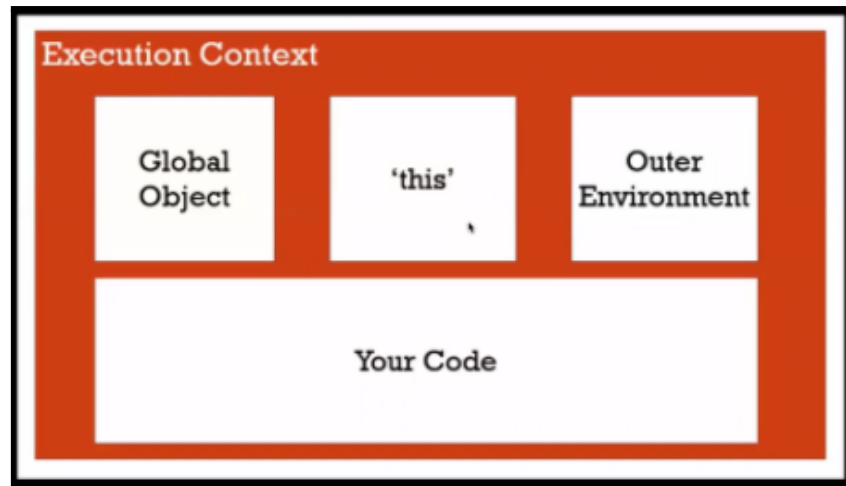
**Execution context :** El contexto de ejecución contiene información sobre el código que se está ejecutando en cada momento, hay un contexto de ejecución global (window en caso de chrome), que es único para cada momento y ejecución, cada entorno tiene su propio contexto global.



Cuando invoque la función person, lo que en realidad sucede es que, le estamos ordenando al motor, que cree el contexto de ejecución (si no la invoco no se crea este contexto), el cual va a almacenar espacio en memoria para las variables que tiene dentro y además va a reconocer su **outer environment** (que hay por fuera de sí(sayHello) (contexto global)). Una vez que se termina de ejecutar la función **se elimina el contexto de ejecución**.

**¿Como se compone este contexto de ejecución?** como ya venimos hablando :

- 1) Objeto global
- 2) This (objeto que se reconoce a sí mismo, el cual almacena información personal sobre el contexto de lo que se está invocando y ejecutando )
- 3) Outer environment
- 4) El código



Cuando creo una variable suelta o una función, estas se almacenan en el contexto global.

```
> var aa = 'agus';
function aaa(){
  console.log('Se almacena en el global por orden alfabetico');
}
< undefined

> this
< Window {window: Window, self: Window, document: document, name: '', location: Location, ...} i
  aa: "agus"
  ▶ aaa: f aaa()
```

Entonces, como conclusión, podemos decir, que los contextos de ejecución se generan en forma de 'pila' (call stack). Cuando genero accidentalmente un loop infinito, se genera un stack overflow.

### SCOPE

Hasta donde alcanza una variable. Si la defino en el contexto global, el scope estará sobre todo el programa y todos pueden acceder a ella. Si defino un let dentro de una función, solo la función tendrá alcance a ella (tener en cuenta para los return statement).

Es por eso que, por ejemplo, para el bucle for, declaramos let i = 0, para que nazca y muera en el bucle.

En el ejemplo siguiente, queda en claro el outer environment:

```

1  var global = 'Hola!';
2
3  function b(){
4      var global = 'Chao';
5      console.log(global); // Chao
6      function a() {
7          // como no hay una variable llamada global en este contexto,
8          // busca en el outer que es scope de b;
9          console.log(global); //Chao
10         global = 'Hello!';
11     }
12     a(); // > se genera dentro de b >> entonces tiene conocimiento de global (b)
13 }
14
15 //a(); Ya no puedo llamar a a desde el scope global, acá no existe.
16 b();
17 console.log(global); // 'Hola!'
18
19
20
21
22
23
24 // apilo contexto de a >
25 // apilo contexto de b > cajita para global, cajita para a();
26 // contexto global > cajita para global, cajita para b()
27
28

```

Entonces al invocar b(): 'chao','chao', b.global = 'Hello!' y console.log(global) // 'Hola!';  
Si quiere invocar a() fuera de b, me dira que a() no esta definida, ya que fuera del contexto de b, a no existe.

### TIPOS DE DATOS PRIMITIVOS

Los datos en JS son de *tipado dinámico*, es decir, las variables no tienen un tipo de dato particular asociado, por lo que podemos asignar y re-asignar cualquier valor a cualquier variable.

Además, es de *tipado débil*, podemos realizar operaciones entre valores de distintos tipos, para esto JS realiza una conversión implícita de tipos para poder concretar la operación.

Por lo tanto, el tipo de dato de una variable se determina cuando se ejecuta la línea de código que contiene a la variable y depende de la operación que realizamos sobre ella.

**¿Que es un dato primitivo?** Son datos básicos, como un string o numero.

\*No poseen ni métodos ni propiedades asociadas.

\*Son inmutables (no podemos cambiar una porción del valor).

Para saber que tipo de dato es el que estamos utilizando, tenemos el operador **TYPE OF**:



```

> var nombres = "agus";
  var numeros = 20;
  var boolean = true;
< undefined
> typeof nombres
< 'string'
> typeof numeros
< 'number'
> typeof boolean
< 'boolean'

```

#### TIPOS DE DATOS PRIMITIVOS:

- 1) **Strings(explayado mas adelante)**: Representan texto y se encierran entre "", ", o `` (nos permite interpolar una variable o función dentro del string ).

```

var nombre = 'agus';
var edad = 20;
undefined
console.log(`Hola me llamo ${nombre} y tengo
${edad} años.`)
Hola me llamo agus y tengo 20 años.   VM539:1
undefined
|

```

Para convertir una variable en sting:

```

> edad.toString();
< '20'
> edad + '';
< '20'

```

<b>s</b>	<b>String()</b>	= 'text'
<b>PROPERTIES</b>		
<b>n</b>	<b>.length</b>	string size
<b>METHODS</b>		
<b>s</b>	<b>.charAt(index)</b>	char at position <b>[i]</b>
<b>n</b>	<b>.charCodeAt(index)</b>	unicode at pos.
<b>n</b>	<b>.codePointAt(index)</b>	cp at position
<b>s</b>	<b>.fromCharCode(n1, n2...)</b>	code to char
<b>s</b>	<b>.fromCodePoint(n1, n2...)</b>	cp to char
<b>s</b>	<b>.concat(str1, str2...)</b>	combine text <b>+</b>
<b>b</b>	<b>.startsWith(str, size)</b>	check beginning
<b>b</b>	<b>.endsWith(str, size)</b>	check ending
<b>b</b>	<b>.includes(str, from)</b>	include substring?
<b>n</b>	<b>.indexOf(str, from)</b>	find substr index
<b>n</b>	<b>.lastIndexOf(str, from)</b>	find from end
<b>n</b>	<b>.search(regex)</b>	search & return index
<b>n</b>	<b>.localeCompare(str, locale, options)</b>	
<b>a</b>	<b>.match(regex)</b>	matches against string
<b>a</b>	<b>.matchAll(regex)</b>	return iterator w/all
<b>s</b>	<b>.normalize(form)</b>	unicode normalize
<b>s</b>	<b>.padEnd(len, pad)</b>	add end padding
<b>s</b>	<b>.padStart(len, pad)</b>	add start padding
<b>s</b>	<b>.repeat(n)</b>	repeat string n times
<b>s</b>	<b>.replace(str regex, newstr func)</b>	
<b>s</b>	<b>.slice(ini, end)</b>	str between ini/end
<b>s</b>	<b>.substr(ini, len)</b>	substr of len length
<b>s</b>	<b>.substring(ini, end)</b>	substr fragment
<b>a</b>	<b>.split(sep regex, limit)</b>	divide string
<b>s</b>	<b>.toLowerCase()</b>	string to lowercase
<b>s</b>	<b>.toUpperCase()</b>	string to uppercase
<b>s</b>	<b>.trim()</b>	remove space from begin/end
<b>s</b>	<b>.trimEnd()</b>	remove space from end
<b>s</b>	<b>.trimStart()</b>	remove space from begin
<b>s</b>	<b>.raw``</b>	template strings with <b>\${vars}</b>

- 2) **Number (Para pasar string a Number('100')//100)** : Sirve para representar todos los números reales. Asi como existe el 0 y -0, pero ambos tienen el mismo valor (0 === -0 // true).

Los números con decimales tienen problemas a la hora de trabajar con ellos, por eso es importante usar la función **.toFixed(n)** para poder elegir cuántos decimales utilizar (esta devuelve un string, si ponemos el signo + antes de la operación nos da el dato en tipo number).

```

> var numero = 0.1 + 0.2;
< undefined
> numero
< 0.30000000000000004
> var numero = (0.1 + 0.2).toFixed(2);
< undefined
> numero
< '0.30'
> var numero = +(0.1 + 0.2).toFixed(2);
  numero
< 0.3

```

Para saber el número máximo y mínimo con los que podemos operar de forma segura en 64 bits, usamos **Number.MAX\_VALUE** y **Number.MIN\_VALUE**.

Para verificar si un valor es finito, utilizamos: **isFinite(n)** (isFinite(Infinity)//false)

**Infinity** y **-Infinity** son el resultado de dividir un valor por 0 o -0.

El último valor de tipo number es **NaN** y es el resultado de computos invalidos. NaN no es igual a nada, ni siquiera a sí mismo.

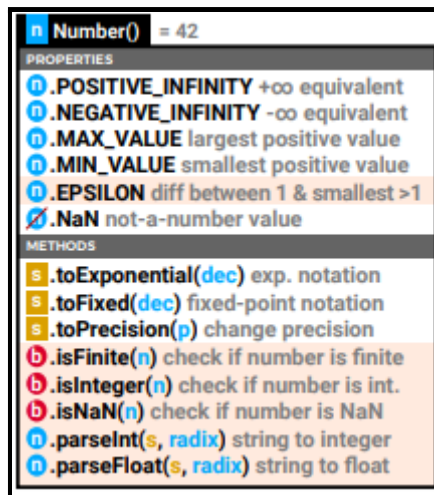
Para verificar si un valor es NaN, utilizamos: **isNaN(n)** (isNaN(NaN)//true)

```

> var a = 8;
< undefined
> Number.MAX_VALUE;
< 1.7976931348623157e+308
> Number.MIN_VALUE;
< 5e-324
> isNaN('hola');
< true
> a.toExponential(12); //(12) digitos despues de la coma
< '8.000000000000e+0'
> a.toPrecision(8); // valores despues de la coma
< '8.000000'
> isFinite(a);
< true

```

(toExponential y toPrecision)



(resumen de sus metodos)

- 3) **Boolean:** Solo puede tener dos valores, **true** o **false**. Podemos verlo como un interruptor que solo tiene dos estados, prendido o apagado. Los valores no booleanos pueden identificarse como tal en casos de validaciones de `if`, `for` o `while`.

**Valores falsos (además de false) :** `"`, `0`, `null`, `undefined` y `NaN`.

**Valores verdaderos (además de true) :** cualquier otro valor (`-1`, `1`, `'false'`, `obj{}`, `arr[]`);

```
> function mensajesSinLeer(number){
    if(number){
        console.log(`Tenes ${number} mensajes
sin leer.`)
    }else{
        console.log(`No tenes mensajes sin
leer.`)
    }
}
mensajesSinLeer(10);
mensajesSinLeer(0);
```

Tenes 10 mensajes sin leer.	<a href="#">VM1157:3</a>
No tenes mensajes sin leer.	<a href="#">VM1157:5</a>

Para convertir un valor a un boolean usamos `!!` (`!!" //false`, `!!'hola'//true`).

¿Por qué podemos usar métodos como `.toUpperCase()` en strings? Anteriormente dijimos que los datos primitivos no tienen métodos ni propiedades asociados. Cada vez que queremos llamar a un método con un valor primitivo (`str`, `number` o `boolean`), JS crea un

objeto temporal para poder ejecutar el atributo o método. A este objeto temporal se lo llama OBJECT WRAPPER, luego lo borra de la memoria

- 4) **Null**: Es el tipo de dato para representar la ausencia de valor. Sirve para decir que una variable no contiene nada, está vacía o que todavía no conocemos su valor.
- 5) **Undefined**: Significa tipo de dato desconocido, se le asigna automáticamente a las variables cuando las declaramos y no las inicializamos. Es distinto a null.  
(Existe un bug en JS, en donde typeof null devuelve object, esto no quiere decir que null sea un objeto ni que actúe como tal.)
- 6) **Symbol**: Se usa para crear valores únicos e irrepetibles. Para crearlos debemos llamar a la función Symbol, además de esto podemos agregarle una descripción pero solo servirá para entender mejor el código o cuando estemos buscando bugs. Podemos usar la misma descripción para varios Symbol() pero estos no serán iguales.

```
> var s = Symbol("descripcion");  
s  
◀ Symbol(descripcion)
```

(en este caso un symbol solamente es igual a si mismo)

A menos que lo creemos en el registro global de símbolos.

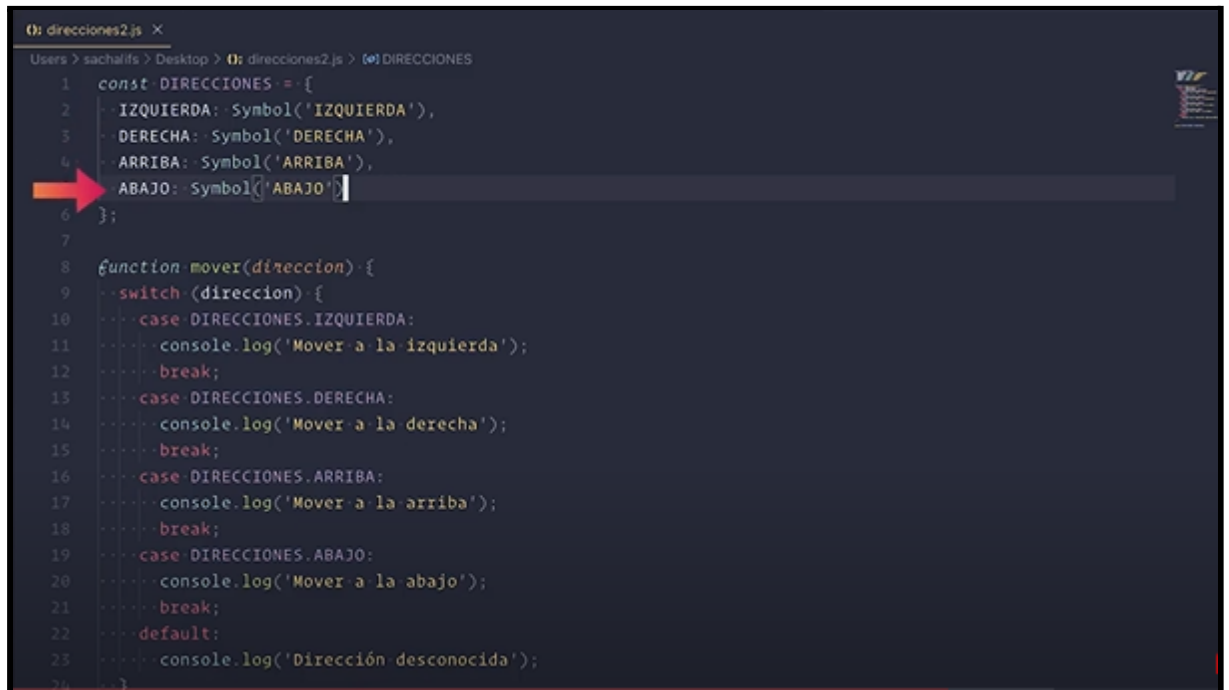
```
> var s1 = Symbol.for('descripcion1');  
var s2 = Symbol.for('descripcion2');  
s1;  
◀ Symbol(descripcion1)  
  
> s1 === s2;  
◀ false
```

De esta manera podremos acceder a un symbol en distintas partes de nuestro programa podemos hacerlo mediante su descripción. De manera inversa si tenemos un símbolo del registro global y queremos saber su descripción podemos usar **.keyFor(S)**.

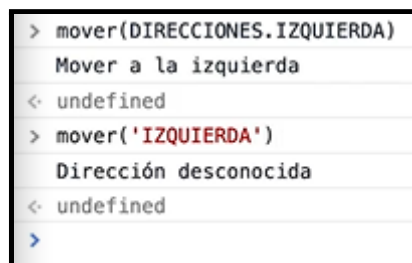
```
> var des = Symbol.keyFor(s1);  
◀ undefined  
  
> des  
◀ 'descripcion1'
```

**Los símbolos se usan generalmente para :**

\*) Valores constantes que podrían llegar a ser strings: De esta forma, deberíamos pasar a la función una dirección específica y no un string con la dirección. De esta forma, las direcciones pasan a ser el símbolo necesario para que la función se ejecute correctamente. (ejemplo de abajo)



```
1 const DIRECCIONES = {
2   IZQUIERDA: Symbol('IZQUIERDA'),
3   DERECHA: Symbol('DERECHA'),
4   ARRIBA: Symbol('ARRIBA'),
5   ABAJO: Symbol('ABAJO')
6 };
7
8 function mover(direccion) {
9   switch (direccion) {
10     case DIRECCIONES.IZQUIERDA:
11       console.log('Mover a la izquierda');
12       break;
13     case DIRECCIONES.DERECHA:
14       console.log('Mover a la derecha');
15       break;
16     case DIRECCIONES.ARRIBA:
17       console.log('Mover a la arriba');
18       break;
19     case DIRECCIONES.ABAJO:
20       console.log('Mover a la abajo');
21       break;
22     default:
23       console.log('Dirección desconocida');
24   }
25 }
```



```
> mover(DIRECCIONES.IZQUIERDA)
Mover a la izquierda
< undefined
> mover('IZQUIERDA')
Dirección desconocida
< undefined
>
```

\*) Para identificar propiedades únicas de un objeto y evitar colisiones entre nombres de las propiedades de los objetos.

Además podemos agregar propiedades ocultas a los objetos, que no aparezcan si precisamos `.key`, `.value` o `.entries`.

La forma de acceder a los símbolos es **`Objeto.getOwnPropertySymbols(objeto)`**.

**Símbolos destacados:** son propiedades (que a su vez son símbolos) de la función `Symbol` y permiten modificar el comportamiento de algunas funcionalidades del lenguaje.

**`Symbol.iterator`:** nos permitirá recorrer o expandir los arrays de atrás hacia adelante.

7) **BIGINT**: Permite utilizar números enteros sin límites ni errores de aproximación.

```
> var big = BigInt(999999999999);  
    var big2 = 999999999991n;  
  
< undefined  
  
> big  
< 999999999999n  
  
> big2  
< 999999999991n
```

(no podemos realizar sumas de enteros y bigint, así como tampoco .Math con bigint)

Todo valor que no sea de estos tipos, es un **objeto**(arrays, funciones, fechas, expresiones regulares).

### JAVASCRIPT ORIENTADO A OBJETOS

Para comenzar es importante definir el concepto de **herencia**, en JS, un objeto puede heredar propiedades y métodos de otro objeto (o clases), todos los objetos que creamos o están ya creados (todo aquello que no sea un dato primitivo, es un objeto), están referenciados a un objeto "proto{}", el cual posee sus propiedades y métodos. Cuando queramos acceder de un "objeto1", a una propiedad que no está definida dentro de él, JS buscará en el "proto" de este objeto, pudiendo acceder a ella en caso de que exista (a su vez los protos pueden estar referenciados a otros proto). Esto se llama Prototype Chain.

Sin embargo, hay un objeto especial llamado "Base Object", el cual no posee proto y se encuentra al final de la Prototype Chain, de todas maneras este base object posee definidas propiedades y métodos, accesibles por todos los objetos imaginables en JS (ya que esta al final del chain)

```

> var agus = {
  nombre : 'Agus',
  apellido: 'Castro',
  edad : 20,

  getNombre(){
    return `Me llamo ${this.nombre}
    ${this.apellido}`;
  }
}

var lucas = {
  nombre : 'Lucas',
  apellido : 'Montoto',
}

lucas.__proto__=agus;
< {nombre: 'Agus', apellido: 'Castro', edad: 20, getNombre: f}
> lucas.getNombre();
< 'Me llamo Lucas Montoto'

```

Ahora lucas tendra de \_\_proto\_\_ al objeto agus, por lo que podremos invocar sus metodos.

```

> let f = {};
  let b = [];
  let c = function(){};
< undefined
> f.__proto__
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
> b.__proto__
< [constructor: f, concat: f, copyWithin: f, fill: f, find: f, ...]
> c.__proto__
< f () { [native code] }

```

Proto de los objetos. A su vez podemos ver `f.__proto__.__proto__ // Object{}`

**Reflexion:** Reflexion es la capacidad que tienen los objetos de mirarse a sí mismos, listando y cambiando sus propiedades y métodos. Todos los objetos tienen un método llamado `.hasOwnProperty` el cual se fija solo en las propiedades del objeto llamado y no sigue el prototype chain.



En el ejemplo, el objeto lucas a pesar de tener de proto al objeto agus, con el uso de .hasOwnProperty() en el bucle for...in, revisará solo sus propiedades y valores, sin ver a su proto.

<pre>for(var key in lucas){   console.log(key + ':' + lucas[key]); }</pre>	
nombre:Lucas	<a href="#">VM2250:19</a>
apellido:Montoto	<a href="#">VM2250:19</a>
edad:20	<a href="#">VM2250:19</a>
getNombre:getNombre(){ return `Me llamo \${this.nombre} \${this.apellido}`; }	<a href="#">VM2250:19</a>
undefined	
<pre>for(var key in lucas){   if(lucas.hasOwnProperty(key)){     console.log(key + ':' + lucas[key]);   } }</pre>	
nombre:Lucas	<a href="#">VM2448:3</a>
apellido:Montoto	<a href="#">VM2448:3</a>

Mediante el uso de este concepto podemos hacer algo similar al prototipado pero con diferencias importantes.

## OPERADORES ARITMÉTICOS Y LÓGICOS

Utilizaremos los operadores matematicos para realizar operaciones entre valores, asi como tambien realizar comparaciones entre ellos, en su mayoría son los ya conocidos y utilizados en las matemáticas (+,-,\*,/,etc), pero no son los únicos, a continuación veremos ejemplos para aclarar sus usos:

<pre>&gt; var a = 2; //ult levels var b = 3;</pre>	
<	undefined
> a + b; //suma	
<	5
> a - b; //resta	
<	-1
> a * b; //producto	
<	6
> (a/b).toFixed(2); //division	
<	'0.67'
> b % 2; //resto	
<	1
> a ** b; //potencia	
<	8

### (operadores de asignación)

```
> a < b; // menor que...
< true
> a > b; // mayor que...
< false
> a <= b; // menor igual que...
< true
> a >= b; // mayor igual que...
< false
> a === b; // igual (tipo de dato y valor)
< false
> a !== b; // distinto
< true
```

### (operadores de comparacion)

```
> if(a < 0 || b > 2) true;
< true
> if (a >= 0 && b >= 0) false;
< false
```

### (operadores lógicos)

Tipos de notación para la utilización de operadores:

- 1) Infix =  $a + b / c$ ;
- 2) Prefix =  $+a * bc$ ;
- 3) Postfix =  $abc*+$ ;

### Precedencia de los operadores y asociatividad:

Los operadores poseen algo llamado precedencia, es decir un orden de importancia mediante el cual se van a ejecutar en cierto orden ( $3 + 2 * 2 = 7$ ).

Cuando la precedencia de los operadores es la misma (y no usamos parentesis para determinar el orden de ejecución) entra en juego la asociación right-to-left ( $a=1, b=2, c=3 \ \&\& \ a=b=c // 3$ ).

El siguiente link nos dirige a todos los operadores y sus respectivas precedencias:

([https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Operator_Precedence))

## OPERADORES CONDICIONALES TERNARIOS

Funciona al igual que in if, el operador ternario tiene tres operandos. Si la condición pasada es true, el operador tomará el valor de la expr1, del caso contrario tomará el valor de expr2.

**condición ? expr1 : expr2**

```
> function mayor(a,b){  
  return (a>b) ? `${a} es mayor que ${b}` : `${b} es mayor que  
  ${a}`  
}  
<> undefined  
> mayor(10,2)  
<> '10 es mayor que 2'  
> mayor(2,10)  
<> '10 es mayor que 2'
```

El operador condicional ternario es asociativo:

```
> var primero = 0;  
  var segundo = 0;  
  
  final = primero ? false : segundo ? false : true;  
<> true  
> final;  
<> true
```

## COERCIÓN DE DATOS

La coerción es el cambio automático de un dato en otro.

Algunos casos en los que puede existir una confusión:

- 1) **Suma, resta, producto, division y modulo de string:** Si tengo dos string '3' + '6' = 9; '33' - '3' = 30 (JS convierte automáticamente strings en números para las operaciones aritméticas)
- 2) **Number(false) // 0 && Number(true)//1; (10 - true = 9);**
- 3) **Number(undefined) // NaN**
- 4) **Number(null)//0;**

## VARIABLES VS REFERENCIAS

**Para datos primitivos:**

Para terminar de comprender el funcionamiento de las variables y de esta manera evitar bugs en nuestro programa, es importante entender que sucede cuando asignamos, re-asignamos y referenciamos valores de variables.

Para ponerlo en un ejemplo practico, si yo creo una variable y le asigno un valor (fruta 1="banana") y luego creo otra variable pasándole como valor la variable anterior (fruta 2=fruta 1), esta realiza una copia idéntica de la misma, podriamos decir que es como pasarle el mismo valor a ambas, puedo realizar cambios en alguna de ellas sin que la otra se inmute en lo mas minimo, vamos a verlo:

```
> var fruta1 = "banana";  
   var fruta2 = fruta1;  
  
   fruta1 = "mandarina";  
  
   fruta2;  
◀ 'banana'
```

(las variables son independientes)

**Para objetos, funciones y arrays:**

Los objetos no ocupan un espacio estático en la memoria del programa, estos pueden ir cambiando su tamaño y propiedades (estos se almacenan en el heap).

Cuando creamos un objeto, sus propiedades y valores no se guardan de manera literal como en los valores primitivos, lo que JS hace es buscar un espacio libre en el heap, y asignarle como valor la referencia de dirección donde este se almacena.

Supongamos que el siguiente objeto fue almacenado en la dirección A8F72 del heap:

```
var frutas = {  
  nombre = "banana",  
  cantidad = 3;  
}
```

Entonces frutas = A8F72, pero cuando lo llamamos, nos muestra directamente el objeto con lo que contiene adentro, sin manera sencilla de acceder a la dirección en el heap.

```

> var fruta = {
    nombre : 'banana',
    marca : 'tres anclas',
    cantidad : 3,
  }
var fruta2 = fruta;

fruta2.marca = 'serranita';

fruta;
< ▶ {nombre: 'banana', marca: 'serranita', cantidad: 3}
> fruta
< ▶ {nombre: 'banana', marca: 'serranita', cantidad: 3}

```

(a diferencia de los valores primitivos, al tener múltiples variables que referencian al mismo objeto, los cambios en propiedades se verán reflejados en ambas variables)

En el caso de funciones, podemos pasarle variables como parámetros, e incluso definir un valor fijo a esta variable sin que estos se pisén o generen errores entre sí, ya que son independientes y no hay manera de que se unan.

```

> function menosuno(cantidad){
    return cantidad - 1;
  }
< undefined
> var cantidad = 4;
< undefined
> menosuno(cantidad);
< 3
> menosuno(10)
< 9

```

(cantidad seguirá valiendo 4)

En caso de que pasemos objetos como parámetro de una función, lo que se copia es la referencia al objeto, por lo que veremos los cambios reflejados.

```
> var fruta = {  
    nombre : 'banana',  
    color : 'amarillo',  
}  
  
function changeName(fruta){  
    fruta.nombre = 'limon';  
    return fruta;  
}  
  
↵ undefined  
  
> changeName(fruta);  
↵ ▶ {nombre: 'limon', color: 'amarillo'}  
  
> fruta  
↵ ▶ {nombre: 'limon', color: 'amarillo'}
```

## FUNCIONES / FIRST CLASS FUNCTIONS

Las funciones son objetos, por lo que puedo acceder a propiedades de ellas como el code, el name, etc.

**Function.prototype** => El prototipo de la función

**Function.length** => El número de argumentos que espera la función.

**Function.name** => El nombre de la función.

```
> function funcion(a,b,c){
  if(a == true && b == true && c == true){
    console.log('Funcion simple de ejemplo');
  }else{
    console.log('Faltan parametros por ingresar');
  }
}
< undefined
> funcion(1,1,1);
Funcion simple de ejemplo VM2046:3
< undefined
> funcion(1,1)
Faltan parametros por ingresar VM2046:5
< undefined
> funcion.length;
< 3
> funcion.name;
< 'funcion'
> funcion.prototype;
< ▶ {constructor: f}
> funcion.prototype.getSaludo = function(){
  return 'Hola :) !';
}
< f (){
  return 'Hola :) !';
}
> funcion.prototype;
< ▶ {getSaludo: f, constructor: f}
```

Los métodos apply, bind y call estan explicados en el objeto this.

## CLOSURES

Las closures son, por así decirlo, dos funciones englobadas entre sí (una dentro de la otra), esto permite que una de ellas pueda referenciar al lexical environment de la otra.

Lo importante a tener en cuenta es el contexto de ejecución de las closures, el cual puede arruinarnos o ayudarnos si lo tenemos en claro.

Lo que las closures nos permiten, es guardar determinadas variables, para su uso posterior, a pesar de que el contexto de ejecución se haya eliminado, veámoslo:

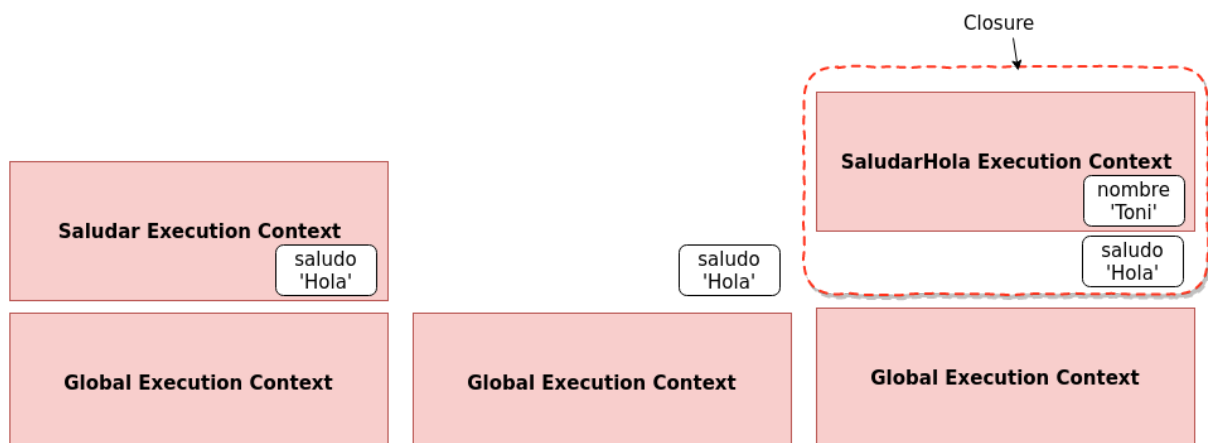
```
> function saludar(saludo){
  return function(nombre){
    console.log(`${saludo} ${nombre}`);
  }
}
< undefined

> var saludarHola = saludar('Hola');
< undefined

> saludarHola('Agus');
Hola Agus VM301:3
< undefined
```

En este caso la variable (saludo), se guarda en memoria (no así su contexto de ejecución), asociada a saludarHola(), entonces, ahora puedo invocar a saludarHola(), con diferentes parámetros pasados a ella.

```
> saludarHola('nico');
Hola nico
```



- 1) Se crea el contexto de ejecución global, donde se reserva espacio en memoria para la función y la variable saludarHola.
- 2) Almaceno en la variable, el valor de retorno de ejecución de saludar (function(nombre){console.log(...)}) con 'Hola' como saludo



- 3) Cuando alcanzó el return, se elimina el contexto de ejecución, pero queda saludo en memoria, la única forma de acceder y conocer 'saludo' es mediante la función interna que tenga (ya que conoce el lexical environment de quien la invocó)
- 4) Entonces el closure será: el contexto de ejecución de saludarHola(), con el parámetro que le pasemos y la variable (saludo) asociada a ella

Vamos a un ejemplo un poco más complejo:

```
> function deArgumentsAArr(){
  var arr = [];
  for(let i = 0; i < arguments.length; i++){
    if(arguments[i] > 0){
      arr.push(arguments[i])
    }
  }
  return function(base){
    return arr.map(element =>
Math.pow(element, base));
  }
}

var elevado =
deArgumentsAArr(-2, -1, 0, 1, 3, 5, 10);
< undefined

> elevado(2);
< ▶ (4) [1, 9, 25, 100]

> elevado(3);
< ▶ (4) [1, 27, 125, 1000]
```

En caso de que quisiera realizar la ejecución de toda la closure en una única línea, separo los argumentos de cada función con ().

```
> var elevao =
  deArgumentsAArr(-2, -1, 0, 1, 3, 5, 10)(2);
< undefined

> elevao
< ▶ (4) [1, 9, 25, 100]
```

Otro ejemplo: en este caso vamos a estar pusheado a un arreglo, funciones que `console.log`ean(i) del for. Es decir => `[function(){console.log(i)},[Function],[Function]]`;  
A la hora de invocar estas funciones (`arr[0]()`), cual sera nuestro output?!?! :

```
var creaFuncion = function(){
  var arreglo = [];

  for ( var i=0; i < 3; i++){
    arreglo.push(
      function(){
        console.log(i);
      }
    )
  }
  return arreglo;
}

var arr = creaFuncion();

arr[0]() // 3 sale un 3, qué esperaban ustedes??
arr[1]() // 3
arr[2]() // 3
```

Al declarar a `i` con un `var`, el valor se va realmacenando con cada iteración (`i=0 => i=1 => i=2` y `i=3`) del for, (cuando llega a `i=3`, reconoce que debe dejar de iterar y queda con ese valor). Al ejecutar `arr[0]()`, el contexto de ejecución local busca el valor de `i`, al no tenerlo, consulta en el lexical environment de la función que lo ejecuto si posee el valor, el cual esta almacenado en memoria (`i=3`) y es lo que se imprime en pantalla, en cambio:

Si yo declaro al '`i`' del for como un `let`, `arr[0]() // 0`, `arr[1]() // 1` y `arr[2]() // 2`, esto debido que cuando se consulta al lexical environment el valor de `i`, no se encuentra, ya que tanto `i` como su bloque de ejecución se destruyen al momento. Es por eso que JS asocia directamente el valor que vale `i` en cada contexto de ejecución, para cuando `i=0`, `i=1` e `i=2`:

```
> var creaFuncion = function(){
  var arreglo = [];

  for ( let i=0; i < 3; i++){
    arreglo.push(
      function(){
        console.log(i);
      }
    )
  }
  return arreglo;
}

var arr = creaFuncion();

arr[0]();
arr[1]();
arr[2]();
```

0	<a href="#">VM1997:7</a>
1	<a href="#">VM1997:7</a>
2	<a href="#">VM1997:7</a>

Otra forma de ver y hacer esto:

```
> var creaFuncion = function(){
    var arreglo = [];

    for ( var i=0; i < 3; i++){
        arreglo.push(
            (function(j){
                console.log(j);
            })
        )(i)); //de esta forma la funcion se ejecuta automaticamente al momento
    }
    return arreglo;
}

var arr = creaFuncion();

0 VM2187:7
1 VM2187:7
2 VM2187:7
< undefined
> arr;
< ▶ (3) [undefined, undefined, undefined]
```

La inmediatamente function box se ejecuta al instante e imprime los valores de i, sin pushear nada al array.

Otro ejemplo aparte:

```
function hacerSaludo( lenguaje ){
    return function(){
        if ( lenguaje === 'en'){
            console.log('Hi!');
        }

        if ( lenguaje === 'es'){
            console.log('Hola!');
        }
    }
}

var saludoIngles = hacerSaludo('en');
var saludoEspaniol = hacerSaludo('es');
```

Una forma mas:

En el ejemplo siguiente, si logramos pushear al array la immediately function box, para que pueda ser ejecutada, pasandole j como parametro, el cual sera el valor de i al momento de cada iteración.

Es una forma de hacer lo mismo que usando let, pero con closures y var.

```
> var creaFuncion = function(){
  var arreglo = [];

  for ( var i=0; i < 3; i++){
    arreglo.push(
      (function(j){
        return function(){console.log(j)};
      })(i))
  }
  return arreglo;
}

var arr = creaFuncion();
< undefined
> arr[0]();
0
VM2408:7
```

## CLOSURES & CALLBACKS

Texto extraido y modificado de FT-M1/README.md :

Ahora que sabemos lo que son los closures, si pensamos en todo lo que hicimos algunas vez con JS, es muy probable que nos demos cuenta que ya lo veníamos usando.

```
> function saludarMasTarde(){
  var saludo = 'Hola';

  setTimeout( function(){
    console.log(saludo);
  },3000)
};

saludarMasTarde();
|
```

En el ejemplo de arriba, cuando invocamos a `saludarMasTarde` estamos creando un execution context, en el que invocamos a la función `setTimeout` y donde definimos la variable `saludo`. Ese execution context es destruido, pero `setTimeout` contiene una referencia a `saludo`. Closure, Maybe? Lo que realmente ocurre es que cuando pasan los tres segundos, se lanza un evento diciendo que hay que ejecutar el callback, que es justamente una función expression. Entonces se crea un execution context para esa función, y dentro de ella se usa `saludo`, pero no está en ese contexto, entonces el interprete sale a buscarla afuera, y la encuentra en el closure!

### CONCEPTOS A RECORDAR PARA UN BUEN USO DE CLOSURES:

- 1) Solo puede existir un único return accesible por cada función (lo que no quiere que deba haber un solo return en todo el código de la closure).
- 2) Tener muy presente el scope de cada función de la closure para evitar errores .
- 3) Let y closures generan conflictos, son poco compatibles en casos de uso de bucles, ya que let se destruye junto a su bloque de ejecución.

### RECUSIÓN (función que se llama a si misma)

La idea de la recursión es hacer una especie de 'loop' entre los llamados de función, cuando la genero esta se auto invoca hasta llegar al resultado a retornar.

Tenemos que tener mucho cuidado y prestar suma atención a los llamados a la función. En la recursión, las funciones nunca llegan a terminar de ejecutarse que ya se auto-invocan nuevamente (hasta llegar al valor final).

Cuando en el problema nos presentan una acción que se debe repetir (de binario a decimal por ej), la idea es apuntar a la recursividad.

```
> function factorial(num){  
  if(num === 1 || num === 0) return 1;  
  if(num < 0 )return 0;  
  
  return num * factorial(num - 1);  
}  
↵ undefined  
> factorial(5)  
↵ 120
```

Call Stack: algo así debería verse la pila de ejecución, al ejecutar `factorial(5)`, la función queda detenida en `return num * factorial(num - 1)`, la cual añade una nueva tarea a la pila (`//factorial(4)`), hasta que esta no termine de ejecutarse, `factorial(5)` no puede completar su resultado (ya que tenemos `5 * function //invalido`) y así sucesivamente hasta que se cumpla alguna condición del principio.

```
//factorial(1) = 1
//factorial(2) = 2*factorial(1)
//factorial(3) = 3*factorial(2)
//factorial(4) = 4*factorial(3)
//factorial(5) = 5*factorial(4)
```

Cuando se cumple la condición y uno de los llamados arroja un resultado, los contextos de ejecución van siendo eliminados uno a uno, pero con el resultado guardado, veamos:

```
//factorial(1) = 1 =>se elimina
//factorial(2) = 2 *1
```

y la pila quedara algo asi:

```
//factorial(2) = 2 *1
//factorial(3) = 3*factorial(2)
//factorial(4) = 4*factorial(3)
//factorial(5) = 5*factorial(4)
```

como te podrás imaginar, esto continua!!

```
//factorial(2) = 2*1 => se elimina
//factorial(3) = 3*2 => se elimina
//factorial(4) = 4*6 => se elimina
//factorial(5) = 5*24 => ahora si arroja un resultado final y puede eliminarse
```

```
//factorial(5) // 120
```

y la pila de ejecución quedara vacía.

Vamos a otro ejemplo:

```
> function decimalABinario(num){
    if(num === 1) return '1';

    return decimalABinario(Math.floor(num/2)) + (num%2);
}
```

Pila de ejecución:

```
decimalABinario(3) = decimalABinario(1) + 1%2
decimalABinario(7) = decimalABinario(3) + 3%2
```

```
decimalABinario(15) = decimalABinario(7) + 15%2  
decimalABinario(30) = decimalABinario(15) + 30%2
```

El primer valor que tendremos sera decimalABinario(1) = '1' (por el if), entonces:

```
decimalABinario(3) = '1' + 1%2 = '11' => se elimina  
decimalABinario(7) = '10' + 3%2 = '111' => se elimina  
decimalABinario(15) = '101' + 15%2 = '1111' => se elimina  
decimalABinario(30) = '1011' + 30%2 = '11110' => se elimina y finaliza
```

### EXPRESIONES REGULARES

Las expresiones regulares son patrones que se utilizan para hacer coincidir combinaciones de caracteres en cadenas. Las podemos utilizar para extraer, validar, verificar, etc datos de una string.

Los métodos que requieren de expresiones regulares son:

***exec(), test(), RegExp;***

***match(), matchAll(), replace(),replaceAll(), search() y split(),*** metodos de string

Para construirlas:

**\*De manera literal:** patron encerrado entre barras (su uso puede mejorar el rendimiento)

**let ExpReg = /patron/banderas;**

**\*Con una función constructora:**

**let ExpReg = new RegExp(patron,banderas);**

```
{  
let findA = new RegExp("a",'g','i')  
let re = /ab+c/ig;  
}
```

```

var cadena = "Lorem Ipsum es un texto de
marcador de posición comúnmente utilizado en
las industrias gráficas, gráficas y
editoriales para previsualizar diseños y
maquetas visuales. "

let findA = new RegExp("a",'g','i');

console.log(findA.test(cadena));
console.log(findA.exec(cadena));

```

true VM72:5

VM72:6

```

['a', index: 31, input: 'Lorem Ipsum es un
texto de marcador de posición co... para prev
isualizar diseños y maquetas visuales. ', g
roups: undefined]

```

### **Morfología:**

#### **Alternación:**

Una barra vertical separa las alternativas, las cuales son evaluadas de izquierda a derecha. Por ejemplo, "amarillo|azul" se corresponde con amarillo o azul.

#### **Cuantificación:**

Un cuantificador tras un carácter especifica la frecuencia con la que este puede ocurrir. Los cuantificadores más comunes son "?", "+" y "\*":

? = indica que el carácter que le precede puede aparecer como mucho una vez. Por ejemplo, "ob?scuro" se corresponde con oscuro y obscuro.

+ = indica que el carácter que le precede debe aparecer al menos una vez. Por ejemplo, "ho+la" describe el conjunto infinito hola, hoola, hoolola, hooooola, etcétera.

\* = indica que el carácter que le precede puede aparecer cero, una, o más veces. Por ejemplo, "0\*42" se corresponde con 42, 042, 0042, 00042, etcétera.

#### **Agrupación:**

() = pueden usarse para definir el ámbito y precedencia de los demás operadores. Por ejemplo, "(p|m)adre" es lo mismo que "padre|madre", y "(des)?amor" se corresponde con amor y con desamor.



Los constructores pueden combinarse libremente dentro de la misma expresión, por lo que "H(ae?|ä)ndel" equivale a "H(a|ae|ä)ndel".

```
> var cadena = "abcabcabcaBBBBBc"

let findA = /ab*c/ig;

console.log(cadena.match(findA));
console.log(findA.exec(cadena));
```

---

```
▶ (4) ['abc', 'abc', 'abc', 'aBBBBBc']
```

---

```
▶ ['abc', index: 0, input: 'abcabcabcaBBBBBc', groups: undefined]
```

---

```
> var cadena = "amor";

let ExpReg = new RegExp('(des)?amor');

console.log(ExpReg.test(cadena));
```

---

```
true
```

**.** (punto) = se interpreta como “cualquier carácter” incluyendo los saltos de línea. Por ejemplo si pedimos que se busque 'g.t' de “gato,geto,gito,goot”, encontrara gat,get y git, pero goot no, ya que el puto representa un solo caracter.

**!** (signo de admiración) = se utiliza para buscar una “búsqueda anticipada negativa”. Por ejemplo, para excluir exactamente una palabra, habrá que utilizar `^(palabra.+|(?!palabra).*)$`

**\** (barra inversa) = La barra inversa no se utiliza nunca por sí sola, sino en combinación con otros caracteres. Al utilizarlo por ejemplo en combinación con el punto “\.” este deja de tener su significado normal y se comporta como un carácter literal.

Combinaciones especiales :

- `\t` — Representa un tabulador.
- `\r` — Representa el "regreso al inicio" o sea el lugar en que la línea vuelve a iniciar.

- `\n` — Representa la "nueva línea" el carácter por medio del cual una línea da inicio.
- `\a` — Representa una "campana" o "beep" que se produce al imprimir este carácter.
- `\e` — Representa la tecla "Esc" o "Escape"
- `\f` — Representa un salto de página
- `\v` — Representa un tabulador vertical
- `\x` — Se utiliza para representar caracteres [ASCII](#) o ANSI si conoce su código.
- `\u` — Se utiliza para representar caracteres [Unicode](#) si se conoce su código.
- `\d` — Representa un dígito del 0 al 9.
- `\w` — Representa cualquier carácter [alfanumérico](#).
- `\s` — Representa un espacio en blanco.
- `\D` — Representa cualquier carácter que no sea un dígito del 0 al 9.
- `\W` — Representa cualquier carácter no alfanumérico.
- `\S` — Representa cualquier carácter que no sea un espacio en blanco.
- `\A` — Representa el inicio de la cadena. No un carácter sino una posición.
- `\Z` — Representa el final de la cadena. No un carácter sino una posición.
- `\b` — Marca la posición de una palabra limitada por espacios en blanco, puntuación o el inicio/final de una cadena.
- `\B` — Marca la posición entre dos caracteres alfanuméricos o dos no-alfanuméricos.

| (**la barra**) = Sirve para indicar una de varias opciones.

**\$ (dolar)** = representa el final de una cadena de texto. No representa un carácter especial sino una posición. Si se usa `\.$` representa todos los lugares donde un punto finalice una línea.

**^ (acento circunflejo)** = Si esta solo representa el inicio de la cadena. Por ejemplo `^[a-z]` el motor encontrará todos los párrafos que den inicio con una letra minúscula. Cuando se utiliza en conjunto con los corchetes de la siguiente forma `[^\w ]` permite encontrar cualquier carácter que NO se encuentre dentro del grupo indicado.

**{}** = Representan una cantidad de repeticiones de una expresión. Por ejemplo `"d{2}"` Esta expresión le dice al motor de búsqueda que encuentre dos dígitos contiguos. Utilizando esta fórmula podríamos convertir el ejemplo `"^d\d/d\d/d\d\d\d$"` que servía para validar un formato de fecha en `"^d{2}/d{2}/d{4}$"` para una mayor claridad en la lectura de la expresión.

**[]** = Representa un listado literal, `[0-9]` por ejemplo.

## CICLOS O BUCLES

Utilizamos los bucles para realizar una tarea repetidamente de forma sencilla, ya sea comprobar valores, recorrer variables, o pushear valores en un array (los bucles pueden trabajar en conjunto con todos los métodos, objetos, y definiciones que venimos trabajando):

```
> var obj = {
  numeros : [],
  suma : 'Mediante el operador +',
  resta : 'Mediante el operador -',
}
< undefined
> for(var i=0;i<=10;i++){
  obj.numeros.push(i);
}
< 11
> obj.numeros;
< ▶ (11) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

**Bucle FOR:** Este ciclo se repite hasta que una condición especificada se evalúe como falsa. Este se compone por una variable, una condición de ejecución y por último un actualizador de la variable que va a manejar nuestro bucle.

-for(var i = 0; i<10; i++);

### Buenas prácticas en el uso del bucle for:

\*) Cuando recorro un array y le indico que `i<arr.length`, esto es un gasto de memoria ya que se analiza por cada iteración, además del acceso al array dentro del objeto:

```
> const obj = {
  unArray : new Array(10000),
}
< undefined
> const array = obj.unArray;
< undefined
> for(let i=0, longitud=array.length;i<longitud;i++){
  array[i]="Buenas practicas";
}
< 'Buenas practicas'
> obj
< ▼ {unArray: Array(10000)} ⓘ
  ▶ unArray: (10000) ['Buenas practicas', 'Buenas practicas', '...'
  ▶ [[Prototype]]: Object
```

(con el uso de `'`, inicializo la longitud al lado de `i`)

**Bucle DO...WHILE:** Este ciclo itera siempre y cuando la condición evaluada en while sea verdadera.

```
> function dowhile(){
  let i = 0;
  var arr = [];
  do{
    arr.push(i);
    i = i+1;

  }while(i<=10);
  return arr;
}
< undefined
> dowhile();
< ▶ (11) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

(al igual el bucle while(){}, funciona como el mismo do...while)

**Bucle FOREACH:** Bucle creado específicamente para recorrer arrays, en el argumento de este método debemos pasar una función, la cual recibe como parámetros un item(elemento de un arr) y un índice(indice del elemento).

Al igual que es bucle for, tenemos **buenas practicas** para foreach, como por ejemplo reemplazar la función pasada como parametro por una **arrow function**.

```
> const arr = [1,2,3,4];
< undefined
> arr.forEach((item,index)=>{
  console.log(`El indice del item: ${item}, es ${index}`)});
El indice del item: 1, es 0 VM260:2
El indice del item: 2, es 1 VM260:2
El indice del item: 3, es 2 VM260:2
El indice del item: 4, es 3 VM260:2
```

## ARRAYS

<b>a</b> <b>Array()</b> = [1, 2, 3] <b>PROPERTIES</b> <b>n</b> .length number of elements <b>METHODS</b> <b>b</b> .isArray(obj) check if obj is array <b>b</b> .includes(obj, from) include element? <b>n</b> .indexOf(obj, from) find elem. index <b>n</b> .lastIndexOf(obj, from) find from end <b>s</b> .join(sep) join elements w/separator <b>a</b> .slice(ini, end) return array portion <b>a</b> .concat(obj1, obj2...) return joined array <b>a</b> .flat(depth) return flat array at n depth <b>MODIFY SOURCE ARRAY METHODS</b> <b>a</b> .copyWithin(pos, ini, end) copy elems <b>a</b> .fill(obj, ini, end) fill array with obj <b>a</b> .reverse() reverse array & return it <b>a</b> .sort(cf(a,b)) sort array (unicode sort) <b>a</b> .splice(ini, del, o1, o2...) del&add elem <b>ITERATION METHODS</b> <b>a</b> .entries() iterate key/value pair array <b>a</b> .keys() iterate only keys array <b>a</b> .values() iterate only values array	<b>CALLBACK FOR EACH METHODS</b> <b>b</b> .every(cb(e,i,a), arg) test until false <b>b</b> .some(cb(e,i,a), arg) test until true <b>a</b> .map(cb(e,i,a), arg) make array <b>a</b> .filter(cb(e,i,a), arg) make array w/true <b>o</b> .find(cb(e,i,a), arg) return elem w/true <b>n</b> .findIndex(cb(e,i,a), arg) return index <b>a</b> .flatMap(cb(e,i,a), arg) map + flat(1) <input checked="" type="checkbox"/> <b>.forEach(cb(e,i,a), arg)</b> exec for each <b>o</b> .reduce(cb(p,e,i,a), arg) accumulative <b>o</b> .reduceRight(cb(p,e,i,a), arg) from end <b>ADD/REMOVE METHODS</b> <b>o</b> .pop() remove & return last element <b>n</b> .push(o1, o2...) add elem & return length <b>o</b> .shift() remove & return first element <b>n</b> .unshift(o1, o2...) add elem & return len
---	--

Utilización de los métodos (todos los métodos son compatibles con arrays incluidos en objetos) :

**arr.length**: devuelve el número de elementos

```
> var arr = [1,2,3,4,5]
< undefined
> arr.length;
< 5
```

**arr.indexOf(element)** : devuelve el índice donde se encuentra el elemento, si no esta arroja -1

```
> arr.indexOf(5);
< 4
```

**arr.lastIndexOf(element)** : en esencia funciona igual que .indexOf(), solo que nos devuelve la última posición en donde se encuentra el elemento

```
> [1,2,3,4,1].lastIndexOf(1);
< 4
```

**join(' '), ('-'),(''),etc**: devuelve un string con los elementos del array concatenados.

```
> ['Hola', 'Mi', 'Nombre', 'Es', 'Agus'].join(' ( Hola ) ');
< 'Hola( Hola )Mi( Hola )Nombre( Hola )Es( Hola )Agus'
```

**slice():** devuelve una copia del array dentro de uno nuevo, no modifica el array original (no es un método destructivo):

```
> var arr = [1,2,3,4,5];
< undefined
> arr.slice(1,3);
< ▶ (2) [2, 3]
> arr
< ▶ (5) [1, 2, 3, 4, 5]
```

**concat(obj1,obj2) || (arr1,arr2) :** Con este método podemos unir dos o mas arrays sin reemplazar los arrays existentes (método no destructivo).

```
> var arr1 = [1,2,3,4,5];
  var arr2 = [0];
  var arr3 = [-5,-4,-3,-2,-1];
< undefined
> var arrfinal = [].concat(arr3,arr2,arr1);
< undefined
> arrfinal;
< ▶ (11) [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
```

**copyWithin(target a cambiar, start,end):** este método realiza una copia plana de un índice a otro dentro del mismo array, sin modificar su length.

```
> var arr = [1,2,3,4,['Hola',null],undefined].copyWithin(0,4,5);
< undefined
> arr;
< ▼ (6) [Array(2), 2, 3, 4, Array(2), undefined] ⓘ
  ▶ 0: (2) ['Hola', null]
    1: 2
    2: 3
    3: 4
  ▶ 4: (2) ['Hola', null]
    5: undefined
    length: 6
  ▶ [[Prototype]]: Array(0)
```

En este caso copie el array en posición 4 del array, a la posición 0.

**fill(value para rellenar,start,end):** con este método podemos reemplazar todos los elementos de un array desde un inicio hasta un fin por un valor estático.

```
> var arr = [1,2,3,4,['Hola',null],undefined].fill('420',0,4);
< undefined
> arr;
< ▶ (6) ['420', '420', '420', '420', Array(2), undefined]
```

**reverse():** con este método podemos invertir los elementos de un array.

```
> console.log(['estas!', 'como', 'Hola'].reverse());  
▶ (3) ['Hola', 'como', 'estas!']
```

```
> var obj = {  
  a : 1,  
  b : 2,  
  c : 3,  
}  
◀ undefined  
> obj.d = [1,2,3,4,5];  
◀ ▶ (5) [1, 2, 3, 4, 5]  
> obj.d.reverse();  
◀ ▶ (5) [5, 4, 3, 2, 1]  
> obj  
◀ ▼ {a: 1, b: 2, c: 3, d: Array(5)} ⓘ  
  a: 1  
  b: 2  
  c: 3  
  ▶ d: (5) [5, 4, 3, 2, 1]  
  ▶ [[Prototype]]: Object
```

**sort(first element, second element):** el uso de sort nos permite ordenar los elementos de un array. Cuando no pasamos parámetros, se ordena según el valor de UNICODE.

- 1) Si son strings en minúsculas, se ordena alfabéticamente. Si hay mayúsculas, se posicionarán al principio las mayúsculas sin importar su orden en el alfabeto en relación a las minúsculas
- 2) Si son números, se ordena de menor a mayor, pero sin tener en cuenta el dígito completo, es decir, si tengo [20,1,3], se ordenará [1,20,3].
- 3) Los números van por delante de las letras, teniendo en cuenta la coerción.

```
> var arr = [3,20,1,'0 palabra', 'a', 'B', 'b', 'c'];  
◀ undefined  
> arr.sort();  
◀ ▶ (8) ['0 palabra', 1, 20, 3, 'B', 'a', 'b', 'c']
```

**splice(start, delete count, item1,item2):** cambia el contenido de un array eliminando elementos de un array, a su vez podemos agregar nuevos elementos en esas posiciones eliminadas.

```
> var arr = [3,20,1,'0 palabra','a','B','b','c'];
  arr.splice(0,4,'a','e','i','o');
< ▶ (4) [3, 20, 1, '0 palabra']
> arr
< ▶ (8) ['a', 'e', 'i', 'o', 'a', 'B', 'b', 'c']
```

Podemos guardar los valores eliminados en un nuevo array para usarlos en otro momento.

**entries(),keys() y values():** Estos métodos devuelven lo que sus respectivos nombres determinan, se complementan con el uso de **next()** para su correcto funcionamiento.

```
> var arr = ['a','e','i','o','u'];
< undefined
> var iterador = arr.entries(); // creamos una variable para el
  iterador
< undefined
> var arr2 = [];
< undefined
> arr2.push(iterador.next().value);
< 1
> arr2;
< ▶ [Array(2)]
> // podemos seguir pusheando el iterador hasta arr.length;
< undefined
> arr2.push(iterador.next().value);
< 2
> arr2.push(iterador.next().value);
< 3
> arr2.push(iterador.next().value);
< 4
> arr2;
< ▼ (4) [Array(2), Array(2), Array(2), Array(2)] ⓘ
  ▶ 0: (2) [0, 'a']
  ▶ 1: (2) [1, 'e']
  ▶ 2: (2) [2, 'i']
  ▶ 3: (2) [3, 'o']
  length: 4
  ▶ [[Prototype]]: Array(0)
```

Si pusheamos el iterador más veces que `arr.length`, solo pusheara `undefined`.



Esto podemos combinarlo con un `for...of` para imprimir todos los pares `entries()` del array.

`.keys()` devolverá los índices y `.values()` los elementos.

**find(callback)** : para este ejemplo supongamos que tenemos un array de objetos.

```
const posts = [{
  id:1,
  title:"Primer post",
  image:'https://img.com/1',
  tags : ['javascript','webdevelopment'],
},{
  id:2,
  title:"Mi experiencia con React",
  image:'https://img.com/2',
  tags : ['javascript','webdevelopment','react']
},
,{
  id:3,
  title:"Por que deje Angular",
  image:'https://img.com/3',
  tags : ['javascript','webdevelopment',
    'angular'],
}]
```

Con el método `find` podremos recorrer los objetos del array en busca de algo de nuestro interés, por ejemplo, queremos que nos devuelva aquellos objetos que tengan como título 'Por que deje angular':

```
> posts.find(post => post.title == "Por que deje Angular");
< {id: 3, title: 'Por que deje Angular', image: 'https://img.com/3', tags: Array(3)}
```

**some()** : devuelve `true` o `false` dependiendo de si existe o no lo que estamos buscando (usaremos el mismo array 'posts' del ejemplo anterior).

```
> posts.some(post => post.id == 2);
< true
```

a esto lo podemos combinar de manera muy útil con **.includes()**;

```
> posts.some(post => post.tags.includes('views'));
< false
> posts.some(post => post.tags.includes('javascript'));
< true
```

recorremos el array 'posts' y le especificamos que, si existe algún objeto del array que en su propiedad `tags` tenga una palabra especificada, nos devuelva `true`, en caso contrario devolverá `false`.

**every()** : comprueba si algo se cumple en todas las instancias especificadas:

```
> posts.every(post => post.tags.includes('javascript'));  
< true  
  
> posts.every(post => post.tags.includes('angular'));  
< false
```

**map()** : es una especie de bucle for, pero de una manera más declarativa (es decir, especificando **que** se quiere hacer). Seguimos usando el array 'posts'.

El método map ejecuta una función o condición sobre cada elemento del array y devuelve un nuevo array con los elementos convertidos.

Por ejemplo, quiero hacer un nuevo array, el cual posea únicamente los 'title' de cada objeto, para esto:

```
> posts.map(post => post.title);  
< ▶ (3) ['Primer post', 'Mi experiencia con React', 'Por que deje Angular']  
  
>  
  let array = [1,2,3,4,5];  
< undefined  
  
> array.map(elemento => elemento*2);  
< ▶ (5) [2, 4, 6, 8, 10]
```

**filter():**

## OBJETOS

Es común escuchar que javascript es un lenguaje basado en objetos, esto aunque no es real al 100%, si tiene mucha razón.

Podemos decir que un objeto es una colección de propiedades con un valor asociado a ellas, permiten generar listas mas complejas.

**Maneras de construir objetos:**

### 1) Objetos literales:

```
var agus = {  
  nombre : 'Agus',  
  apellido: 'Castro',  
  edad : 20,  
  
  getNombre(){  
    return `Me llamo ${this.nombre}  
    ${this.apellido}`;  
  }  
}
```

## 2) Funciones constructoras y new :

Para entenderlo, primero debemos tener en cuenta que new es un operador, que, en primer lugar crea un objeto vacío, luego invoca la función que le pasamos como argumento, con la particularidad que bindea el nuevo objeto vacío que había creado, de tal forma que this haga referencia a el. Por ultimo retorna el objeto.

```
> function Persona(nombre,apellido,edad){
  this.nombre = nombre || 'N';
  this.apellido = apellido || 'N';
  this.edad = edad || 'Sin edad';
}
< undefined
> var Agus = new Persona('Agustin','Castro',20);
  var Cris = new Persona('Cristian','Castro',20);
< undefined
> Agus;
< ▶ Persona {nombre: 'Agustin', apellido: 'Castro', edad: 20}
> Cris;
< ▶ Persona {nombre: 'Cristian', apellido: 'Castro', edad: 20}
```

Si queremos ver qué objeto tiene seteado como proto las funciones creadas a partir de una 'function constructor', nos damos cuenta que es el mismo constructor. A su vez las funciones tienen una propiedad llamada "prototype" y que se setea como un objeto vacío. Cuando invocamos la función con el operador new, la propiedad prototype de la función va a ser usada como el proto de todos los objetos que hayan sido creadas con ella, por lo tanto, todo lo que pongamos dentro de la propiedad prototype de la funcion constructora , va a ser heredado por los objetos creados usando la función con new.

```

> function Persona(nombre,apellido,edad){
  this.nombre = nombre || 'N';
  this.apellido = apellido || 'N';
  this.edad = edad || 'Sin edad';
}

Persona.prototype.getName = function(){
  return `Me llamo ${this.nombre} ${this.apellido}`
}

Persona.prototype.getEdad = function(){
  return this.edad;
}

var Agus = new Persona('Agustin','Castro',20);
var Cris = new Persona('Cristian','Castro',20);
< f (){
  return this.edad;
}

> Agus.getName();
< ' Me llamo Agustin Castro'

> Cris.getEdad();
< 20

```

Ahora Agus, `__proto__` // {getName: f, getEdad: f, constructor: f} al igual que cris.

### 3) Object.create y pure prototypal inheritance:

```

> var person = {
  nombre: '',
  apellido: '',
  edad: null,
  getAge(){
    return this.edad;
  }
}
< undefined

> var Agus = Object.create(person);
< undefined

> Agus.nombre = 'Agus';
Agus.apellido = 'Castro';
Agus.edad = 20;
< 20

> Agus.getAge()
< 20

> Agus.__proto__
< {nombre: '', apellido: '', edad: null, getAge: f}

```

Object.create recibe un objeto base como parametro y crea un nuevo objeto cuyo prototipo es el objeto base que le pasamos. Si queremos asignarle valores a las propiedades debemos hacerlo manualmente una por una.

**This en objetos** tomara como valor el objeto en el cual lo estemos ejecutando, siempre y cuando tengamos cuidado que tipo de bindeo se este ejecutando.

Si quiero que el this tenga correctamente el valor del objeto donde lo estoy invocando, debo declarar las funciones o métodos dentro del mismo y no de forma suelta, ya que en ese caso this hara referencia al objeto global.

```
> var obj = {
  nombre : 'objeto',
  log(){
    this.nombre = 'Cambiado';
    console.log(this)
    var that = this;
    var cambia= function(str){
      that.nombre = str;
    };
    cambia('hola!!');
    console.log(this)
  }
}
< undefined
> obj.log()
  ▶ {nombre: 'Cambiado', log: f}
  ▶ {nombre: 'hola!!', log: f}
```

(para analizar el valor de this)

```
> const obj = {
  nombre : 'Agustin',
  apellido : 'Castro',
  edad : 20,
  colorfav : 'verde',
  familiares : [{
    nombre : 'Yanina',
    parentesco : 'Madre',
  },
  {
    nombre : 'Cristian',
    parentesco : 'Hermano',
  }],
  getSaludo(){
    console.log(`Hola ${this.nombre}`);
  }
}
< undefined
> obj
< {nombre: 'Agustin', apellido: 'Castro', edad: 20, colorfav: 'verde', familiares: Array(2), ...}
> obj.getSaludo();
Hola Agustin VM925:14
< undefined
> obj['nombre'] = 'Francisco Agustin';
< 'Francisco Agustin'
> obj.getSaludo();
Hola Francisco Agustin VM925:14
```

Un arreglo permite todo tipo de datos.

Si quiero cambiar el 'key' de una propiedad la cual conozco el nombre puedo:

**Obj['nombre']='nuevo nombre'; (puedo poner funciones por ejemplo)**

Para acceder a un valor del objeto:

**Obj.nombre;**

Si quiero añadir una nueva propiedad puedo:

**Obj.numerofav = 7; o Obj['comidafav']=pure;**

Para eliminar una propiedad de un objeto:

**delete obj.nombre; o delete obj['nombre'];**

### OBJETO GLOBAL MATH

Math es un objeto incorporado en el motor de JS y posee propiedades, métodos y funciones, usaremos Math para simplificar nuestro código a la hora de realizar ciertas operaciones matemáticas estáticas. (Math no funciona con BigInt).

A diferencia de los demás objetos globales, Math no se puede editar, todas sus propiedades y métodos son estáticos. Las constantes se definen con la precisión completa de los números reales.

**Propiedades estáticas (deben usarse como tal, no requieren valores ingresados):**

- 1) Math.E
- 2) Math.LN2 / Math.LN10 / Math.LOG2E / Math.LOG10E

- 3) Math.PI
- 4) Math.SQRT2 / Math.SQRT1\_2

**Métodos** ( tener en cuenta que las funciones trigonometricas devuelven angulos en radianes, para pasarlos a grados  $\Rightarrow$   $(\text{Math.sin}(200) * (\text{Math.Pi} / 180))$ )

**De menor importancia** : Math.acos(x), Math.acosh(x), Math.asin(x), Math.asinh(x), Math.atan(x), Math.atanh(x), Math.atan2(y, x), Math.cos(x), Math.exp(x), Math.expm1(x), Math.log(x), Math.log10(x), Math.log2(x), Math.random(x), Math.sin(x), Math.tan(x) y Math.tanh(x)

**De mayor uso e importancia:**

```
> Math.abs(-2);
< 2

> Math.cbrt(27);
< 3

> Math.ceil(5.00000001); // redondea hacia arriba
< 6

> Math.floor(4.9999999); // redondea hacia abajo
< 4

> var array = [-4,2,-2,4,7,10];
  Math.max.apply(null,array);
< 10

> Math.max(1,4,7);
< 7

> Math.min.apply(null,array);
< -4

> Math.pow(4,2) // potencia (base,exponente)
< 16

> Math.round(7.6) // redondea al mas cercano
< 8

> Math.sign(-8) // devuelve el signo del parametro pasado
< -1

> Math.sqrt(4); // devuelve la raiz cuadrada
< 2
```

Y el uso de Math.trunc(x), que devuelve la parte entera de un numero fraccionario (uso parecido a .toFixed())

Al igual que muchos de los objetos por defecto de JS, Math puede ser ampliado con propiedades y métodos personalizados, para esto **NO DEBE USARSE PROTOTYPE**, si no:

```
Math.propName = propValue;
Math.methodName = methodRef;
```

```

> Math.sumas = function(){
  let suma = 0;
  for(var i = 0; i<arguments.length; i++){
    suma += arguments[i]
  }
  return suma;
}
Math.sumas(1,2,3,5,7,10);
< 28

```

## THIS

Existen diferentes tipos de binding para el uso de 'this', esto quiere decir, que valor va a tomar this dentro de una función.

Podemos cambiar el valor de this para reutilizar métodos entre distintos objetos.

Podríamos decir que 'this' es un 'parámetro especial', preguntar que valor tiene, es equivalente a preguntar qué objeto está ejecutando la función en ese caso puntual (en que contexto se esta ejecutando).

Para comenzar veremos un casos en el que el uso de this y la llamada a la función generan confusión, al final de la explicación lograremos prever lo que sucederá en estas:

```

> const obj = {
  nombre : 'Agustin',
  saludar(){
    console.log(`Hola ${this.nombre}`);
  }
}

< undefined

> const saludar = obj.saludar;
< undefined

> saludar();
Hola undefined

```

Ahora saludar() no posee el valor de this que esperaríamos. Esto debido a que estamos llamando a una función suelta y el enlazamiento (binding), no está hecho a la función que nosotros esperamos, si no a la que JS utiliza como default (el objeto global, el cual no posee la propiedad nombre)

**THIS BINDING:** (el orden no es en vano, JS ve que tipo de enlazamiento es correcto aplicar, si no es el primero, pasara al segundo y asi). Para saber a que se enlazo el this, el motor de JS observa:

\*)Como fue escrita la función

\*)Si le hicimos alguna modificación desde que la creamos

\*)El lugar de invocación (call site), ultimo lugar donde pudimos cambiar el valor de this.



**LEXICAL BINDING:** (tipo de enlazamiento cuando utilizamos arrow functions):

**NEW BINDING:** (tipo de binding que usa JS cuando instanciamos objetos):

**EXPLICIT BINDING:** (cuando invocamos una función indirectamente con algún método de invocación) : Sirve para que nosotros elijamos exactamente que objeto queremos que sea this cuando se ejecuta la función. Esto sirve cuando queremos usar métodos de un objeto en otro, sin generar errores en el valor de this.

```
> const agus = {
  nombre : 'Agustin',
  saludar(gritando, conDespedida){
    const saludonormal = `Hola me llamo ${this.nombre}!`;
    const despedidanormal = 'Chau!';

    const saludo = gritando ?
      saludonormal.toUpperCase() : saludo;
    const despedida = gritando ?
      despedidanormal.toUpperCase() : despedida;

    console.log(saludo);
    if(conDespedida){
      console.log(despedida);
    }
  }
};
< undefined
> agus.saludar(true,true);
HOLA ME LLAMO AGUSTIN! VM3631:12
CHAU! VM3631:14
< undefined
> agus.saludar(true,false);
HOLA ME LLAMO AGUSTIN! VM3631:12
< undefined
```

supongamos que tenemos este objeto, con un método saludar un poco mas complejo, si queremos que ahora este método se ejecute en el contexto del objeto 'Pepe':

**Function.prototype.call(objeto,parametro,parametro);**

```
> const pepe = {
  nombre : 'Pepe',
}
< undefined
> agus.saludar.call(pepe,true,true);
HOLA ME LLAMO PEPE!
CHAU!
```

donde objeto = pepe y los parámetros true||false especificados en el método agus.saludar.

### **Function.prototype.apply(objeto, [parametro,parametro]);**

realiza lo mismo que .call, solo que los parámetros los pasamos en forma de array.

```
> agus.saludar.apply(pepe,[true,true]);  
HOLA ME LLAMO PEPE!  
CHAU!
```

### **Function.prototype.bind() => (método de enlazamiento fuerte);**

Nos retorna una nueva función con el contexto enlazado al objeto que le digamos, a diferencia de .call y .apply, .bind no se ejecuta en el instante que la llamamos

```
> const agus = {  
  nombre : 'Agustin',  
  saludar(){  
    console.log(`Hola me llamo ${this.nombre}`);  
  }  
}  
< undefined  
> const volverASaludar = agus.saludar.bind(agus);  
< undefined  
> volverASaludar();  
Hola me llamo Agustin VM4600:4
```

Volviendo al ejemplo con el que empezamos la explicación, ahora la nueva variable a la cual le asignamos como valor el método del objeto agus, no generara conflicto al leer el valor de this, ya que esta bindeada en una nueva función al método del objeto.

En el caso siguiente se muestra un botón en pantalla y cada vez que se toca se imprime el saludo sin errores.

```
const sacha = {  
  nombre: 'Sacha',  
  saludar: function () {  
    console.log(`Hola, me llamo ${this.nombre}!`);  
  }  
};  
  
const boton = document.getElementById('miBoton');  
boton.addEventListener('click', sacha.saludar.bind(sacha));
```

(todavía no vimos creación de eventos, pero para tener una idea)

**IMPLICIT BINDING:** (cuando esta implícito en el método de un objeto): es el mas intuitivo, se produce cuando invocamos al método de un objeto.

```
> const yo = {
  nombre : 'Agustin',
  getSaludo(){
    console.log(`Hola ${this.nombre} :)`);
  },
  hermano : {
    nombre : 'Cristian',
    getSaludo(){
      console.log(`Hola ${this.nombre} :)`);
    },
  },
};
< undefined
> yo.getSaludo();
  Hola Agustin :) VM2382:4
< undefined
> yo.hermano.getSaludo();
  Hola Cristian :) VM2382:9
```

No generamos conflictos en el uso de this, ya que esta enlazado con la funcion invocada en la propiedad dada.

**DEFAULT BINDING:** (cuando invocamos una función directamente): Si JS ve que no es correcto aplicar ningún binding anterior, correrá este, se da en los casos donde tenemos una función normal y de invocación directa.

```
> function salud(){
  console.log(`Hola yo soy ${this}`);
}
< undefined
> salud();
  Hola yo soy undefined
```

En modo estricto ('use strict';) la funciones sueltas están realmente sueltas y this no adquiere valor. Si no estamos en presencia del modo estricto, this hara referencia al objeto global window.

### BUENAS PRACTICAS DEL USO DE THIS:

- 1) No usar this en las funciones globales.
- 2) Cuando no sepamos el valor de this, consologear su valor puede darnos una buena ayuda (console.log(this));
- 3) Cuando trabajamos con eventos del DOM, this por defecto, es el elemento que dispara el evento.

- 4) Una función que fue creada con bind no puede volver a ser enlazada a otro objeto

### COMBINANDO EL DEFAULT BINDING CON IMPLICIT BINDING:

```
> const yo = {
  nombre: 'Agustin',
  twitter: 'IlCabezon',
  getSaludo(){
    console.log(`Hola ${this.nombre} :)`);
    this.seguimeEnTwitter();
  },
  seguimeEnTwitter(){
    console.log(`Seguime en twitter ${this.twitter}`);
  }
};
< undefined
> yo.getSaludo();
Hola Agustin :)
Seguime en twitter IlCabezon
```

Esto no genera conflictos, ya que `seguimeEnTwitter()` esta expresada como método del objeto en lugar de funcion suelta, además de su invocación mediante `this.seguimeEnTwitter()` dentro de `saludar()`, para que al saludarnos invoque también el metodo `seguimeEnTwitter()`;

**Mas ejemplos del uso de bind, call y apply:** Teniendo en cuenta es siguiente objeto:

```
var persona = {
  nombre: 'Franco',
  apellido: 'Chequer',

  getNombre: function(){
    var nombreCompleto = this.nombre + ' ' + this.apellido;
    return nombreCompleto;
  }
}

var logNombre = function(){
  console.log(this.getNombre());
}
```

**bind():**

```
> var logNombreP = logNombre.bind(persona);  
< undefined  
> logNombreP();  
Franco Chequer VM2527:12
```

**call():**

```
var logNombre = function(arg1, arg2){  
  console.log(arg1 + ' ' + this.getNombre() + ' ' + arg2);  
}  
  
logNombre.call(persona, 'Hola', ', Cómo estas?'); //Hola Franco Chequer , Cómo esta:
```

**apply():**

```
var logNombre = function(arg1, arg2){  
  console.log(arg1 + ' ' + this.getNombre() + ' ' + arg2);  
}  
  
logNombre.apply(persona, ['Hola', ', Cómo estas?']); //Hola Franco Chequer , Cómo e:
```

Si quisiera hacer una función que multiplique y por alguna razón, dejar algunos parámetros fijos: (el primer parametro del bind en este caso puede ser this o null ya que no lo estamos usando)

```
> function multiplica(a,b,c,d){  
  return a*b*c*d;  
}  
< undefined  
> var multiplicaBind = multiplica.bind(this||null,10,3);  
< undefined  
> multiplicaBind(1,1);  
< 30
```

## OBJETO ARGUMENTS

El objeto arguments nos permite pasar una cantidad indefinida de parámetros a una función, esto es útil cuando no sabemos con exactitud cuantos parámetros serán ingresados por el usuario.

Para manipular el objeto arguments, lo trabajamos como un array clásico (ojo. se trabaja como un array pero es un **objeto**).

```
function mayoresaseis(){
  var arr = [];

  for(var i=0, longitud=arguments.length;
  i<longitud;i++){
    if(arguments[i] >=6){
      arr.push(arguments[i]);
    }
  }

  return arr;
}
```

Nótese que a la función no se le pasa ningún parámetro.

## CLASES

Son una mejora sintáctica sobre la herencia basada en prototipos de JS.

```
> class Person{
  constructor(name,hobbie,age){
    this.name = name,
    this.age = age,
    this.hobbie = hobbie
  }
  getAge(){
    return this.age;
  }
  isAdult(){
    if(this.age > 21){
      return `${this.name} is adult`;
    }else{
      return `${this.name} is Wachin`;
    }
  }
}
< undefined
> var agus = new Person('Agustin','Skateboarding',20);
< undefined
> agus.isAdult()
< 'Agustin is Wachin'
```

Usamos el statement class, definimos un constructor y los métodos que queremos que tenga la clase. De todas maneras, la clase Person sigue siendo un objeto, y este objeto es utilizado como prototipo para los nuevos objetos creados con ella.

```
> agus.__proto__  
↳ {constructor: f, getAge: f, isAdult: f}
```

### Extends & Super :

```
> class Concursante extends Person {  
  constructor(name,hobbie,age,id,discipline){  
    super(name,hobbie,age);  
    this.id = id,  
    this.discipline = discipline  
  }  
}  
↳ undefined  
> var Lucas = new  
  Concursante('Lucas','Read',24,'#001','basketball');  
↳ undefined  
> Lucas.__proto__  
↳ ▶ Person {constructor: f}
```

Para agregar un prototipo a la clase creada se usan extends. El método super nos permite usar el constructor de la clase de la cual estamos heredando, en este caso Person.

```
> Concursante.__proto__  
↳ class Person{  
  constructor(name,hobbie,age){  
    this.name = name,  
    this.age = age,  
    this.hobbie = hobbie  
  }  
  getAge(){  
    return this.age;  
  }  
  isAdult(){  
    if(t...  
  }  
}  
> Lucas.isAdult()  
↳ 'Lucas is adult'
```

Nótese que los métodos de Person son aplicables al objeto creado, ya que la clase Person es el proto de concursante.

## EVENT LOOP



Es la manera en que JS se comporta, trabaja y ejecuta (web apis)

```
function saludarMasTarde(){  
  var saludo = 'Hola!';  
  
  setTimeout (function (){  
    console.log(saludo);  
  },3000);  
  
  saludarMasTarde()  
}
```

Si tenemos este ejemplo, el `setTimeout` debería parar todo por 3 segundos y ahí saludar. Javascript trabaja siempre sobre otro motor (v8 en nuestro caso), entonces en casos donde haya tareas que van a tardar, se delegan a 'otra persona' (el motor), y sigue ejecutando lo demás, por lo que no se crea otra línea de ejecución, si no que sigue de forma asíncrona, una vez terminado el call stack, vuelvo a recibir la tarea que se tardó, y la ejecuto.

**¿Por qué demoraría una tarea? En casos de que necesite información de una api, una base de datos (necesito una información de fb), leer un documento, etc, puedo demorar la ejecución de estas para que el programa siga corriendo sin hacer esperar al usuario.** (todo eso debido a que JS corre de una línea a la vez)

¿Te imaginas tener que esperar 5 minutos a que tu web reciba el nombre de facebook de un usuario?? Claramente tu web no sería eficiente y los usuarios no la elegirían, es por esa razón que demoramos o delegamos ciertas tareas, para que no haya que esperar para realizar la ejecución completa.



## MANEJO DE ERRORES EN JAVASCRIPT

Es normal encontrarnos con situaciones que no esperamos que sucedan, por eso es importante saber manejar los errores para que nuestro código no se rompa.

### Tipos de errores:

- 1) Errores de sintaxis: se produce cuando el programador no respeta las reglas sintácticas.
- 2) Errores semánticos : se dan por el mal uso de algún statement.
- 3) Errores lógicos: aparecen cuando el código no hace lo que esperábamos que haga
- 4) Errores en tiempo de compilación: Aparecen cuando nuestro código es parseado por un compilador o intérprete
- 5) Errores de programación: Cuando usamos mal una función o pasamos mal los argumentos (estos son los bugs).
- 6) Problemas genuinos: Aquellos que escapan a las manos del programador en programas bien codificados.

### Manejo de excepciones:

Las excepciones en programación son imprevistos que ocurren durante la ejecución de un programa que impiden o alteran el flujo natural. Básicamente es posible levantar o tirar una excepción. Podríamos decir que es parecido a un return que puede, no solo puede salir de la función en la que está, si no volver e incluso saltar varios execution contexts hasta llegar al entorno principal donde se haya iniciado la serie de invocación que dieron lugar a la excepción. Por suerte podemos intentar “catchear” una excepción que va subiendo o bajando por el stack de ejecución, de tal manera que ejecutemos el código en donde agarramos la excepción y seguir desde ahí:

```
try {  
  //codigo a ejecutar  
  [break;]  
}
```

```
catch(error){  
  //codigo a ejecutar si ocurre una excepción  
  [break;]  
}
```

```
//el finally es opcional  
[finally {  
  //siempre se ejecuta este código haya o no una excepción  
}]
```

Vamos a un ejemplo:

```
function lastElement(array) {  
  if (array.length > 0)  
    return array[array.length - 1];  
  else  
    throw "No existe el último elemento de un arreglo vacío.";  
}  
  
function lastElementPlusTen(array) {  
  return lastElement(array) + 10;  
}  
  
try {  
  print(lastElementPlusTen([]));  
}  
catch (error) {  
  print("Hubo un problema ", error);  
}
```

En este código definimos una función la cual nos devuelve el último elemento de un array. En esta parte introducimos las keyword **throw** (sera quien cree la excepcion), en este caso esta especificando que si nos pasan un array vacío en lugar de retornar undefined no arroje un mensaje, **try () catch**

## ESTRUCTURA DE DATOS

### <DEMO />

Cuando hablamos de estructura de datos, nos referimos a cómo organizamos los datos cuando programamos. Básicamente, este tema trata de encontrar formas particulares de organizar los datos de tal manera que puedan ser utilizados de manera eficiente.

**Estructuras prefabricadas por tipo de dato:** Las que ya conocemos (integer,float,character,pointer)

**Estructuras definidas por el usuario:**

- 1) Arrays: arreglos típicos.

Index	0	1	2	3	4
	H	e	l	l	o
Address	0x23451	0x23452	0x23453	0x23454	0x23455

## SETS

Son lo mismo que los arreglos solo que no permiten elementos repetidos. Cuando invocamos a set me devuelve un nuevo **objeto** con la particularidad que no tiene repeticiones (no los ordena). Los sets aceptan cualquier tipo de datos.

```
var arreglo = [1,2,3,4,4,5,5,1,2]
var set1 = new Set(arreglo)
console.log(arreglo) // [ 1, 2, 3, 4, 4, 5, 5, 1, 2 ]
console.log(set1) // Set { 1, 2, 3, 4, 5 }
```

Sets también tiene sus métodos asociados, proto y su prototipo, asu vez se pueden recorrer con for...of

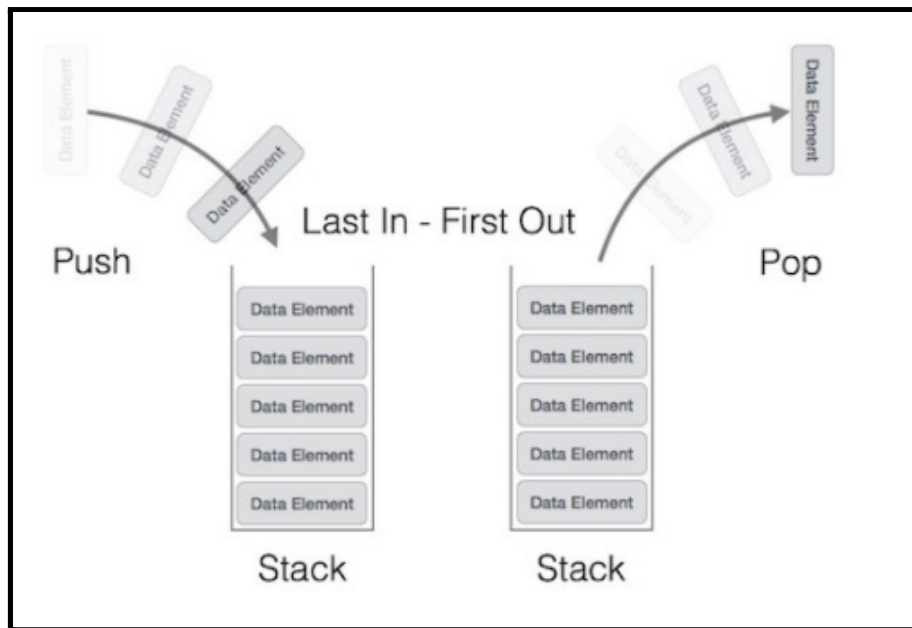
### 2) Lists:

**(simplemente enlazadas)** : podemos asemejar a una fila de niños (donde cada niño representa un **nodo o elemento**), los cuales están todos con la mano en el hombro del de adelante. Desde el punto de vista de uno de estos niños, sólo puedo conocer información del que esté adelante, no el que esté atrás. Si alguno de ellos saca la mano, se rompe la fila, **se rompe la lista**.

**(doblemente enlazado)** : en estas listas los **nodos** pueden conocer la información de lo que tienen adelante y tambien atras, de igual manera si rompo o saco un elemento, se rompe la lista (\\(°-°)/ \\(°-°)/ \\(°-°)/ (fila de niños de uno al lado del otro jaja))

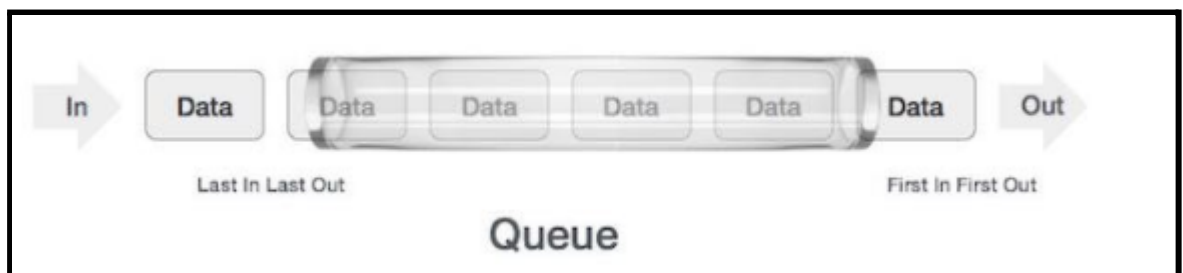
#### a)Linear list:

\*)Stacks: Imaginemos la call stack, en donde los procesos están uno encima de otros conectados con el de adelante. (last in,first out).



(nos manejamos con `.push()` y `.pop()`, comparamos su funcionamiento con el de un array (salvando la excepción de `shift()` ,`unshift()`))

\*Queue (colas): colas (supermercado por ejemplo), el primer en llegar, sera el primero en irse (first in,first out).



(al igual que stack, lo podemos pensar como un array con `.push()`, `shift()`)  
Con todo esto, también puedo hacer listas con prioridades, en donde analizo la cola que tengo y dependiendo las necesidades, acomodo el nuevo elemento en la cola (lista reversionada)

Para armar una Queue necesitamos una función constructora (o clase) la cual creará, cada vez que la invoquemos, un array en donde almacenaremos nuestros elementos:

```
function Queue(){
  |   this.data = [];
}
```

También tendrá en su prototipo (métodos) funciones de utilidad como:

- \*) Agregar elementos al final de la queue
- \*) Sustraer el primer elemento en entrar
- \*) Tamaño de la misma
- \*) Contenido

```

Queue.prototype.dequeue = function(){
    if(this.data.length > 0){
        this.data.shift();
    }else{
        return undefined;
    }
}

Queue.prototype.enqueue = function(a){
    return this.data.push(a);
}

Queue.prototype.size = function(){
    return this.data.length;
}

Queue.prototype.getQueue = function(){
    return this.data;
}

```

#### **b)Non-linear list:**

\*)Trees : estructura donde tengo un nodo principal donde tengo conectado diferentes hijos, hermanos y padres, aquellos que estén por debajo serán hijo, aquellos que estén a mi altura serán mis hermanos y los que están arriba serán mis padres (el DOM es una tree por ejemplo). Hay muchos tipos de arboles pero ya vamos a profundizar en estos.

\*)Graphs: conjunto entre nodos y aristas, donde las aristas tiene la conectividad de los nodos

#### **Diferencias:**

- a) Arreglos y listas: básicamente los arreglos ocupan posiciones contiguas en memoria, es decir, que arranca en 251, el proximo sera el 252, esto tiene

desventajas como: Requiere espacios libres contiguos en memoria, por ejemplo, tengo libres los casilleros 250 y 254 (antes y después ocupados), si necesito un arreglo de 5 posiciones no puedo guardarlo ahí, y esos espacios seguirán libres hasta encontrar un arreglo de esas posiciones o no llegaran a utilizarse nunca (las listas se almacenan en diferentes lugares pero tienen conocimiento de los datos a los que están relacionados)

**Listas enlazadas** : son nodos (entidad propia, una clase por si misma) que poseen en su interior data y una propiedad llamada **this.next** el cual representa el lazo con el de adelante (este “brazo” va a tener data si y solo si tiene a alguien adelante).



En este ejemplo los cuadrados naranja representan la data y el punto gris a this.next, que observa quien está delante de sí. El último next se mantiene en null.

Las listas enlazadas están formadas por:

- 1) Un head, que será el primero de la lista (tiene conocimiento sólo del que le sigue), compuesto de un this.length (inicialmente será cero ) y un this.head. (entonces cuando quiera saber algo de algún nodo, comenzaré por el head, recorriendo toda la lista. Cuando un nodo tenga como this.length:null sabremos que ese es el último de lista ya que no tiene a nadie por delante)

```
function list(){
  this.length = 0;
  this.head = null;
}
```

- 2) Nodos, los cuales buscan estar vinculados a la lista anterior.

```
function Nodo(data) {
  this.data = data;
  this.next = null;
}
```

**Creación de listas enlazadas:**

```
function Lista(){ //cuando invoquemos let list = new Lista() =>list sera nuestro head
```

```

        this.head = null;
        this.length = 0;
    }

    function Nodo(data) { //función constructora de nodos
        this.data = data;
        this.next = null;
    }

    Lista.prototype.newNode = function(data){

        let NuevoNodo = new Nodo(data);

        let current = this.head; //creamos un auxiliar que vaya variando su valor, que sera
        quien nos ayude a recorrer la lista

        if(!current){ //si current === null, quiere decir que este nodo sera el primero de la
        list
            this.head = NuevoNodo; // entonces vinculamos el head a este nodo, por lo que
            ahora list.head :{NuevoNodo};
            this.length ++; // y aumentamos en 1 el length de la lista
            return NuevoNodo;
        }
        while(current.next){ //ahora, si list.head.next (current.next) existe, significa que
        hay un nodo asociado en la posicion next
            current = current.next; //modificaremos su valor para que entre en el siguiente
            nodo, verificando una vez más si next ya tiene valor

        }
        current.next= NuevoNodo; //cuando nos topemos con un next sin valor asociado,
        salira del while, y asociara ese valor current.next:{NuevoNodo}
        this.length ++; //aumentando una vez mas el length de la lista
        return NuevoNodo;
    }
}

```

**Tener en cuenta que dentro de data se admite cualquier valor, y dentro del next puedo anidar incluso otra lista.**

```
> let ListaUno = new Lista;

ListaUno.newNode(10);
console.log(ListaUno);

▶ Lista {head: Nodo, length: 1} VM4777:4
< undefined
> ListaUno.newNode(20);
console.log(ListaUno.head.next);

▶ Nodo {data: 20, next: null} VM4785:2
< undefined
> ListaUno.newNode(30);
console.log(ListaUno.head.next.next);

▶ Nodo {data: 30, next: null} VM4958:2
< undefined
> ListaUno;
< ▼ Lista {head: Nodo, length: 3} ⓘ
  ▼ head: Nodo
    data: 10
    ▼ next: Nodo
      data: 20
      ▶ next: Nodo {data: 30, next: null}
      ▶ [[Prototype]]: Object
      ▶ [[Prototype]]: Object
```

Aca vemos como ir creando nodos y como quedaría lista después de crear varios nodos.

Cosas que podemos hacer con una lista:

- *Iterar sobre la lista:* Recorrer la lista viendo sus elementos o hasta que encontremos el elemento deseado.
- *Insertar un nodo:* La operación va a cambiar según el lugar donde queramos insertar el nodo nuevo:
  - Al principio de la lista.
  - En el medio de la lista.
  - Al final de la lista.
- *Sacar un nodo:*
  - Del principio de la lista.
  - Del medio de la lista.



## MÉTODOS DE LINKEDLIST:

En general podremos crear los métodos en base al tipo de información que precisemos, pero existen métodos generales para poder manipularlas con comodidad:

**remove()** = extrae el último elemento de la lista:

```
LinkedList.prototype.remove = function() {  
  let current = this.head; //let actual=this.head  
  if (!current)  
    return null; //si la lista esta vacia  
  if (!current.next) { //si la lista posee un solo nodo  
    this.length --;  
    //guardamos ese nodo  
    this.head = null; //asignamos al head como null para cortar el lazo con el nodo  
    return current.value; //retornamos el nodo.value eliminado  
  }  
  if (current.next) {  
    let previous = this.head;  
    while (current.next.next) { //vemos dos posiciones adelante, mientras current.next.next existe...  
      previous = current;  
      current = current.next; //nos introduciremos en el nodo siguiente para verificar dos posiciones adelante  
    } //cuanto no exista nodo dos posiciones adelante de el  
  
    let eliminado = previous.next.value;  
    previous.next = null; //cortaremos el lazo del penultimo nodo, pasandolo a null  
    this.length --;  
    return eliminado;  
  }  
}
```

**search()**= realiza una búsqueda dentro de la lista y lo compara con el valor pasado como parametro devolviendo el mismo en caso de que se encuentre:

```
LinkedList.prototype.search = function(val) {  
  let current = this.head;  
  if (typeof val !== 'function') {  
    while (current) {  
      if (val === current.value)  
        return val;  
      current = current.next;  
    }  
    return null;  
  } if (typeof val === 'function') {  
    while (current) {  
      if (val(current.value) === true) {  
        return current.value;  
      }  
      current = current.next;  
    }  
    return null;  
  }  
}
```

**switchPosNode** = cambiaremos los valores de un nodo con otro, teniendo en cuenta las posiciones pasadas por parametro:

```
LinkedList.prototype.switchPosValue = function(pos1, pos2){
  let count = 0;
  let current = this.head;
  let pos1val = '';
  let pos2val = '';
  if(pos1 < 0 || pos2 < 0) return null;
  if(pos1 > list.size() || pos2 > list.size()) return null;
  let aux = null;
  while(current){
    if(count === pos1){
      pos1val = current.value;
      aux = current;
    }
    else if(count === pos2){
      pos2val = current.value;
      current.value = pos1val;
    }
    count ++;
    current = current.next;
  }
  aux.value = pos2val;
  return true;
}
```

Cabe destacar que este método solo intercambia los valores, mas no los nodos en si.

Existe otro método para mover el nodo completo, esto nos será útil cuando el mismo tenga muchas propiedades dentro, de esta manera ahorraremos tener que declarar muchas variables y evitaremos que nuestro código se vuelva ilegible

### **LISTAS DOBLEMENTE ENLAZADAS:**

En las listas que vimos recién solo podemos recorrerlas en un sentido. En algunos casos nos puede servir recorrer las listas en ambos sentidos, para eso los nodos deben estar enlazados a un next y a un prev

```
> function lista(){
  this.head = null;
  this.length = 0;
}

function nodo(data){
  this.data=data;
  this.prev=null;
  this.next=null;
}

lista.prototype.add = function(data){
  let current = this.head;
  let newNode = new nodo(data);

  if(!current){
    this.head = newNode;
    this.length ++;
  }else{
    while(current.next){
      current.prev = current;
      current = current.next;
    }
    current.next=newNode;
    this.length ++;
  }
  return newNode;
}
```

Aca se puede ver que el método de enlace doble funciona:

```
> var Listeiyon = new lista;
< undefined

> Listeiyon.add(10)
< ▶ nodo {data: 10, prev: null, next: null}

> Listeiyon.add(20)
< ▶ nodo {data: 20, prev: null, next: null}

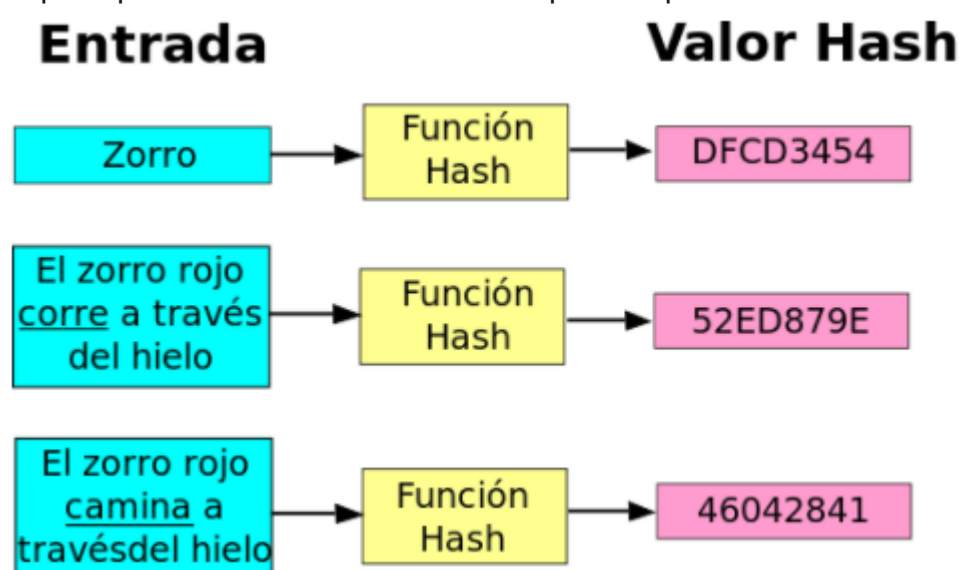
> Listeiyon.add(30)
< ▶ nodo {data: 30, prev: null, next: null}

> Listeiyon.add(40)
< ▶ nodo {data: 40, prev: null, next: null}

> Listeiyon
< ▼ lista {head: nodo, length: 4} ⓘ
  ▼ head: nodo
    data: 10
    ▶ next: nodo {data: 20, prev: nodo, next: nodo}
    ▶ prev: nodo {data: 10, prev: nodo, next: nodo}
    ▶ [[Prototype]]: Object
    length: 4
    ▶ [[Prototype]]: Object
```

## HASH TABLE

La principal función de estas es la búsqueda rápida de información



Para construir una hash table vamos a necesitar:

- *Una estructura de datos:* Acá vamos a guardar los datos y buscarlos por el índice. Puede ser un arreglo, o un árbol, etc.
- *Una función hashheadora:* Vamos a necesitar una función que transforme lo que elegimos de key a un hash que será nuestro índice.
- *Una política de resolución de colisiones:* Es la política que definiremos para decidir qué pasa cuando dos keys distintas generar dos hash iguales (las funciones no son perfectas !).

- 1) Estructura de datos: en este caso usaremos un arreglo con 'buckets' (baldes) adentro, los cuales nos servirán para casos de colisiones, es decir cuando dos keys apuntan al mismo índice:

```
function HashTable(buckets) {
  this.numBuckets = buckets || 35;
  this.data = new Array(this.numBuckets);
}
```

- 2) Una función hash que convertirá nuestra key en un índice, en este caso a través del código ASCII de cada letra del string.

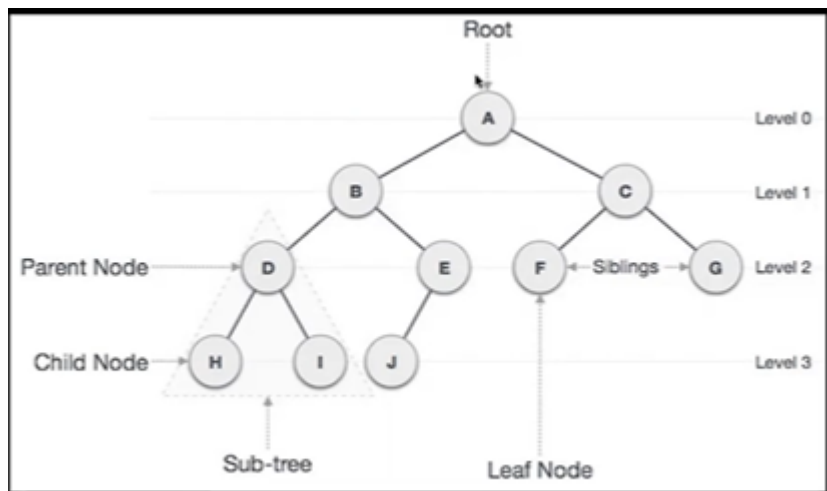
```
HashTable.prototype.hash = function(key){
  let result = 0;
  for(let i=0;i<key.length;i++){
    result += key.charCodeAt(i);
  }
  return result % this.numBuckets;
}
```

A través de la función Hash podremos: buscar elementos dentro de la HashTable, de manera más veloz que con otros metodos, agregar e incluso obtener la key original, entre otros métodos.

```
HashTable.prototype.set = function(key,value){
  let bucket = this.hash(key);
  if(typeof key !== 'string'){
    throw new TypeError ('Keys must be strings')
  }
  if(!this.data[bucket]){
    this.data[bucket] = {};
  }
  this.data[bucket][key] = value;
}
HashTable.prototype.get = function(key) {
  let bucket = this.hash(key);
  if(!this.data[bucket]) return undefined;
  return this.data[bucket][key];
}
HashTable.prototype.hasKey = function(key){
  let bucket = this.hash(key);
  if(!this.data[bucket]) return false;
  return this.data[bucket].hasOwnProperty(key);
}
```

## ÁRBOLES

Un árbol es un elemento raíz (nodo) sobre el cual se comenzaran a desprender hijos, a su vez hijos pueden tener asociados árboles. Nuestro elemento comienza en el nivel cero, el nivel uno, que sera los hijos, y así sucesivamente.



En este caso **A**, es el elemento de raíz, nivel cero.

**B y C** son hijos de A (por que comparten el mismo padre), nivel uno y son hermanos.

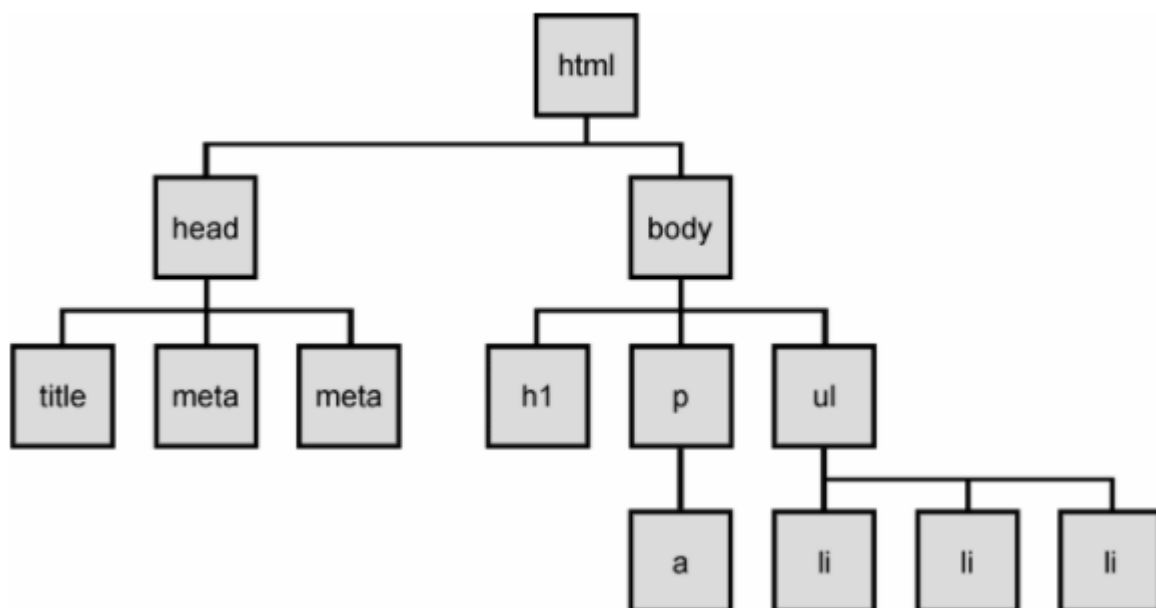
**F y G** son nodos 'hojas', no tienen hijos.

A su vez, cada nodo es un árbol, incluso los nodos hoja (son árboles sin hijos).

Entonces la construcción de un árbol es a su vez la creación de subárboles

Los árboles NO pueden tener ciclos y la información fluye siempre hacia abajo, a conoce a b pero no así b a a.

Si nos ponemos a analizar, el DOM de cada página es un árbol, a medida que voy creando lo que necesito, se va enlazando como hermano, hijos y así sucesivamente.

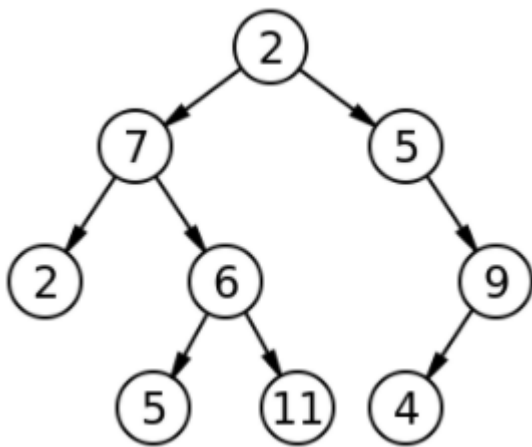


### Binary tree & Binary search tree

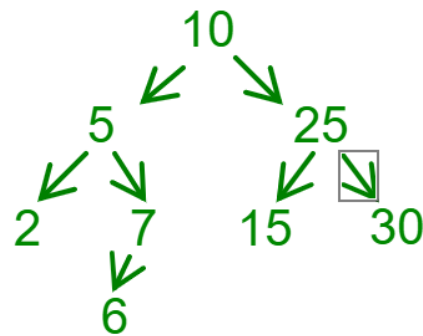
**Arboles binarios:** Son árboles donde cada nodo puede tener como máximo dos hijos

**Binary search tree:** Cada nodo puede tener como máximo dos hijo, pero además, se nos adiciona la restricción de que todos los números mayores al elemento raíz

irán en la derecha y los menores a la izquierda, en caso de valores repetidos dependerá de mi si lo vuelvo a almacenar, lo descarto, etc.

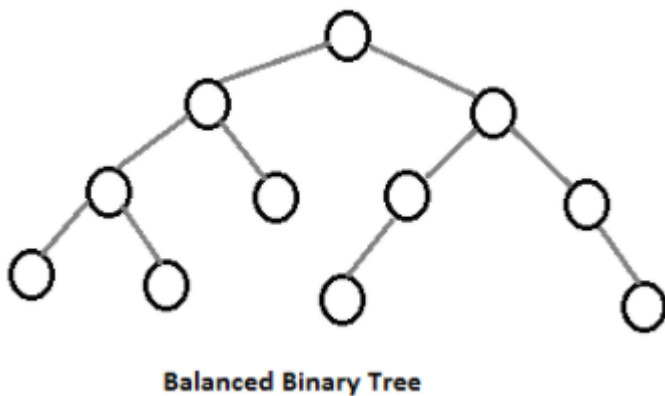


Binary tree



Binary search tree

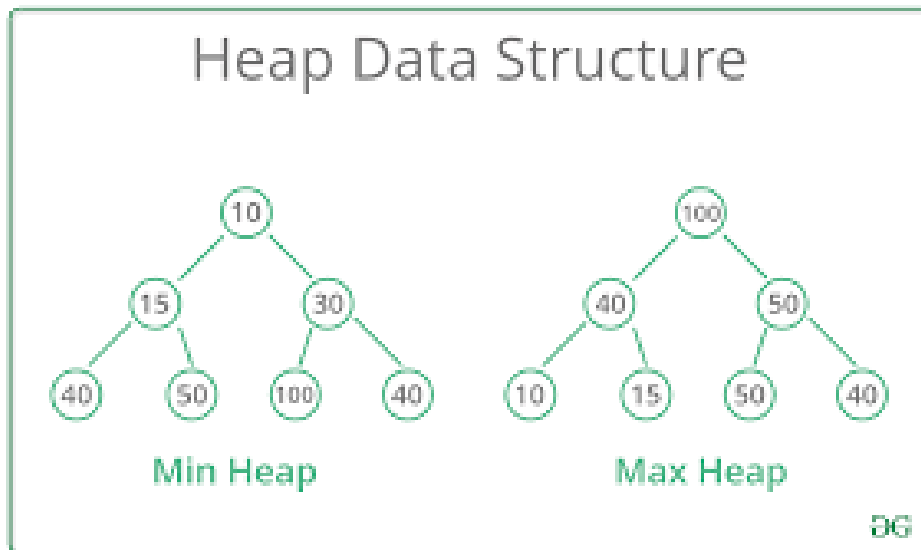
**Árbol binario autobalanceado (AVL)** : será un AVL si, la cantidad de elementos a izquierda es la misma o menos uno a la cantidad de elementos que tengo a derecha



(no lo programaremos pero esta bueno para practicar)

**Max heap**: en este tipo de árbol, el elemento más grande del árbol debe estar en el root y tiene que estar completo (cada nodo tiene un hermano o lo que es lo mismo, cada nodo alcanzó su cupo maximo de hijos)





se trabaja con recursividad

### CONSTRUCCIÓN DE UN BINARY SEARCH TREE:

Al igual que las demás estructuras de datos, los árboles se crean a partir de una función constructora base(o clase), la cual irá añadiendo nuevos árboles al árbol ya existente (eso si tenemos en cuenta que cada nodo del árbol es un árbol)

Luego de estos podemos implementar los métodos que sean necesarios y trabajen acorde al tipo de información que precisamos, en este ejemplo veremos metodos como: ***insert()***, ***search()***, ***size()***, ***contains()***, y ***manera de recorrer el arbol como: in order, pre order, post order, por nivel, etc.***

Función constructora:

```
function BinarySearchTree(value) {
  this.value = value
  this.left = null;
  this.right = null;
}
```

insert():

```
BinarySearchTree.prototype.insert = function
(value){
  if(this.value > value){
    if(!this.left){
      this.left = new BinarySearchTree(value);
    } else{
      this.left.insert(value);
    }
  } else{
    if(!this.right){
      this.right = new BinarySearchTree(value);
    } else {
      this.right.insert(value);
    }
  }
}
```

size(), contains():

```
BinarySearchTree.prototype.size = function (){
  if(!this.left && !this.right) return 1;
  if(this.left && !this.right) return 1 + this.
left.size();
  if(this.right && !this.left) return 1 + this.
right.size();
  if(this.right && this.left) return 1 + this.
right.size() + this.left.size();
}

BinarySearchTree.prototype.contains = function
(valor){
  if(this.value === valor) return true;
  if(this.value > valor && this.left){
    return this.left.contains(valor)
  }else if(this.right){
    return this.right.contains(valor);
  } return false;
}
```

height():

```
BinarySearchTree.prototype.height = function(){
  if(!this.left && !this.rigth) return 1;
  if(this.left && !this.rigth){
    return this.left.height() + 1;
  }
  if(!this.lefy && this.rigth){
    return this.rigth.height() + 1;
  }
  return Math.max(this.left.height()+1,this.rigth.height()+1);
}
```

## RESUMEN DE LO VISTO EN ESTRUCTURAS DE DATOS:

Estuctura	Ventajas	Desventajas
Arreglo	Rápida inserción, acceso muy rápido si conocemos el index.	Búsqueda Lenta, borrado lento.
Arreglo Ordenado	Mejor búsqueda que el arreglo normal	idem arreglo
Pila	Asegura LIFO	Acceso lento a los demás items
Cola	Asegura FIFO	Acceso lento a los demás items
Lista Enlazada	Rápida inserción, rapido borrado	Búsqueda Lenta
Árbol Binario	Muy rápido si está balanceado	Algoritmo de borrar es complejo, hay que balancear
Árbol Red-black	Rápido y siempre está balanceado	Es Complejo
Árbol 2-3-4	Muy bueno para guardar datos	Es Complejo
Hash	Muy rápido si conocemos el key. Insercion rapida	Borrado lento, buscado lento, consume mucha memoria
Heap	Rápida inserción, borrado y acceso al item mas grande	Acceso lento a los demás items

## ALGORITMOS

Para comenzar, un algoritmo es un conjunto de pasos a seguir, definidos y ordenados, para completar una tarea, estos no deben dejar dudas a quien realice dicha actividad. Los resultados deben ser esperados y no aleatorios.

Entonces, los algoritmos:

- 1) Resuelven un problema
- 2) Deben ser comprensibles
- 3) Deben ser lo más eficientes posibles

Pero, **¿cómo podemos medir la eficiencia de un algoritmo?**

La forma mas facil seria midiendo el tiempo que le toma al algoritmo encontrar la respuesta, pero esto solo nos servirá para la máquina que lo está procesando, como también el lenguaje que estemos usando

Podríamos analizar el espacio y la cantidad de recursos que precisamos para llevar a cabo la tarea, tal vez una mezcla de tiempo espacio y recursos sea una buena forma de comparar la eficiencia.

Como podemos deducir, un algoritmo puede tardar mucho o poco dependiendo del caso, los datos que se usen como input, etc. Por ejemplo, un algoritmo de búsqueda puede tardar más en casos donde hayan muchos elementos. En muchas ocasiones deberemos elegir entre mayor espacio en memoria ocupado y velocidad de ejecución (por ejemplo una hashtable, que ocupa mucho espacio pero realiza búsquedas rápidas) .

Para analizar qué tan compleja es la ejecución de un algoritmo en un problema, analizo desde el peor de los casos, es decir, la situación en la que más se tarda en encontrar la respuesta y a partir de ahí comparar.

Ahora, **¿Por qué nos importa medir la complejidad de los algoritmos?**

- 1) Predecir el comportamiento: hay casos donde algo puede tardar tanto que no tenemos el lujo de prueba y error, tenemos que conocer de antemano si un algoritmo va a terminar en un tiempo adecuado para el problema.
- 2) Compararlos: Según el problema vamos a tener que decidir cuál es el mejor algoritmo para usar, tampoco podemos ponernos a probar uno por uno.

**BIG O NOTATION (cota superior asintótica).**

La Big O notation intenta analizar la complejidad según crece el número de entradas (n) que tiene que analizar y lo que busca es una función que crezca de una forma con respecto a n, tal que nuestro algoritmo nunca crezca más rápido que esa función.

En palabras más simples podemos resumirlo a: 'este algoritmo nunca va a tardar más que esto, capaz tarda menos, pero más no'

**¿Cómo calculamos la Big O?**

- 1) Si un algoritmo hace  $f(n)$  pasos para llegar al resultado, entonces  $O(f(n))$

```

> function menor(arguments){
  let aux = arguments[0];
  for(let i=0;i<arguments.length;i++){
    if(arguments[i]<aux){
      aux = arguments[i];
    }
  }
  return aux;
}
< undefined
> menor([2,5,7,3,-1,-5])
< -5

```

En este caso  $n$  = elementos de arguments ( $O(6)$ )

- 2) Si un algoritmo hace  $f(n)$  pasos seguido de  $g(n) \Rightarrow O(f(n)+g(n))$ .

```

> function encontrarNum(arguments,num){
  let arraux=[];
  for(let i=0;i<arguments.length;i++){
    if(i%2 === 0){
      arraux.push(arguments[i]);
    }
  }
  for(let i=0;i<arraux.length;i++){
    if(arraux[i] / 2 === num){
      return `El numero ${num} se encuentra en el array`;
    }else{
      return `El numero ${num} no se encuentra en el array`
    }
  }
}
< undefined
> encontrarNum([2,4,6,8,10],3)
< 'El numero 3 no se encuentra en el array'

```

En este caso tendremos un primer for ( $f(x)$ ) que recorrerá  $n$  elementos de arguments, luego un segundo for ( $g(x)$ ) que recorre el array auxiliar que contendrá los elementos que coincidan con la búsqueda.

En el ejemplo sera:  $O(6 + 6) \Rightarrow O(12)$ .

- 3) Si  $f(n) > g(n)$ , podemos decir que  $O(f(n) + f(n)) = O(f(n))$ , esto debido a que podemos obviar constantes
- 4) Si el algoritmo hace  $g(n)$  pasos por cada  $f(n)$  pasos, el algoritmo es  $O(f(n)*g(n))$ .

El ejemplo mas simple que se me ocurre es el caso de dos for anidados, en donde, por cada iteración de  $i$ , se itera  $j$  por todos los elementos del array, entonces, en caso de  $\Rightarrow [1,2,3,4,5] \Rightarrow i=5$  y  $j=25$ , entonces  $O(\text{Math.pow}(5,2))$ , o lo que es lo mismo  $O(25)$

5) Debemos ignorar constantes múltiples  $\Rightarrow O(C \cdot f(n)) = O(f(C \cdot n)) = O(f(n))$

A continuación presentare un ejemplo de optimización de un algoritmo, el cual busca el maximo y minimo de un array:

```
var max = array[0];
for( var i = 0; i <= array.length; i++){
    if( array[i] > max) {
        max = array[i];
    }
}
var min = array[0];
for( var i = 0; i <= array.length; i++){
    if( array[i] < max) {
        min = array[i];
    }
}

console.log(max);
console.log(min);
}; // O( N + N ) = O(2N)
```

En este caso notamos que, con la presencia de dos bucles for no anidados  $O(N + N) \Rightarrow O(2N)$ , sin embargo, estamos duplicando la ineficiencia del algoritmo innecesariamente, una forma de solucionarlo seria trabajar con un único bucle que realice la tarea completa:

```
var max = array[0];
var min = array[0];
for( var i = 0; i <= array.length; i++){
    if( array[i] > max) {
        max = array[i];
    }
    if( array[i] < min){
        min = array[i]
    }
}
console.log(max);
console.log(min);
}; // O( N ) = O(N)
```

Ejemplos de funciones típicas con su respectivo Big O:

1) `var max = 10;`  $\Rightarrow O(1)$ ;

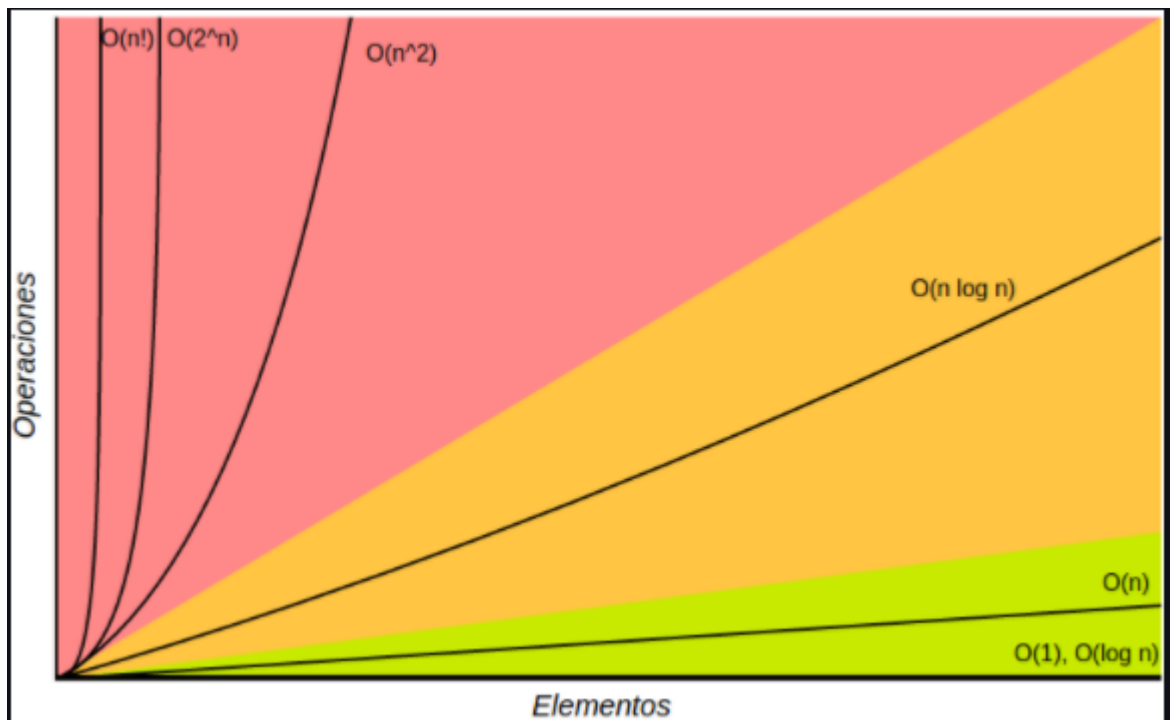
- 2)  $O(n)$ , hace una acción por cada entrada:

```
array.forEach( function(elem){  
    console.log(elem * 2);  
})
```

- 3)  $O(\text{Math.pow}(n,2))$ : por cada entrada recorre todas las entradas de nuevo

```
array.forEach( function(elem){  
    array.forEach( function(elemdos){  
        console.log(elem * elemdos);  
    })  
})
```

- 4)  $O(nc)$   $N$  elevado a la  $C$ : es el concepto general del anterior, por cada entrada el algoritmo recorre todas las demás entradas  $c$  veces.
- 5) Otros  $O$ : podemos encontrar algoritmos que sean  $O$  de la raíz de  $N$ , ó  $O$  de  $N$  elevado a un medio.
- 6)  $O(\log N)$  : se aplica en problemas donde en cada paso tenemos que recorrer la mitad de las entradas que quedan.
- 7)  $O(N!)$  ( $n$  factorial) : Esta complejidad aparece en casos donde debemos acomodar items, por ejemplo para los números 1,2 y 3, puede haber  $3!$  combinaciones distintas.



A continuación, estas tablas realizan una comparativa de tiempo si pudiéramos ejecutar 1000 instrucciones a la vez, con cada algoritmo:

Runtime	F(1,000)	Time
$O(\log N)$	10	0.00001 sec
$O(\sqrt{N})$	32	0.00003 sec
$N$	1,000	0.001 sec
$N^2$	1,000,000	1 sec
$2^N$	$1.07 \times 10^{301}$	$3.40 \times 10^{287}$ years
$N!$	$4.02 \times 10^{2567}$	$1.28 \times 10^{2554}$ years

Ahora, ¿qué cantidad de datos podría procesar cada algoritmo por segundo?

Runtime	N
$O(\log N)$	$> 1 \times 10^{300,000}$
$O(\sqrt{N})$	1 trillion
$N$	1 million
$N^2$	1 thousand
$2^N$	20
$N!$	10

Vemos que los últimos son los menos eficaces y se podrían usar con entradas pequeñas.

### PROBLEMAS P Y NP:

P=> Tiempo polinómico: Si la cantidad de operaciones que necesita un algoritmo para terminar es un polinomio, decimos que el algoritmo termina en tiempo polinómico. Además, si para llegar al resultado realiza una cierta cantidad de pasos, y siempre va a realizar los mismos, podemos decir que el algoritmo es determinístico. Esto es bueno, quiere decir que podemos calcular cuanto va a tardar a priori.

NP=> Tiempo polinómico no determinístico: Estos algoritmos, no son determinísticos, esto se debe a que encontrar la solución real nos puede llegar a tomar muchísimo tiempo. Entonces para resolverlos, lo que hacemos es ir probando algunas soluciones, o descartando las soluciones que sabemos que no son soluciones.

### ALGORITMOS DE BÚSQUEDA:

Para ver los algoritmos de búsqueda de una forma mas grafica y facil de entender, recomiendo el uso de la siguiente página:

[Algoritmos de búsqueda y ordenamiento](#)



## INSERTION SORT:

Consiste en extraer el elemento del conjunto y agregarlo a la posición que corresponde según el ordenamiento que estemos usando (creciente por ejemplo). Para hacerlo, el algoritmo tiene que ver todos los elementos y compararlo con el objeto extraído hasta que encontrar la posición que corresponda:

```
function insertionSort(array) {  
  for(let i=1;i<array.length;i++){  
    let aux = array[i];  
    let j = i-1;  
    while((j>-1) && aux < array[j]){  
      array[j+1] = array[j];  
      j--;  
    }  
    array[j+1] = aux;  
  }  
  return array;  
}
```

Peor de los casos =>  $O(n^2)$

Mejor de los casos =>  $O(n)$

## SELECTION SORT:

En este caso, el algoritmo intenta ordenar por posición, empieza en la posición mínima y busca el elemento que corresponda a ese lugar.

```
var selectionSort = function(array){  
  for(var i = 0; i < array.length; i++){  
    //set min to the current iteration of i  
    var min = i;  
    for(var j = i+1; j < array.length; j++){  
      if(array[j] < array[min]){  
        min = j;  
      }  
    }  
    var temp = array[i];  
    array[i] = array[min];  
    array[min] = temp;  
  }  
  return array;  
}
```

La complejidad es de  $O(n^2)$

## BUBBLE SORT:

- 1) Recorre los elementos: si dos items adyacentes están desordenados, hacé un swap.
- 2) Si terminas un recorrido sin hacer un swap, ya está ordenado.

```
function bubbleSort(array) {  
  let aux = 0;  
  for (let i = 1; i < array.length; i++) {  
    for (let j = 0; j < array.length - i; j++) {  
      if(array[j] > array[j + 1]){  
        aux = array[j];  
        array[j] = array[j + 1];  
        array[j + 1] = aux;  
      }  
    }  
  }  
  return array;  
}
```

Su complejidad sera:  $O(N^2)$

## ECMAScript 6

Una de las implementaciones más importantes fue la utilización de **const y let**, así como las **arrow function (let a = ()=>{})**, a continuación expondremos los usos de los features mas importantes:

**Destructuring:** La sintaxis de desestructuración es una expresión de JavaScript que permite desempacar valores de arreglos o propiedades de objetos en distintas variables.

Con arrays:

```
> const arr = [1,2,3,4,5];  
  
const [one,two,...rest] = arr;  
↵ undefined  
  
> console.log(`The plus of one and two is ${one+two}, and the rest of array is  
  ${rest}`);  
The plus of one and two is 3, and the rest of array is 3,4,5
```

Con objetos:

```
> const obj = {
  siete : 7,
  tres : 3,
  operaciones : {
    doble : 6,
    mitad : 1.5,
    cuadrado : 9,
  }
}

< undefined

> const {siete,tres,operaciones:{doble,cuadrado}} = obj;
< undefined

> siete
< 7

> tres
< 3

> cuadrado
< 9
```

En caso de que ya tengo una variable declarada con el nombre siete, para no romper el código y que este no sepa a qué siete estoy haciendo referencia, uso “siete : newSiete”, de esta forma el programa entenderá que le estoy pidiendo una nueva instancia de siete que se encuentra dentro del objeto:

```
> const obj = {
  siete : 7,
  tres : 3,
  operaciones : {
    doble : 6,
    mitad : 1.5,
    cuadrado : 9,
  }
}
const siete = 'siete';
const {siete: newSiete ,tres,operaciones:{doble,cuadrado}} = obj;
< undefined

> newSiete
< 7

> siete
< 'siete'
```

Con funciones: incluso cuando la función requiera de un parámetro, puedo aplicarle un destructuring.

```
> function fn(nombre){  
    return [nombre,'castro',20];  
}  
< undefined  
  
> const [name,surname,age]=fn('agustin');  
< undefined  
  
> name  
< 'agustin'  
  
> surname  
< 'castro'  
  
> age  
< 20
```

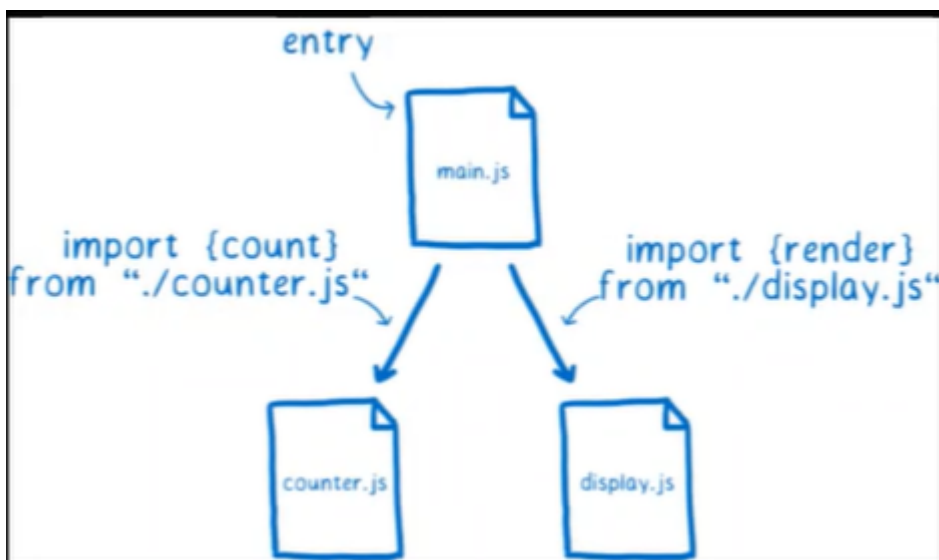
## MODULARIZACIÓN

(Antes de empezar, una aclaración, si a la hora de trabajar lo hacemos desde el backend => commonJS, pero si lo hacemos del frontend => EC6)

La ventaja de modularizar nuestros archivos JS es evitar errores en las variables, de esta manera nos ahorramos el problema de que las variables se pisen.

En un ejemplo simple, un módulo funciona como una caja, donde las variables tienen sentido y definición dentro de esa caja y solo hago público o exporto una parte del código (por ejemplo una función o un método).

Entonces un modulo es un pedazo de código que cumple una tarea específica y que indica sobre que piezas de código depende(dependencias)



### **PATRÓN DE MÓDULOS IFFE (obsoleto):**

Anteriormente se creaban los módulos dentro de funciones, ya que las variables solo existían dentro del contexto de esta función. Esto se hacía mediante una función autoinvocada, guardada en una variable y si necesito algo de ella la llamo como en el ejemplo (`weekDay.number(weekDay.number('Domingo'))`)

```

1 const weekDay = function() {
2   const names = ["Domingo", "Lunes", "Martes", "Miercoles",
3                 "Jueves", "Viernes", "Sabado"];
4   return {
5     name: function name(number) { return names[number]; },
6     number: function number(name) { return names.indexOf(name); }
7   };
8 }();
9
10 console.log(weekDay.name(weekDay.number("Domingo")));

```

Nótese que se usan closures para poder mantener privada esa información y poder modificarla.

### CommonJS:

Ahí es cuando commonJs implementa una nueva metodología que implementa el **require** y los **module.exports**.

\*Export me da la posibilidad de exportar ciertas cosas (export.name => exporta la funcion name)

\*Cuando lo importo, lo hago guardando la ruta del archivo en una variable  
La importancia del nombre es en el export pero no en el nombre de la función.

```

1 // WeekDays.js
2
3 var names = ["Domingo", "Lunes", "Martes", "Miercoles",
4             "Jueves", "Viernes", "Sabado"];
5
6 exports.name = function name (number) { return names[number]; };
7 exports.number = function number(name) { return names.indexOf(name); };

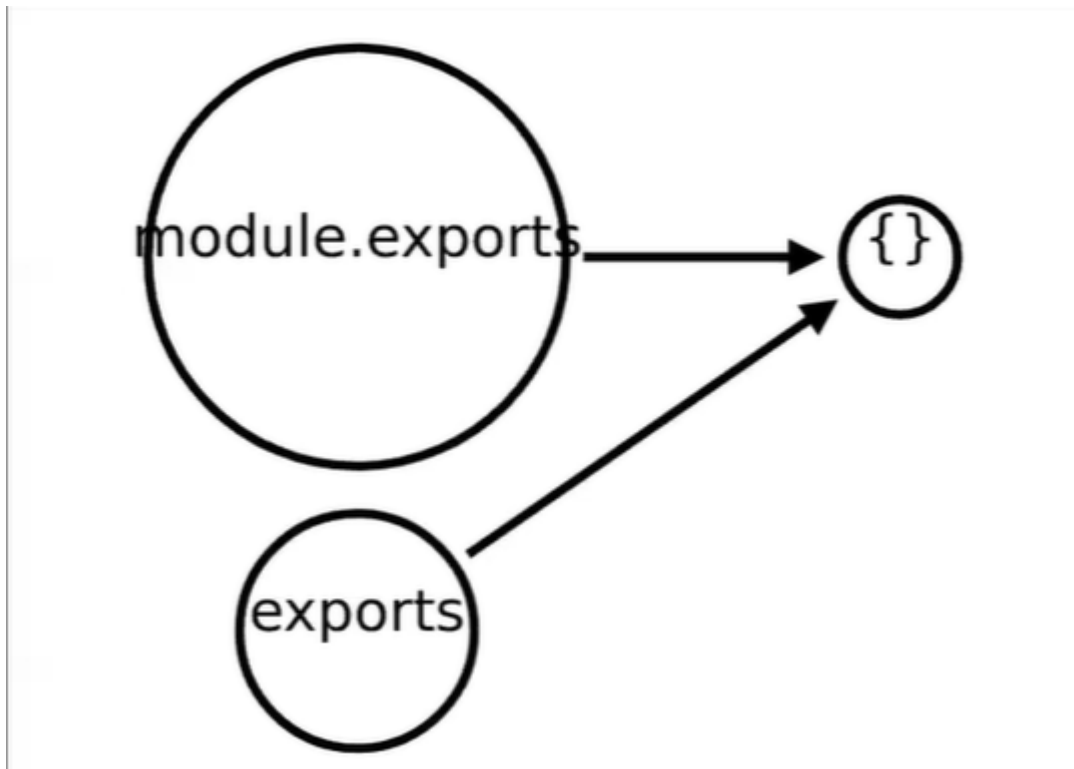
```

```

1 // index.js
2
3 var weekDays = require('./WeekDays.js');
4
5 console.log(weekDay.name(weekDay.number("Domingo")));

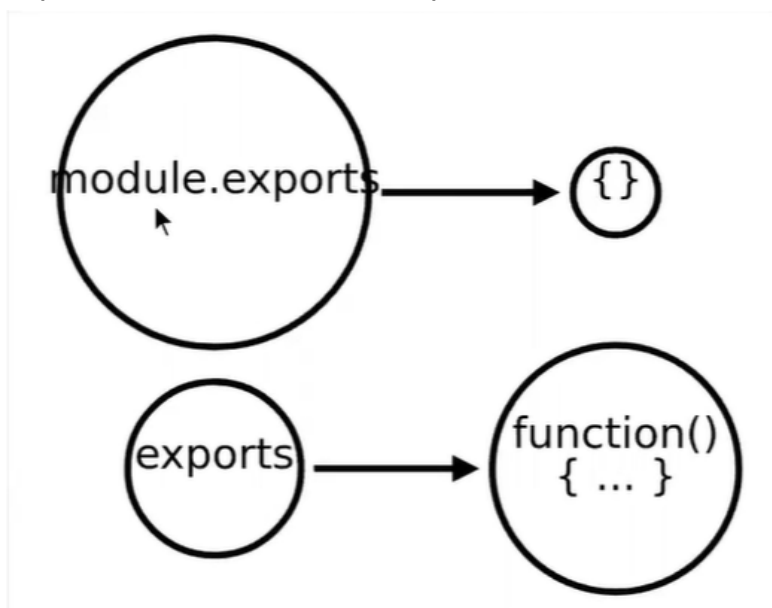
```

Aunque esto no funciona en el navegador, ya que no se admite el require. Otro problema es la diferencia de **module.exports** y **exports**.



module.exports por defecto es un objeto vacío y export (entonces podríamos decir que export no existe, es una especie de acceso directo al objeto module.export y funciona de la misma manera que una variable en JS, haciendo una **referencia** al objeto) es un puntero. Entonces, si quisiera definir una función como valor de module.export, ahora si, tanto export cómo module.export tendrán de valor esa función.

Si pisamos la referencia de export:



(module.export.nombre)

```
1 // module1.js
2
3 module.exports.a = function() {
4     console.log("a");
5 }
6
7 module.exports.b = function() {
8     console.log("b");
9 }

1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // "a"
6
7 example.b(); // "b"
```

En el ejemplo que sigue, el uso de module solo sobrescribe todo lo que habíamos hecho en el objeto module.export (en archivos .js diferentes los módulos se reinician)

```
1 // module1.js
2
3 module.exports.a = function() {
4     console.log("a");
5 }
6
7 module.exports.b = function() {
8     console.log("b");
9 }
10
11 module.exports = function extra() {
12     console.log("Extra function");
13 }

1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // TypeError: example.a is not a function
6
7 example.b(); // TypeError: example.a is not a function
8
9 example.extra(); // TypeError: prueba.extra is not a function
10
11 example(); // "Extra function"
```

tambien podria hacer un module.export = {a:function,b:function}



Veámoslo ahora con `.export` sólo:

```
1 // module1.js
2
3 exports.a = function() {
4     console.log("a");
5 }
6 exports.b = function() {
7     console.log("b");
8 }

1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // "a"
6
7 example.b(); // "b"
```

Y ahora es donde se explota todo lo que creíamos entender:

```
1 // module1.js
2
3 exports.a = function() {
4     console.log("a");
5 }
6 exports.b = function() {
7     console.log("b");
8 }
9
10 exports = function extra() {
11     console.log('Extra function');
12 }

1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // "a"
6
7 example.b(); // "b"
8
9 example.extra() // TypeError: example.extra is not a function
10
11 example(); // TypeError: example is not a function
```

vemos que tanto `example.a` cómo `example.b` funcionan, pero `export` sólo no existe, es por eso que si lo igualó a algo, esto no hace referencia a nada (recordar que `module.export` no es lo mismo que `export`!!!!)

Aca un ejemplo mas claro de lo que sucede con export:

```
> var ex = {a: 1, b: 2}
< undefined
> ex
< {a: 1, b: 2}
> var e = ex
< undefined
> e
< {a: 1, b: 2}
> e.a = 5
< 5
> e
< {a: 5, b: 2}
> ex
< {a: 5, b: 2}
> e = {}
< {}
> ex
< {a: 5, b: 2}
```

e = {} no cambia el valor del objeto ex.

### Desde ES6:

Se agrega **import** que llega para facilitarnos la vida, si trabajamos desde el front:

```
1 // WeekDays.js
2
3 var names = ["Domingo", "Lunes", "Martes", "Miercoles",
4             "Jueves", "Viernes", "Sabado"];
5
6 export function name (number) { return names[number]; };
7 export function number(name) { return names.indexOf(name); };
8
9 export default function myDefault () {
10   // otras cosas...
11 }
```

```
1 // index.js
2
3 import { number, name } from './WeekDays.js';
4
5 console.log(name(number("Domingo")));
```

export function name() {} => son exportaciones nombradas

Y se suma el **default**, que permite exportar una función general para el objeto.

Ahora, si quiero importar de manera nombrada, pasando mediante **destructuring** y poniendo un from con la ruta del archivo que importó.

Si quiero importar el objeto general, importo el default => import def, {name,number} from './js', ahora sí quiero ejecutar la función default => def(). Si la importación es por defecto, la función se puede pasar con cualquier nombre, por eso def() en lugar de myDeafult() como dice el ejemplo.

import def, {name as Name,number} from './m' => as, nos permite darle un sobrenombre a la función pasada por nombre.

Ahora, si quisiera lograr que funcione esto en Node.js, **agrego un 'type': 'module'** en el package.json del conjunto de archivos donde esté trabajando.

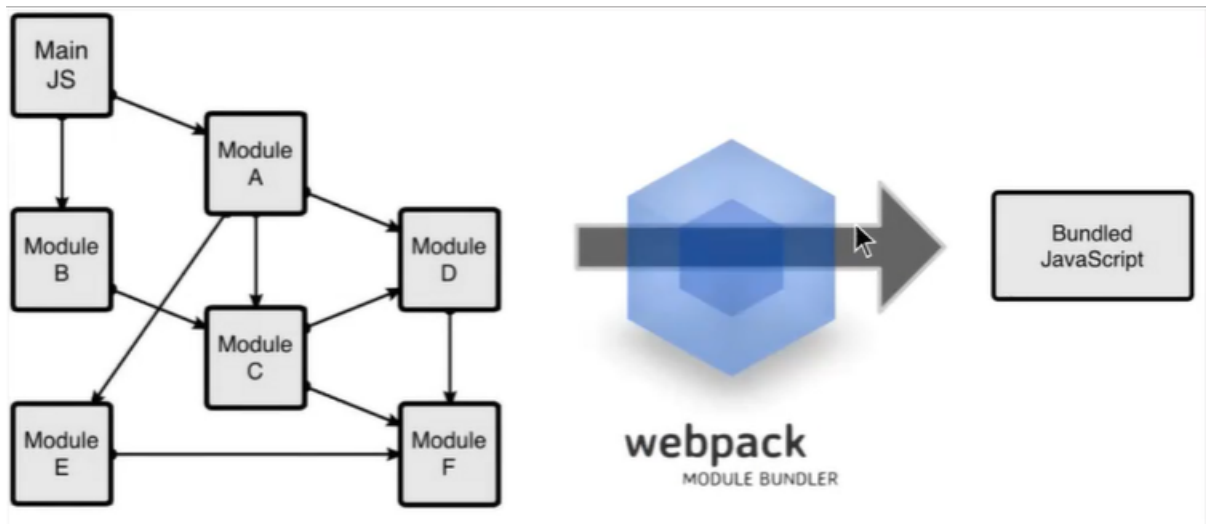
## BUNDLERS

Bien, ahora que sabemos algo sobre módulos, veamos cómo estos revolucionaron la forma de escribir código para el front-end con la introducción de los bundlers.

Como sabemos, la forma de importar librerías (que a su vez son módulos) en HTML es la siguiente:

```
<html>
  <head>My Webpage</head>
  <body>
    <script src="script1.js"></script>
    <script src="script2.js"></script>
    <script src="script3.js"></script>
    <script src="script4.js"></script>
  </body>
</html>
```

Acá es donde entra en juego webpack, que organizará nuestro código, con un main como archivo principal, con sus importaciones, librerías, etc y combina todo esto en un único archivo bundle.js donde está toda la data ordenada y optimizada para el navegador.



Para instalar webpack:

```
1 npm init
2
3 npm install --save-dev webpack
4
5 // package.json
6
7 "scripts": {
8   "start": "node server.js",
9   "build": "webpack"
10 }
11
12 // crear archivo webpack.config.js`
```

(no olvidarse de añadir lo especificado en los scripts de package.json)

Y además debemos crear un archivo webpack.config.js

```
1 // webpack.config.js`
2
3 module.exports = {
4   entry: './browser/app.js', // el punto de arranque de nuestro programa
5   output: {
6     path: __dirname + '/browser', // el path absoluto para
7       // el directorio donde queremos que el output sea colocado
8     filename: 'bundle.js' // el nombre del archivo que va a contener
9       // nuestro output - podemos nombrarlo como queramos pero bundle.js es lo típico
10   }
11 }
```

Que tendrá un entry => archivo principal del código

Y un output que dirá en qué ruta saldrá este código y con qué nombre (npm run build despues de eso)

Dentro del html llamo a:

```
<!doctype html>
<html>
  <head>
  </head>
  <body>

    <script src="./dist/bundle.js"></script>

  </body>
</html>
```

