

# JavaScript Basic

### Guía #1: JavaScript Fundamentals

Esta guía hace parte del curso mencionado en el título y les permitirá estudiar con más detenimiento los conceptos de clase, además de servir como instrumento de la misma. La guía contará con código base, el desarrollo conceptual de los temas y enlaces externos para el estudio independiente.

## 1. Declaración de variables

Ya conocemos las variables, esos contenedores de valores que usamos todo el tiempo al programar. En **JavaScript** dichos contenedores pueden almacenar cualquier valor sin la necesidad de la declaración explicita del tipo de variable.

//Creación de la variable  
var spartans = 299;  
  
//Reasignación de la misma  
spartans = 300;  
  
//Imprime la variable spartans por consola  
console.log(spartans) //Output: 300

### 1.1 Nombramiento de Variables

En **JavaScript** los nombres de variables son conocidos como identifiers y estos siguen las siguientes normas:

* Los nombres de las variables únicamente podrán ser conformados por letras (a-z), números (0-9), símbolos de pesos ($) y guión bajo (\_).
* Los nombres no pueden contener espacios o tabulaciones.
* Los nombres de las variables no pueden iniciar con números.
* Los nombres responden a case sensitive.
* Los nombres no pueden ser una [palabra reservada de **JS**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Reserved_keywords_as_of_ECMAScript_2015).

### 1.2 Las tres maneras de declarar variables desde ES6

A continuación se disponen las tres palabras reservadas para la declaración de variables con JavaScript y las propiedades de cada una:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keyword** | **Scope** | **Hoisting** | **Reasignación** | **Redeclaración** |
| **var** | Function Scope | Sí | Sí | Sí |
| **let** | Block Scope | No | Sí | No |
| **const** | Block Scope | No | No | No |

### 1.3 Variable Scope (Contexto de una variable) & Closure (cierre)

El Scope en JavaScript hace referencia al contexto actual del código, siendo el mismo el que determina la accesibilidad de las variables. Existen dos tipos de contextos:

* **Variable global**: Toda variable declarada fuera de un bloque.
* **Variable local**: Toda variable declarada dentro de un bloque.

Entiendo un bloque como funciones, declaraciones if, y ciclos for y while.

//Estas variables son globales pues no están definidas dentro de ningún bloque.  
 var thanos = "RIP";  
   
 //Una función es un bloque  
 function viajeCuantico(){  
 //Esta variable es local pues está definida en el contexto de la función  
 var thanos = "inevitable";  
 console.log(thanos);  
 }  
   
 //Otra función  
 function viajeTradicionalEnElTiempo(){  
 viajeCuantico();  
 //Las funciones no comparten el scope  
 console.log(thanos); //Se asume el valor del identificador global  
 }  
   
 console.log(thanos); //Output: RIP  
 viajeCuantico(); //Output: inevitable  
 console.log(thanos); //Output: RIP  
 viajeTradicionalEnElTiempo(); //Output: RIP

### Closures

Para los casos en los que una función es definida dentro de otra, la función interna tendrá acceso a las variables del contexto de la que le contiene. Debido a lo anterior se dice que el alcance externo encierra (hace **enclose**) del nuevo alcance.

function addCuadrado(a,b) {

function cuadrado(x) {

return x \* x;

}

return cuadrado(a) + cuadrado(b);

}

a = addCuadrado(2,3); // retorna 13

b = addCuadrado(3,4); // retorna 25

c = addCuadrado(4,5); // retorna 41

Dado que la función interna forma un cierre, puede llamar a la función externa y especificar argumentos para la función externa e interna.

function fuerade(x) {

function dentro(y) {

return x + y;

}

return dentro;

}

resultado = fuerade(3)(5); // retorna 8

#### ¿Cuál sería el resultado de la impresión en consola del siguiente bloque de código?

for (var i=1; i<=5; i++) {  
 setTimeout(function(){  
 console.log("i: " + i);  
 },i\*1000);  
 }

# Acá iría una foto de tu cara pensando, pero por razones de presupuesto sólo verás este texto…

El resultado de la ejecución de dicho código será i: 6 i: 6 i: 6 i: 6 i: 6, esto debido al **closure**.

for (var i=1; i<=5; i++) {  
 setTimeout(function(){  
 console.log("i: " + i);  
 },i\*1000);  
 }

Al analizar a detalle el código nos encontramos con 3 nested scopes:

* global scope.
* setTimeout() scope.
* El scope de la función anónima que ejecuta la impresión en consola.

#### ¿Cómo solucionamos esto para que el resultado final sea: i: 1 i: 2 i: 3 i: 4 i: 5?

# Acá iría una foto de tu cara pensando, pero por razones de presupuesto sólo verás este texto…

#### Damas y caballeros, una cálida bienvenida a las IIFE

Las IIFE (Immediately invoked function expression) son funciones anónimas que se ejecutan inmediatamente previniendo el Hoisting y encerrando el scope desde la línea 2 para prevenir que la ejecución del ciclo for se ejecute antes del timeout.

for (var i=1; i<=5; i++) {  
 (function(i){  
 setTimeout(function(){  
 console.log("i: " + i);  
 }, i\*1000);  
 })(i);  
 }

Entendiendo entonces el contexto de las variables podemos ahora realizar un par de ejemplos con los tipos de declaración de variables:

### let vs var

var lentes = false;  
  
// Inicialización de la variable global personaje  
let personaje = "Clark Kent";  
  
if (!lentes) {  
 // Inicialización de variable en contexto de bloque  
 let personaje = "Superman";  
 console.log(`En ese momento al remover su lentes todos vieron que era ${personaje}.`);  
}  
  
console.log(`Debido a sus lentes ${personaje} pasaba desapercibido.`);

Ahora hagamos lo mismo pero usando **var** en todos los casos:

var lentes = false;  
  
// Inicialización de la variable global personaje  
var personaje = "Clark Kent";  
  
if (!lentes) {  
 // Inicialización de variable en contexto de bloque  
 var personaje = "Superman";  
 console.log(`En ese momento al remover su lentes todos vieron que era ${personaje}.`);  
}  
  
console.log(`Debido a sus lentes ${personaje} pasaba desapercibido.`);

### Redeclaración

Teniendo en cuenta la tabla previa, construiremos el siguiente ejemplo:

//Variable global a  
 let a = true;  
   
 if(true){  
 var a = false; //Error, se intentó redeclarar a  
 }

//Variable global a  
 let a = true;  
   
 if(true){  
 let a = false; //Válido pues la variable a declarada será local  
 }

### Reasignación

Teniendo en cuenta la tabla previa, construiremos el siguiente ejemplo:

//Variable global a  
 const a = true;  
   
 if(true){  
 a = false; //Error, se intentó reasignar unas constante  
 }

**Bueno, es momento de que ustedes analicen este caso:**

//Variable global a  
 const a = true;  
   
 for(let i = 0; i < 10; i++){  
 const a = i;  
 console.log(a);  
 }

#### Venga… sabiendo todo esto, ¿se acuerdan del ciclo for con el setTimeout?

for (let i=1; i<=5; i++) {  
 setTimeout(function(){  
 console.log("i: " + i);  
 }, i\*1000);  
 }

### 1.4 Hoisting (Elevación)

Este concepto está directamente relacionado con el contexto de ejecución de JavaScript en su **fase de creación** y **fase de ejecución**. y cabe resaltar con no se aplica estrictamente a su definición, que plantea que las declaraciones tanto de variables como de las funciones son físicamente movidas al comienzo del código, en **JavaScript** en lugar de moverlas físicamente, se les realiza una asignación en memoria durante la **fase de compilación**.

A continuación un ejemplo de Hoisting y cómo este nos permite ejecutar una función aunque aún no esté fisicamente declarada:

hoisting();  
   
 function hoisting(){  
 console.log("Ya estaba cargada en memoria");  
 }

Ahora un más complejo:

var x = 5;  
  
 (function () {  
 console.log("x:", x); // no obtenemos '5' sino 'undefined'  
 var x = 10;  
 console.log("x:", x); // 10  
 }());

Reescribamos el ejemplo como **JavaScript** realmente lo ejecuta:

//variable global x  
 var x = 5;  
  
 (function () {  
 var x; // Se elevó la declaración debido al bloque  
 console.log("x:", x); // undefined  
 x = 10;  
 console.log("x:", x); // 10  
 }());

### Hoisting y la inicialización de variables

var x = 1; // Inicializa x  
 console.log(`${x} ${y}`); // '1 undefined'  
 var y = 2; // Inicializa y

Que implicitamente sería:

var x = 1; // Inicializa x  
 var y;// Se elevó la declaración  
 console.log(`${x} ${y}`); // '1 undefined'  
 y = 2; // Inicializa y

## 2. Tipos

Javascript tiene seis tipos primitivos: Sin definir (undefined), Nulo (null), Lógicos (boolean), Numérico (number), Cadena (string), Símbolo (symbol). Todos los demás tipos son objetos (Object): Array, Date, Promise, etc.

**undefined**

console.assert( typeof undefined === 'undefined' );

console.assert( undefined instanceof Object === false );

console.assert( Object.prototype.toString.call(undefined) === '[object Undefined]' );

// TypeError: Cannot read property "constructor" of undefined

// console.assert( undefined.constructor.name );

Cualquier variable no definida tiene como valor un tipo undefined. Este valor no es un objeto y la mejor estrategia para identificarlo es utilizar typeof o simplemente myVar === undefined ya que existe una variable global con el nombre undefined.

**null**

var testNull = null;

console.assert( typeof testNull === 'object' );

console.assert( testNull instanceof Object === false );

console.assert( Object.prototype.toString.call(testNull) === '[object Null]' );

// TypeError: Cannot read property "constructor" of null

console.assert( testNull.constructor.name );

Como se puede comprobar null es un objeto según typeof, pero por otra parte no es una instancia de Object. Esta es una circunstancia peculiar que nos obliga a un tratamiento específico de null, ya que la única forma de comprobar si un valor es nulo, es compararlo con el literal null:

var testNull = null;

console.assert( (testNull === null) );

**symbol**

var mySymbol = Symbol();

console.assert( typeof mySymbol === 'symbol' );

console.assert( mySymbol instanceof Object === false );

console.assert( Object.prototype.toString.call(mySymbol) === '[object Symbol]' );

console.assert( mySymbol.constructor.name === 'Symbol' );

En ES6 disponemos de un nuevo tipo primitivo, los símbolos (symbol). Estos tipos de datos son utilizados como claves en objetos y mapas de datos sin que puedan ser convertidos a otros tipos, por lo que para poder acceder a estos elementos tendremos que tener una referencia al símbolo con el que se creó.

Cada vez que creamos un símbolo obtenemos un valor único, por lo que son de mucha utilidad a la hora de crear constantes con valores únicos y asegurarnos que no se va a producir confusiones por valores duplicados.

Existen algunos símbolos conocidos y que se utilizan para acceder a algunas características avanzadas del lenguaje como Symbol.iterator que nos permite acceder al método interno que devuelve un objeto iterador de un objeto Array, String, Map o Set.

En todos los casos, la mejor forma de saber si un valor es de tipo symbol es utilizar typeof.

**Lógico (boolean), Numérico (number) y cadena (string)**

En estos tres casos hay que diferenciar los tipos primitivos (normalmente descritos en minúsculas) que se crean por medio de las expresiones literales para estos tipos, y los objetos de igual nombre (normalmente escritos con la primera letra en mayúscula) que se crean llamando a los constructores con new seguido del constructor. Estos objetos son wrappers de los tipos primitivos y ofrece propiedades y métodos para el manejo de este tipo de valores.

Es importante comprender que Javascript realiza una conversión implícita cuando utilizamos un valor primitivo con propiedades o métodos, por ejemplo, cuando consultamos el tamaño de una cadena "text".length. En estos casos el objeto wrapper es creado internamente para realizar la invocación al método o propiedad y posteriormente eliminado.

Como podemos ver las diferentes estrategias funcionan de forma diferente en el caso de los tipos primitivos y en el caso de los objetos wrapper.

// Primitive boolean

var testBoolean = true;

console.assert( typeof testBoolean === 'boolean' );

console.assert( testBoolean instanceof Object === false );

console.assert( testBoolean instanceof Boolean === false );

console.assert( Object.prototype.toString.call(testBoolean) === '[object Boolean]' );

console.assert( testBoolean.constructor.name === 'Boolean' );

// Object Boolean

var testBooleanObject = new Boolean(true);

console.assert( typeof testBooleanObject === 'object' );

console.assert( testBooleanObject instanceof Object === true );

console.assert( testBooleanObject instanceof Boolean === true );

console.assert( Object.prototype.toString.call(testBooleanObject) === '[object Boolean]' );

console.assert( testBooleanObject.constructor.name === 'Boolean' );

// Primitive type number

var testNumber = 10;

console.assert( typeof testNumber === 'number' );

console.assert( testNumber instanceof Object === false );

console.assert( testNumber instanceof Number === false );

console.assert( Object.prototype.toString.call(testNumber) === '[object Number]' );

console.assert( testNumber.constructor.name === 'Number' );

// Object Number

var testNumberObject = new Number(10);

console.assert( typeof testNumberObject === 'object' );

console.assert( testNumberObject instanceof Object === true );

console.assert( testNumberObject instanceof Number === true );

console.assert( Object.prototype.toString.call(testNumberObject) === '[object Number]' );

console.assert( testNumberObject.constructor.name === 'Number' );

// Primitive type string

var testString = 'text';

console.assert( typeof testString === 'string' );

console.assert( testString instanceof Object === false );

console.assert( testString instanceof String === false );

console.assert( Object.prototype.toString.call(testString) === '[object String]' );

console.assert( testString.constructor.name === 'String' );

// Object String

var testStringObject = new String('text');

console.assert( typeof testStringObject === 'object' );

console.assert( testStringObject instanceof Object === true );

console.assert( testStringObject instanceof String === true );

console.assert( Object.prototype.toString.call(testStringObject) === '[object String]' );

console.assert( testStringObject.constructor.name === 'String' );

Nos encontramos con un reto, ya que como se puede observar, las llamadas de Object.prototype.toString.call() y .constructor.name sobre los tipos primitivos convierte estos en objetos (aparecen con la primera letra en mayúscula) y por lo tanto nos pueden confundir. En general los tipos de datos primitivos funcionan bien con typeof.

## 3. Tiempo de repasar codeando

En este [link](Entrenamiento%20en%20codewars.docx) pueden acceder a la serie de JavaScript Basic:

* Templates literales
* Funciones
* Condicionales & operador ternario
* Ciclos
* Objetos
* Manipulación de los objetos Number & String
* Bonus: Introducción a clases con ES6