# Scope.

El contexto actual de ejecución. El contexto en el que los valores y las expresiones son “visibles” o pueden ser referenciadas. Es el alcance (visibilidad) que vamos a tener de una variable para poder utilizarla.

Se puede dividir el scope en dos:

* Global. Se puede acceder desde cualquier parte del código.
* Local. El acceso es limitado, donde a su vez se puede dividir en dos: por **función** y por **bloque.**

A esto también se le conoce como contexto de ejecución, ya que cada vez que JS lee un código, siempre **evalúa su entorno** y **ejecuta código.**

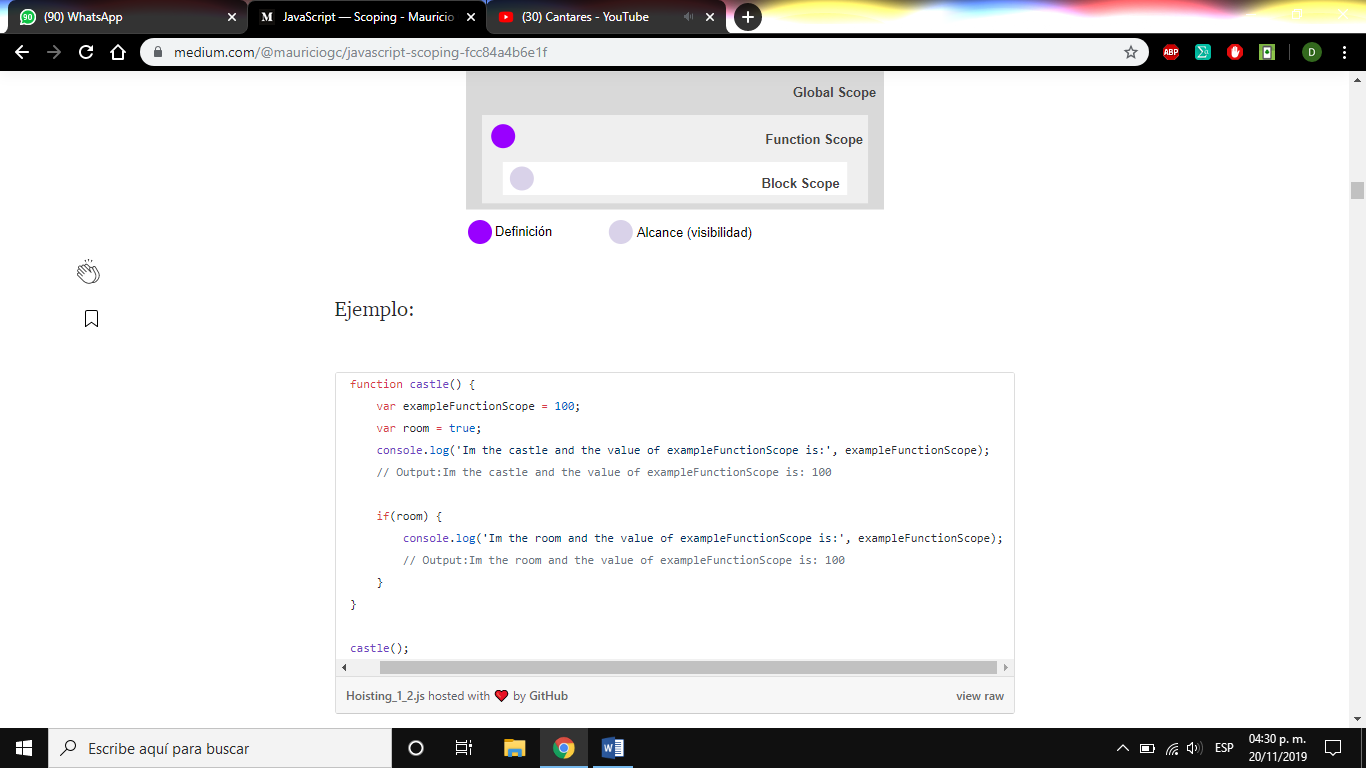
# Variables.

## Var

Es utilizada para declarar una variable, ya sea que se **inicialice** al **momento**  o **después.**

**Global scope:** Podemos definir variables tipo *var* donde su alcance sea global y se pueda acceder desde cualquier lugar.

**Function scope:** Podemos definir una variable de tipo *var,* donde su alcance sea dentro de la función y solo se pueda acceder a ella dentro de la misma.



Ejemplo:

function castle() {

var exampleFunctionScope = 100;

var room = true;

console.log('Im the castle and the value of exampleFunctionScope is:', exampleFunctionScope);

// Output:Im the castle and the value of exampleFunctionScope is: 100

if(room) {

console.log('Im the room and the value of exampleFunctionScope is:', exampleFunctionScope);

// Output:Im the room and the value of exampleFunctionScope is: 100

}

}

castle();

Si intentamos acceder a la variable *exampleFunctionScope* fuera de la función nos arrojaría un **error** ya que no existe y además **rompería** el **código:**

function castle() {

var exampleFunctionScope = 100;

var room = true;

console.log('Im the castle and the value of exampleFunctionScope is:', exampleFunctionScope);

if(room) {

console.log('Im the room and the value of exampleFunctionScope is:', exampleFunctionScope);

}

}

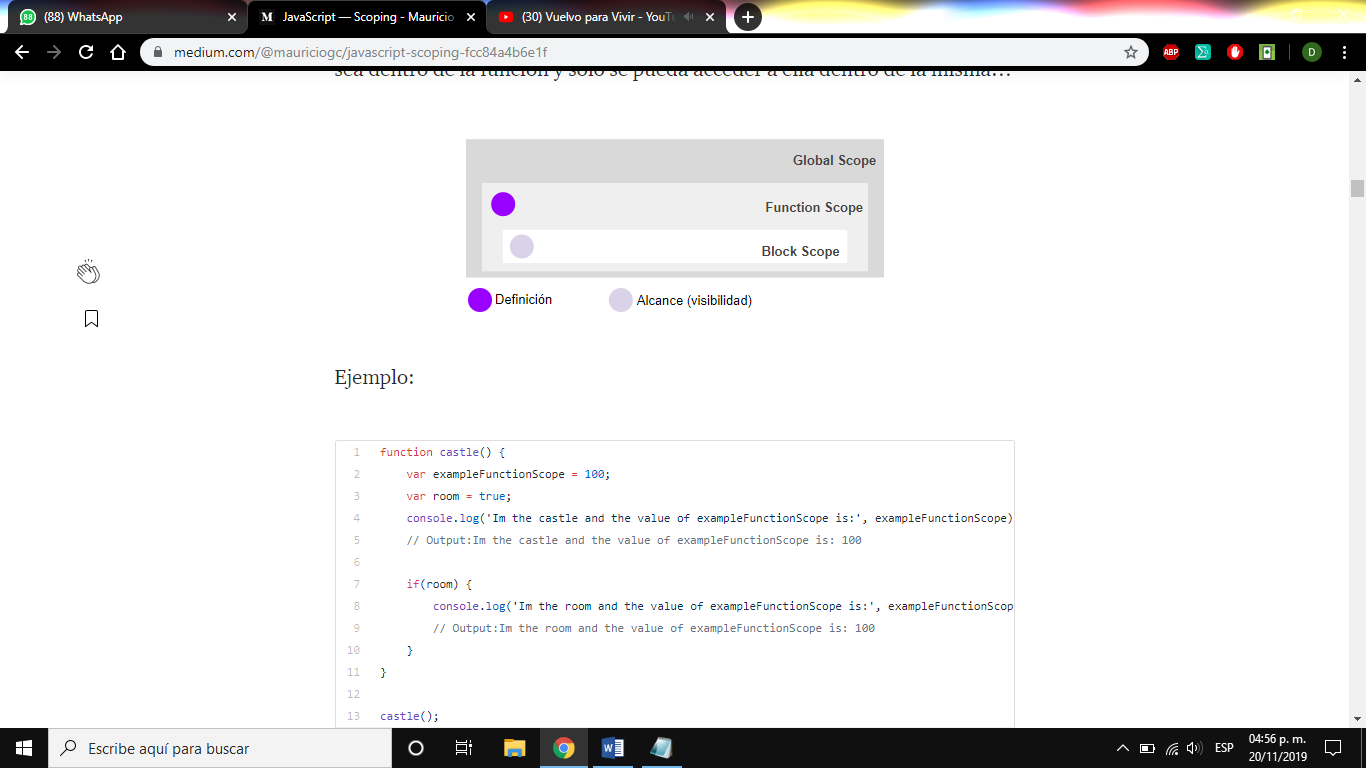
console.log('Im the map and the value of exampleFunctionScope is:', exampleFunctionScope);

//Output: Uncaught ReferenceError: exampleFunctionScope is not defined at <anonymous>

//Ya no se ejecuta

castle();

**Block scope:** Para el caso de la variable tipo *var*, el *block scope* su alcance es el mismo que *function scope*, esto se debe a que son procesadas **antes** de la **ejecución** del **código**.



function castle() {

var room = true;

if(room) {

var exampleBlockScope = 100;

console.log('Im the room and the value of exampleBlockScope is:', exampleBlockScope);

// Output: Im the room and the value of exampleBlockScope is: 100

}

}

castle();

Se define la variable dentro del *if(room)* donde la consola regresa el valor esperado. Su alcance *function scope,* como podemos ver en el siguiente ejemplo:

function castle() {

var room = true;

console.log('Im the castle and the value of exampleBlockScope is:', exampleBlockScope);

// Output:Im the castle and the value of exampleBlockScope is: undefined

if(room) {

var exampleBlockScope = 100;

console.log('Im the room and the value of exampleBlockScope is:', exampleBlockScope);

// Output:Im the room and the value of exampleBlockScope is: 100

}

console.log('Im the castle (2) and the value of exampleBlockScope is:', exampleBlockScope);

// Output:Im the castle (2) and the value of exampleBlockScope is: 100

}

castle();

Se agregaron dos consolas, una antes de la definición de variable y otra después de la definición; se puede observar:

* La primera consola (antes de su definición) la variable *exampleBlockScope* nos retorna *undefined* y no un **error.**
* La segunda consola (dentro del *if*) ya tiene un valor la variable *exampleBlockScope* y por eso nos regresa el valor de 100.
* En el tercera consola (después del *if*) ya tiene el valor de 100 ya que su alcance al ser *function scope* no nos arroja error.

En resumen, el **alcance** que tiene una **variable tipo var SIEMPRE va a ser** *scope function.*

***Las ventajas de usar var:***

* Las variables declaradas con var son procesadas **antes de la ejecución del código.**
* Su **scope** es su **contexto** de **ejecución** (entorno en el que se está en ejecución el código).
* El **scope** declarado **fuera** de una **función** es **global.**

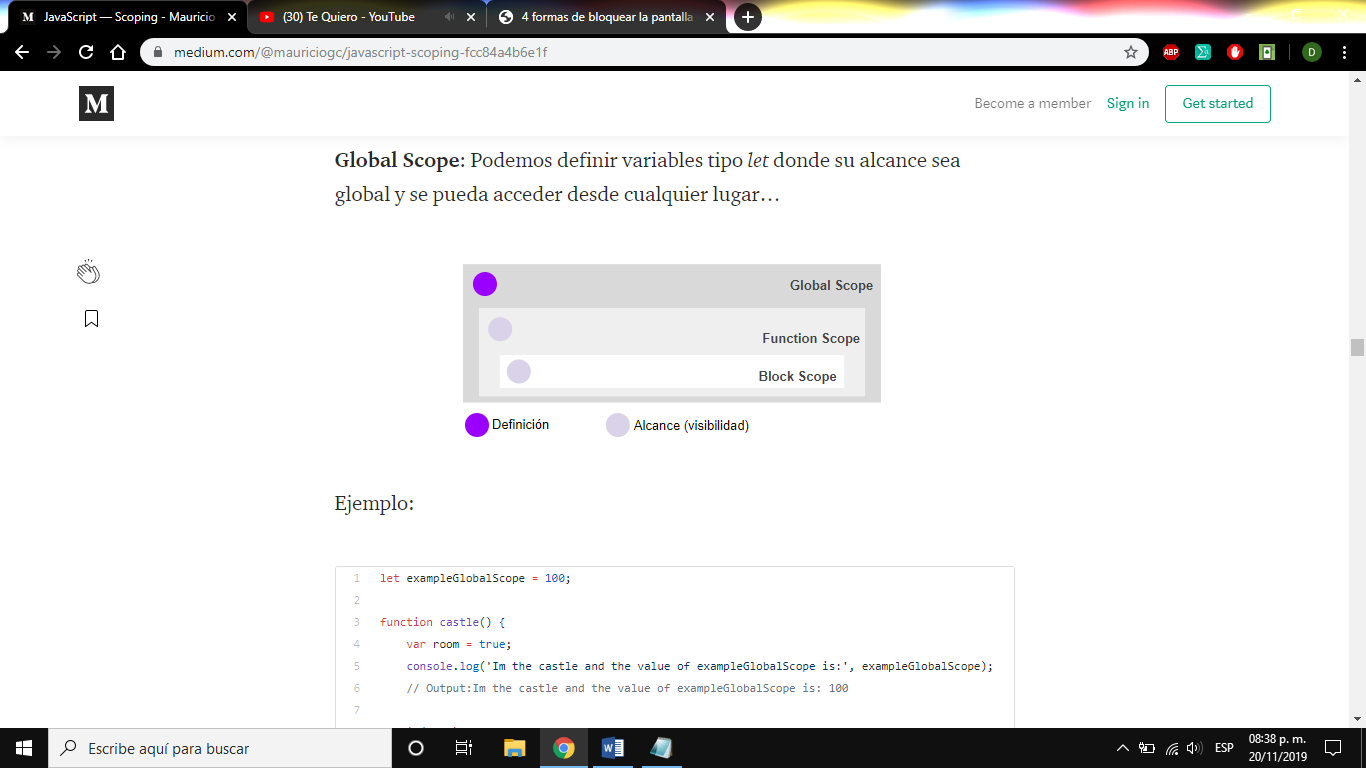
Recordemos que **JavaScript** es un **lenguaje** de **alcance léxico**, esto quiere decir que la **herencia** es de **afuera hacia adentro;** es importante recordarlo ya que las **variables** que están fuera de una **función** pueden ocuparse tanto dentro como **fuera** de ella, pero **no al revés** (una variable dentro de una función no la podemos ocupar fuera).

## Let

Se dice que *let* es el nuevo *var*, pero su **scope** está **definido** por el **ámbito** de **bloque** (block-scoped), esto quiere decir que el **alcance** de la **variable** estará **definido dentro** de un **bloque de código.**

Su **Global Scope** y **Function Scope** funciona **CASI** igual que *var*, la **GRAN** diferencia, es la declaración que esta **dentro** del **ámbito Lexical-Enviroment**, esto quiere decir que su alcance solo será en el bloque que se ha creado y al momento de ser ejecutada.

**Global Scope:** Podemos definir variables de tipo *let* donde su alcance sea global y se pueda acceder desde cualquier lugar.



let exampleGlobalScope = 100;

function castle() {

var room = true;

console.log('Im the castle and the value of exampleGlobalScope is:', exampleGlobalScope);

// Output:Im the castle and the value of exampleGlobalScope is: 100

if(room) {

console.log('Im the room and the value of exampleGlobalScope is:', exampleGlobalScope);

// Output:Im the room and the value of exampleGlobalScope is: 100

}

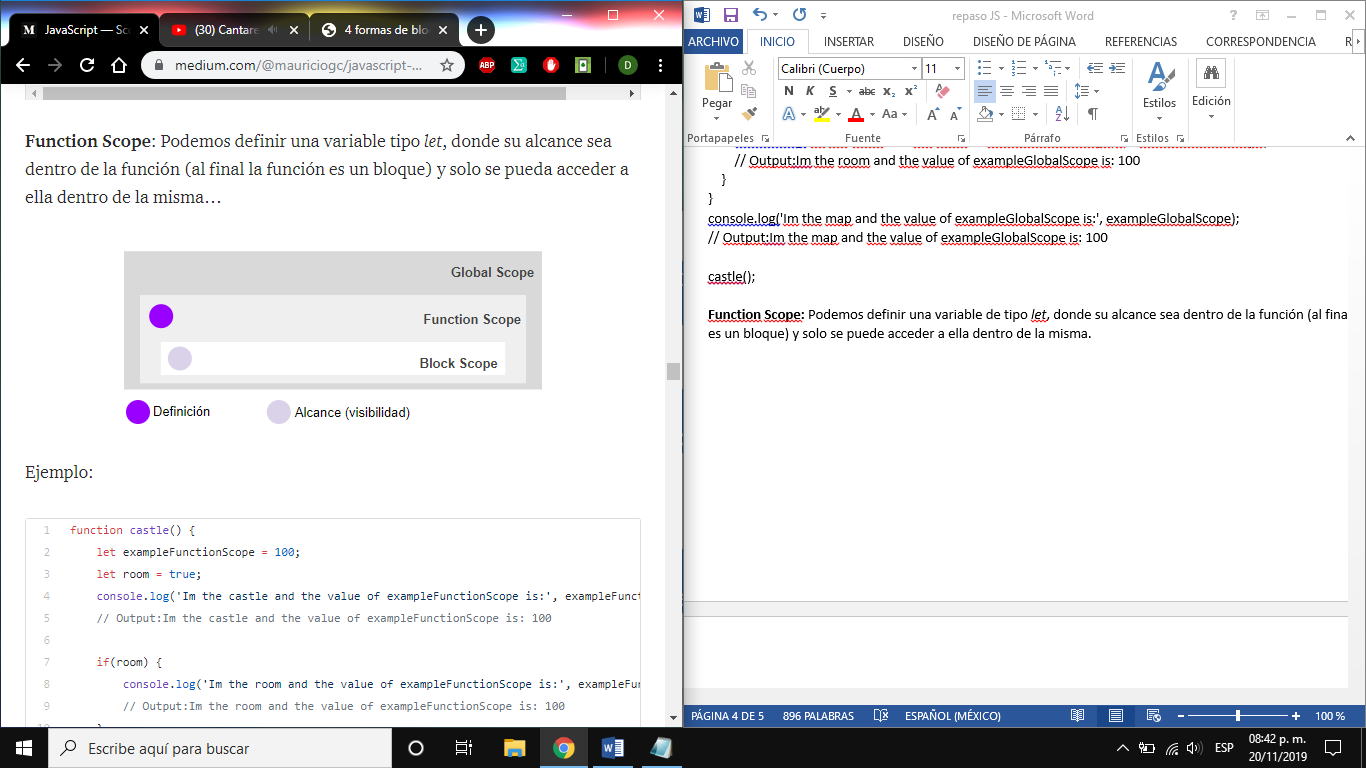
}

console.log('Im the map and the value of exampleGlobalScope is:', exampleGlobalScope);

// Output:Im the map and the value of exampleGlobalScope is: 100

castle();

**Function Scope:** Podemos definir una variable de tipo *let*, donde su alcance sea dentro de la función (al final la función es un bloque) y solo se puede acceder a ella dentro de la misma.



function castle() {

let exampleFunctionScope = 100;

let room = true;

console.log('Im the castle and the value of exampleFunctionScope is:', exampleFunctionScope);

// Output:Im the castle and the value of exampleFunctionScope is: 100

if(room) {

console.log('Im the room and the value of exampleFunctionScope is:', exampleFunctionScope);

// Output:Im the room and the value of exampleFunctionScope is: 100

}

}

castle();

Si nosotros intentamos acceder a la variable *exampleFunctionScope* fuera de la función nos arrojaría un **error** y **rompería** el **código:**

function castle() {

let exampleFunctionScope = 100;

let room = true;

console.log('Im the castle and the value of exampleFunctionScope is:', exampleFunctionScope);

if(room) {

console.log('Im the room and the value of exampleFunctionScope is:', exampleFunctionScope);

}

}

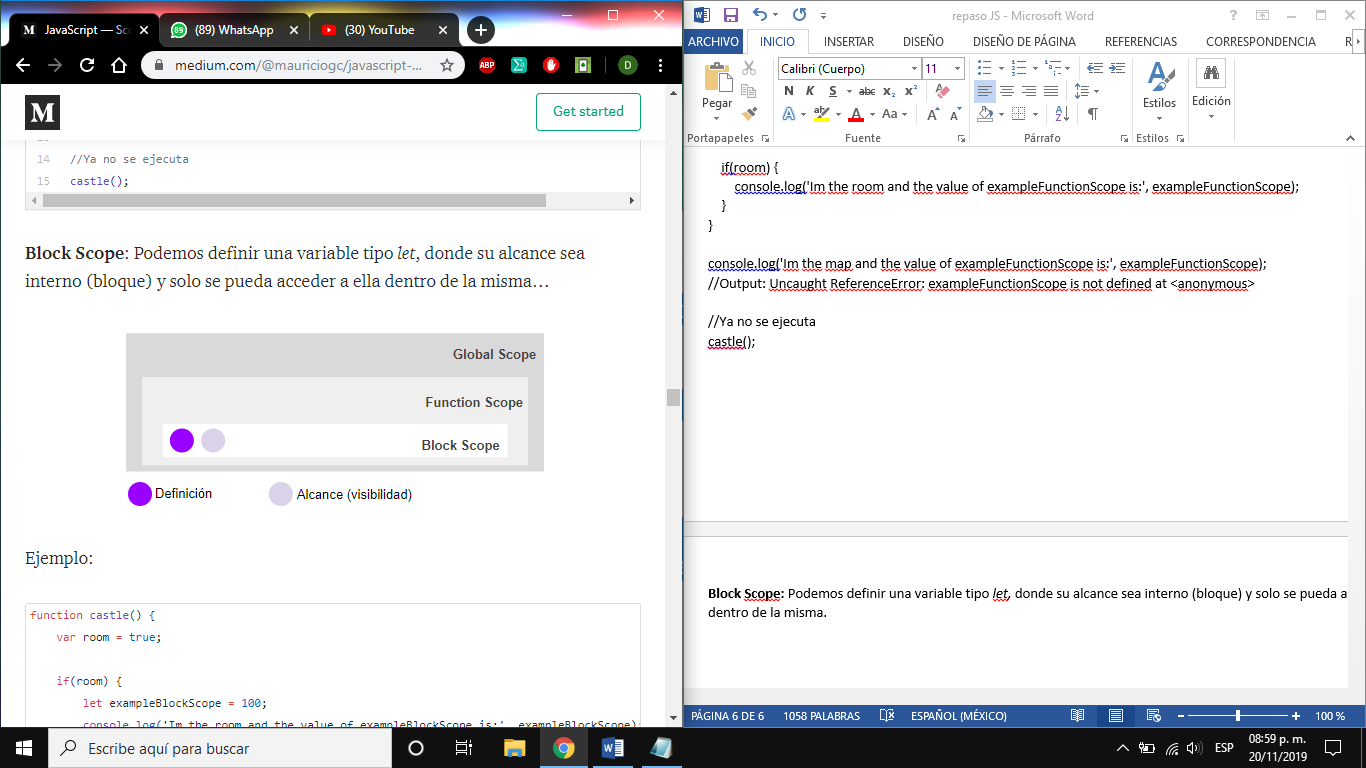
console.log('Im the map and the value of exampleFunctionScope is:', exampleFunctionScope);

//Output: Uncaught ReferenceError: exampleFunctionScope is not defined at <anonymous>

//Ya no se ejecuta

castle();

**Block Scope:** Podemos definir una variable tipo *let,* donde su alcance sea interno (bloque) y solo se pueda acceder a e..a dentro de la misma.



function castle() {

var room = true;

if(room) {

let exampleBlockScope = 100;

console.log('Im the room and the value of exampleBlockScope is:', exampleBlockScope);

// IOutput: Im the room and the value of exampleBlockScope is: 100

}

}

castle();

hemos definido la variable dentro del *if(room)*, donde la consola nos regresa el valor esperado como mencionamos anteriormente, su alcance es *block scope (if, for, etc)*.

function castle() {

var room = true;

console.log('Im the castle and the value of exampleBlockScope is:', exampleBlockScope);

// Uncaught ReferenceError: exampleBlockScope is not defined

//Ya no se ejecuta

if(room) {

let exampleBlockScope = 100;

console.log('Im the room and the value of exampleBlockScope is:', exampleBlockScope);

}

console.log('Im the castle (2) and the value of exampleBlockScope is:', exampleBlockScope);

}

castle();

Automáticamente nos va a salir un **error** y va a **romper** el **código**, ya que la variable no existe fuera del bloque de código.

## Const

*Const* tiene la misma **funcionalidad** y **alcance** que tienen *let* **solo que no nos permite** re-asignar su valor (**primitive**).

const a = 100;

console.log(a);

// … Mucho código

a = 10; // Uncaught TypeError: Assignment to constant variable.

Si intentamos cambiar el valor de una **const** en una consola nos va a mostrar un error, ya que la variable es **constante** y no puede cambiarse su valor.

Mencionamos que solo afecta a los **primitivos,** ya que los **objetos SI** podemos cambiar sus valores (hijos), e inclusive agregar más propiedades.

const a = {

friend: 'Mauricio'

};

console.log(a.friend); // Mauricio

a.friend = 'Benjamin';

console.log(a.friend); // Benjamin

a.otherFriend = 'David';

console.log(a.otherFriend); // David

Podemos observar que declaramos a *friend* como *Mauricio*, después le hemos cambiado su valor a *Benjamin* e inclusive pudimos agregar una nueva propiedad (a.otherFriend = ‘David’).

Entonces **const = inmutabilidad**, si bien las **variables no se pueden reasignar**, como tal **NO** las hace realmente **inmutables,** ya que como hemos visto el contenido de los **objetos** podemos **cambiarlos** sin ningún problema.

La única forma que nos puede salir **error** es si la variable “a” **cambiamos** el **tipo** de **dato**:

const a = {

friend: 'Mauricio'

};

a = []; // Uncaught TypeError: Assignment to constant variable.

# Ventajas de usar let y const

Una de las **ventajas** de usar *let* y *const* es en la declaración de las variables.

var a = 100;

console.log(a); //100

// … Mucho código

var a = 'Hola';

console.log(a); //Hola

Este es un error muy común en los desarrolladores y más cuando varias personas trabajan en la misma aplicación, un desarrollador ha declarado *a = 100*, y después de mucho código llega otro desarrollador y declara la misma variable per con otro valor *a = ‘Hola’*, y si ese valor más abajo se ocupa (antes se esperaba un *number*, pero ahora va a regresar un *string*), y en producción detectar estos errores es muy complicado, por lo que de una vez más *let* y *const* vienen al rescate.

let a = 100;

console.log(a);

// … Mucho código

let a = 'Hola'; // Uncaught SyntaxError: Identifier 'a' has already been declared

console.log(a);

Si se llega a declarar la misma variable dentro de un mismo **block-scope**, la consola nos va a mostrar un **error**, ya que la variable *a* ya fue declarada antes.

*No olvidemos que let y const está definido por el ámbito de bloque block-scoped.*

Veamos el siguiente ejemplo:

{

let a = 1;

const b = 10;

var c = 100;

console.log(a); // 1

console.log(b); // 10

console.log(c); // 100

}

console.log(a); // ReferenceError

console.log(b); // ReferenceError

console.log(c); // 100

Como mencionamos anteriormente *let* y *const* su **ámbito** es **block-scoped** mientras que *var* es su **contexto de ejecución.**

let a = 20;

{

let a = 100;

console.log(a); // 100

}

console.log(a); //20

En el ejemplo anterior tenemos decladara dos veces la variable *a*, pero en **diferentes ámbitos**, veamos otro ejemplo:

for (let i = 0; i < 10; i++) {

console.log(i); //output 1,2,3,4,5,6,7,8,9, 10

}

console.log(i); //ReferenceError: i is not defined

En el ejemplo de arriba, la variable *i* está **definida** con *let* **solo existe dentro** del ciclo *for*, por lo que la consola que está **dentro** de la misma **SI** imprime el valor de *i*, mientras que la consola que está **fuera** imprime un **error** ya que la variable *i* **no** está **definida.**

# Block-Scoped Functions

Al igual que *let* y *const*, las funciones adoptan el estándar **block scoped**, en **ES5** teníamos que emular el **scope** de la función con una **IIFE** (Immediately-invoked function expressions).

// Con ES5

// Emulamos el scope con IIFE

(function () {

var foo = function () {

return 1;

};

console.log(foo() === 1); // true

(function () {

var foo = function () {

return 2;

};

console.log(foo() === 2); // true

})();

console.log(foo() === 1); // true

})();

// Con ES6

//Por funciones de expresión

{

let foo = function () {

return 1;

};

console.log(foo() === 1); // true

{

let foo = function () {

return 2;

};

console.log(foo() === 2); // true

}

console.log(foo() === 1); // true

}

// Por funciones de declaración

{

function foo() {

return 1;

};

console.log(foo() === 1); // true

{

function foo() {

return 2;

};

console.log(foo() === 2); // true

Como podemos observar con **ES6** es más legible y simple el código, ya que solo es necesario que encerremos nuestras funciones dentro del **block** { }.

# Buenas prácticas

Utilizar *let* y *const* para cada una de las variables declaradas ya que tiene sus ventajas:

* Más fácil agregar nuevas variables.
* No hay que estar cambiando el ; por , o viceversa.
* Cuando usamos el debugger es más fácil ir una a una.

// Mal

const items = getItems(),

foo = true,

bar = 'z';

// Mal, ya que al poner ; en foo, bar se vuelve global

const items = getItems(),

foo = true;

bar = 'z';

// Bien

const items = getItems();

const foo = true;

const bar = 'z';

Se recomienda **agrupar** primero los *const* y luego *let*; ya que es más fácil de leer y además facilita la reasignación de valores.

// Mal

let i;

const items = getItems();

let foo;

const team = true;

let bar;

// Bien

const team = true;

const items = getItems();

let foo;

let i;

let bar;

**No** hay que **encadenar variables**, ya que **JavaScript** las asigna de manera implícita como globales.

// Mal

(function example() {

// JS lo interpreta de la siguiente manera:

// let a = ( b = ( c = 1 ) );

let a = b = c = 1;

}());

console.log(a); // throws ReferenceError

console.log(b); // 1

console.log(c); // 1

// Bien

(function example() {

let a = 1;

let b = a;

let c = a;

}());

console.log(a); // throws ReferenceError

console.log(b); // throws ReferenceError

console.log(c); // throws ReferenceError

Si vas a declarar variables **ÚSALAS**

// Mal  -  Si solo resignas un valor

var y = 10;

y = 5;

// Mal

var z = 0;

z = z + 1;

// Solo declara los argumentos que vas a utilizar

function getX(x, y) {

return x;

}

*No OLVIDEMOS* que let y const *están definidas por* block-scoped.

// Cuando usamos var

function varTest() {

var x = 1;

if (true) {

var x = 2; // same variable!

console.log(x); // 2

}

console.log(x); // 2

}

// Lo mismo pero usando let (observar que el resultado es diferente)

function letTest() {

let x = 1;

if (true) {

let x = 2; // otra variable diferente a la de arriba

console.log(x); // 2

}

console.log(x); // 1

}