**Curso Básico de JavaScript**

**Introducción a JavaScript**

**1.- ¿Qué es JavaScript?**

* **¿Cómo nace Javascript?**
  + Nace con la necesidad de generar dinamismo en las páginas web y que a su vez los usuarios y las empresas pudieran interactuar unos con otros.
* **¿Qué es Javascript?**
  + Es un lenguaje interpretado, orientado a objetos, débilmente tipado y dinámico.
* **Orientado a objetos:**
  + Permite generar objetos globales que son conjuntos de características para agrupar distintos elementos de nuestro sistema que a la vez forman parte de una base de datos y todos estos elementos pueden ser ocupados para mostrar información individual.
* **Débilmente tipado:**
  + Los tipos de datos no están bien definidos en el leguaje y permite por ejemplo operaciones entre numerosos y letras porque el lenguaje asume tipos de datos que no necesariamente son los que se quiere representar.

|  |
| --- |
| 4 + "7"; // 47  4 \* "7"; // 28  2 + true; // 3  false - 3; // -3 |

* **Dinámico:** 
  + Corre directamente en la etapa de Runetime sin una etapa de compilación previa. Esto permite probar nuestro código inmediatamente; pero también es lo que hace que los errores se muestren hasta que se ejecuta el programa.
* **¿Realmente es un lenguaje interpretado?**
  + Interpretado quiere decir que el leguaje va a ir leyendo el código linea por linea en el tiempo de ejecución y probando que todo funcione en el proceso con la interacción con el programa y si encuentra un error en ese momento lo marca.
  + Una vez que el navegador obtiene el código js el motor de js lo procesa convirtiéndolo a binario para que sea entendido por la maquina por lo que javascript es un leguaje compilado ejecutado.
* **Backwards & Forwards:**
  + **Forwards:** Ser compatible con versiones futuras quiere decir que al implementar funcionalidades nuevas del leguaje no daña nuestro código y se puede usan en versiones anteriores del motor de compilación.
  + **Backwards:** Todas las funciones nuevas del lenguaje no rompen el entorno de trabajo actual pero no las puedes usar inmediatamente hasta que no se estandaricen.
  + Para hacer este cross versión del lenguaje se utilizan compiladores como BABEL que permite utilizar nuevas funcionalidades del lenguaje javascript dentro de nuestro entorno actual de trabajo traduciendo las nuevas versiones de javascript a versiones anteriores standard que el navegador entiende para que puedan ser utilizadas.

**2. ¿Por qué JavaScript?**

* JavaScript tiene una **comunidad enorme** de desarrolladores que te pueden ir ayudando a generar diferentes cosas.
* Si solo estuvieras interesado en trabajar **aplicaciones web** tienes muchos frameworks y librerías construidas en JavaScript que te van a ayudar a hacer proyectos de forma mucho más rápida, eficiente y robusta (Angular, View, React,entre otros)
* Si no quieres trabajar solo en aplicaciones Web puedes utilizar JavaScript con un framework que se llama React Native para **poder construir aplicaciones nativas** como Android y IOS.
* Puedes construir **aplicaciones de escritorio** con JavaScript, usando un framework llamado Electron, pueden correr en Mac o Windows.
* También puedes trabajar en la parte del **Back-end** o \*\*IOT \*\*(Internet Od Things) es un concepto que se refiere a una interconexion digital de objetos cotidianos con Internet. Esto con un Framework llamado NodeJS, el cual es un entorno de ejecución de JavaScript que corre directamente en el Back-end.
* **WebAssembly:** es un nuevo tipo de código que puede ser ejecutado en navegadores modernos es un lenguaje de bajo nivel, similar al lenguaje ensamblador, con un formato binario compacto que se ejecuta con rendimiento casi nativo y provee un objetivo de compilación para lenguajes como C/C++ y Rust que les permite correr en la web. También está diseñado para correr a la par de JavaScript, permitiendo que ambos trabajen juntos.

**3. Elementos de un lenguaje de Programación: Variables, Funciones y Sintaxis:**

* **PRIMITIVOS:** Son los valores básicos que se utilizan para poder generar códigos.
  + **Número:** 
    - Data sólo de tipo numérico, ejem: 1 2 3 4, etc.
  + **Strig:** 
    - Texto, se utiliza entre comillas dobles.
    - “Haciendo esto”, “Diego Hernández”
  + **Boolean o Booleano:** 
    - Son valores creados por George Boolean dentro de lenguajes de programación para poder validar cosas y/o desiciones, por ello se utilizan: true (1) o false (0).
  + **Empty values:** 
    - null o undefined,
    - se les conoce como valores “placeholders”, pero son valores reservados para un tipo de valor que queda faltante en memoria.
    - También pueden ser tomados como valores de errores.
* **NO PRIMITIVOS O VALORES TIPO OBJETOS:**
  + **Array:** 
    - Se genera ocupando corchetes, los cuales dentro ocupan valores primitivos,
    - ejem: [1,2,3] convirtiéndolos en valores tipo objeto.
  + **Valor tipo Objeto:** 
    - Se generan con corchetes, los cuales dentro ocupan una data que se transforma en objeto,
    - ejem de sintaxis: {nombre: “Diego”}.

**4. Variables en JavaScript**

* **Var:** Era la forma en que se declaraban las variables hasta ECMAScript 5. Casi ya no se usa porque es de forma global y tiene las siguientes características:
  + Se puede reinicializar: osea todas las variables se inicializan, por ejemplo:
    - Var pokemonType = ‘electric’ entonces reinicializar es:
    - Var pokemonType = ‘grass’ osea la misma variable con diferentes datos el último dato predomina.
  + Se puede reasignar: osea la variable ya inicializada le reasignamos otro valor por ejemplo: inicializamos la variable: Var pokemonType = ‘electric’ ahora la reasignamos pokemonType = ‘grass’ ya no va var.
  + Su alcance es función global: osea inicializamos la variable, pero la podemos llamar desde cualquier bloque (una llave abierta y una cerrada {}) pero hay que tener mucho cuidado con ello ya que puede haber peligro, no es recomendable usar VAR.
  + const y let es la forma en que se declaran las variables a partir de ECMAScript 6,
* **const:** sirve para declarar variables que nunca van a ser modificadas:
  + No se puede reinicilizar:
    - Es una const única no puede haber otra inicializada con el mismo nombre. const pokemonType = ‘electric’ no puede haber:
      * const pokemonType = ‘grass’
  + No se pude re asignar:
    - una vez que la hayamos inicializado no la podemos reasignar solo con su nombre: const pokemonType = ‘electric’ no puede ejecutarse:
      * pokemonType = ‘grass’
  + No es inmutable: osea no puede cambiar con objetos:
* **Let:** Son variables que pueden ser modificadas, se pueden cambiar:
  + No se puede reinicilizar:
    - Es una const única no puede haber otra inicializada con el mismo nombre. let pokemonType = ‘electric’ no puede haber:
      * let pokemonType = ‘grass’
  + Se puede reasignar:
    - Osea la variable ya inicializada le reasignamos otro valor por ejemplo: inicializamos la variable: let pokemonType = ‘electric’ ahora la reasignamos pokemonType = ‘grass’
  + Su contexto de es bloque:
    - Solo funciona dentro de un bloque {}, fuera de ello no.

**5. Funciones en JavaScript**

* Las funciones son las tareas que va a llevar a cabo el navegador.
* Ambas pueden llevar parámetros, que son los datos que necesitan para ejecutarse.
* Cada parámetro va separado por una coma.
* Cada instrucción que tenga la función debe terminar con ;
* Si queremos que una función nos dé un numero o dato tenemos que usar la siguiente sintaxis:

|  |
| --- |
| *return El dato que queremos que nos dé;* |

* Para ejecutar las funciones debemos usar la siguiente sintaxis:

|  |
| --- |
| *Nombre de la funcion(Parametros función);* |

* **Existen 2 tipos de funciones**

1. ***Funciones Declarativas (function declaration / function statement):*** 
   1. Las funciones declarativas tienen la siguiente sintaxis:

|  |
| --- |
| *function Nombre de la función (Parámetros de la función) {*  *Instrucciones*  *}* |

* 1. Un ejemplo de una función puede ser una suma:

|  |
| --- |
| function suma (a,b) {  return a+b;  }  suma(4,5); |

***2) Expresiones de función (function expression / funciones anónimas):***

Las funciones de expresión son aquellas que guardamos en una variable, por lo tanto, no es necesario nombrarlas y tienen la siguiente sintaxis:

|  |
| --- |
| *var Nombre de la variable = function(Parametros){*  *Instrucciones*  *}.* |

Un ejemplo de una función de expresión sería:

|  |
| --- |
| *var suma = function(a,b){*  *return a+b;*  *}*  suma(4,5); |

**6. ¿Qué es una función declarativa y una expresiva?**

* **Funciones Declarativas:**
  + En las funciones declarativas, utilizamos la palabra reservada function al inicio para poder declarar la función:

|  |
| --- |
| function saludar(nombre) {  console.log(`Hola ${nombre}`);  }  saludar('Diego'); |

* **Expresión de función:**
  + En la expresión de función, la declaración se inicia con la palabra reservada var, donde se generará una variable que guardará una función anónima.

|  |
| --- |
| var nombre = function(nombre){  console.log(`Hola ${nombre}`)  }  nombre(‘Diego’); |

En la expresión de función, la función podría o no llevar nombre, aunque es más común que se hagan anónimas.

* **Diferencias:**
  + A las funciones declarativas se les aplica hoisting, y a la expresión de función, no. Ya que el hoisting solo se aplica en las palabras reservadas var y function.
  + Lo que quiere decir que, con las funciones declarativas, podemos mandar llamar la función antes de que ésta sea declarada, y con la expresión de función, no, tendríamos que declararla primero, y después mandarla llamar.

**Bases de Javascript**

**7.- Scope:**

* **variables globales:** Pueden ser accedidas desde un scope local o global. las variables globales son definidas fuera de las funciones (Scope global)
  + **Scope local:** Son aquellas variables definidas dentro del cuerpo de la función, estas son solo accedidas desde dentro de la misma función.
  + **Scope Global:** variables que pueden ser accedidas y procesadas por cualquier función dentro del código.



**8.- Hoisting:**

El **Hoisting** es un proceso del compilador de JavaScript, que consiste en que la declaración de las variables y las funciones son llevadas al inicio del código, sin importar su posición, para su procesamiento, sin embargo, la inicialización de las variables no es llevada al inicio del código para su compilación, sino solo su declaración, lo cual suele dar espacio a errores cuando se declara una variable sin inicializarla y se procesa en el código antes de haber llegado a su inicialización, lo cual nos suele dar una variable con valor undefined, ya que la variable sí fue almacenada en memoria, pero no se le asigno un valor hasta después de su ejecución.

|  |
| --- |
| saludo();  function saludo() {  console.log("Hola " + nombre);  }  var nombre = "Aaron"; |
| Hola undefined |

Debido a que como lo hemos dicho, la variable se tomó en cuenta y se le asigno memoria, sin embargo, el compilador no la inicializo y se le dio el valor de *undefined*, y con ese valor se ingresó a la función, y ya después de correr la función se le asigno el valor.

Este comportamiento se puede entender fácilmente si se comprenden estos dos puntos esenciales:

* Las funciones siempre se mueven arriba del scope. Por lo tanto, podemos elegir donde declararlas y usarlas.
* La declaración de las variables se mueve arriba del scope, pero no la asignación. Antes de usar una variable, habrá que crearla y asignarla.

En base al segundo punto, fue por eso que se cometió el error de usar la variable antes de inicializarla, pues sin problema el compilador le asigna memoria, pero no el valor hasta después.

Básicamente por pasos, lo que hizo el compilador fue esto:

Tenemos el código así:

|  |
| --- |
| saludo();  function saludo() {  console.log("Hola " + nombre);  }  var nombre = "Aaron"; |
| 1.- El compilador toma las funciones y variables y las “sube” en el codigo, sin inicializar variables: |
| var nombre;  function saludo() {  console.log("Hola " + nombre);  }  saludo();  var nombre = "Aaron"; |
| 2.- Le asigna memoria a la variable y le da el valor de *undefined* al suceder la asignacion de memoria. |
| var nombre = undefined;  function saludo() {  console.log("Hola " + nombre);  }  saludo();  nombre = "Aaron"; |

Y como hemos visto, la variable se asigna como undefined y posteriormente su utiliza al llegar a la linea:

|  |
| --- |
| saludos() |

pues ahi lo que hace es ejecutar la funcion que ya fue procesada, pero con un valor de la variable que aun no se le asigno, quedando como undefined.

|  |
| --- |
| 1.- Despues de correr la funcion, le asigna el valor correcto a la variable ya declarada: |
| var nombre = "Aaron";  function saludo() {  console.log("Hola " + nombre);  }  saludo(); |

Pero ya es demasiado tarde, pues la función ya fue ejecutada.

***Es por eso que se tiene como buena práctica declarar e inicializar tanto variables como funciones al inicio de nuestro programa, sin importar donde sean utilizadas, pues de esta manera se evita usarlas antes de ser inicializadas.***

Debemos saber que el hoisting solo sucede con las palabras claves var y function, por lo tanto, esto nos dice que solo se da en las versiones de ECMAScript 5 o Inferiores, ya que en la version 6 y superiores se permite la declaración de variables con let y const, que son dos nuevas variables que no activan hoisting.

[¿Qué es el hoisting?. En JavaScript, las declaraciones (por… | by Ana Martínez Aguilar | Medium](https://anamartinezaguilar.medium.com/qu%C3%A9-es-el-hoisting-327870f67b36)

[Hoisting - Glosario | MDN (mozilla.org)](https://developer.mozilla.org/es/docs/Glossary/Hoisting)

**9.- Coercion:**

* Coerción es la forma en la que podemos cambiar un tipo de valor a otro, existen dos tipos de coerción:
  + Coerción implícita:
    - Es cuando el lenguaje nos ayuda a cambiar el tipo de valor.
  + Coerción explicita:
    - Es cuando obligamos a que cambie el tipo de valor.

|  |
| --- |
| //Ejemplos de Coerción:  var a = 4 + "7"; //Convierte a 4 en un string y lo concatena con el "7", por esto regresa un string de valor "47"  4 \* "7"; //Convierte al "7" en un número y realiza la operación, por esto devuelve 28  var a = 20;  var b = a + ""; //Aquí concatenamos para convertir la variable a string (coerción implícita)  console.log(b); |
| var c = String(a); //Aquí obligamos a la variable a convertirse en string (coerción explícita)  console.log(c);  var d = Number(c); //Aquí obligamos a la variable a convertirse en número (coerción explícita)  console.log(d); |

**10.- Valores: Truthy y Falsy:**

**¿Que tipos por default son verdaderos y falsos?**

Usamos la función de JS que es Boolean() dentro del paréntesis ponemos el valor y nos dice si el mismo el False o True.

|  |
| --- |
| **–> Falsy**  Boolean() —> sin ningun valor es false  Boolean(0) —> false  Boolean(null) —> false  Boolean(NaN) —> false // NaN es Not and Number  Boolean(Undefined) —> false  Boolean(false) —> false  Boolean("") —> false  **–> Truthy**  Boolean(1) —> true //cualquier numero que no sea igual a cero es true  Boolean(“a”) —> true  Boolean(" ") —> true // siendo un espacio el valor es true  Boolean([]) —> true // un array nos da un true  Boolean({}) —> true // un objeto nos da el valor de true  Boolean(function() {}) —> true //una funcion tambien es true  Boolean(true) —> true  Todo esto lo vamos a usar en condiciones esto valida si es verdadero o falso para ejecutar cierta acción. |

**11.- Operadores: Asignación, Comparación y Aritméticos**

|  |
| --- |
| **//Operadores binarios:**  3 + 2 //Suma  50 - 10 // Resta  10 \* 20 //Multiplicación  20 / 2 //División  "Diego " + "De Granda" **//concatenación de strings**  **//Operadores unitarios:**  !false //negación de la negación = true  **//Operadores de asignación:**  var a = 1; //Asignamos un valor a la variable  **//Operadores para comparar:**  3 == "3"; //Compara los valores y devuelve "true" en este caso  3 === "3"; //Compara y valida los tipos y valores. Devuelve "falso" en este caso  5 < 3 //Compara y valida si el 5 es menor a 3  5 > 3 //Compara y valida si el 5 es mayor a 3  5 <= 6 //Compara y valida si el 5 es menor o igual al 6  5 >= 6 //Compara y valida si el 5 es mayor o igual al 6  a && b //Valida si ambas variables son verdaderas para que se cumpla la condición  a || b //Aquí se cumple la condición si alguna de las dos variables es verdadera  var edad = 40  edad++ //Incrementa el valor en 1  edad += 2 //Incrementa el valor por 2 |

**Condicionales:**

**12.- Condicionales: If – else – else if**

|  |
| --- |
| var edad = 18;  if(edad === 18){  console.log("Puedes votar, sera tu 1ra votacion");  }  else if(edad > 18){  console.log("Puedes votar de nuevo");  }  else{  console.log("Aun no puedes votar");  }  //condicion ? true: false;  var numero = 1;  var resultado = numero === 1 ? "Si soy un uno" : "No, no soy uno"; |

**13.- Switch**

|  |
| --- |
| var numero = 1;  switch (numero){  case 1:  console.log("Soy uno!");  break;  case 10:  console.log("Soy uno!");  break;  case 100:  console.log("Soy uno!");  break;  default:  console.log("Soy uno!");  break;  } |

**Arrays**

**14.- Arrays:** Su parte mas importante, son los índices y elementos.

|  |
| --- |
| var frutas = ["Manzana", "Platano", "Cereza", "Fresa"];  console.log(frutas);  console.log(frutas.length)  console.log(frutas[0]);  console.log(frutas[1]);  console.log(frutas[2]);  console.log(frutas[3]);  var masFrutas = frutas.push("Uvas");  console.log(masFrutas);  var ultimoFrutas= frutas.pop("Uvas");  console.log(ultimoFrutas);  var nuevaLogintud = frutas.unshift("Uvas");  console.log(nuevaLogintud)  var borrarFruta = frutas.shift("Uvas");  console.log(borrarFruta)  var nuevaLogintud = frutas.unshift("Uvas");  console.log(nuevaLogintud)  var borrarFruta = frutas.shift("Manzana");  console.log(borrarFruta)  var posicion = frutas.indexOf("Cereza");  console.log(posicion) |

**LOOPS**

**15.- Loops: For y for…of**

|  |
| --- |
| var estudiantes = ["Maria", "Sergio", "Rosa", "Daniel"];  function saludarEstudiantes(estudiantes){  console.log('Hola, ' + estudiantes);  }  //Normal  for(var i=0; i<estudiantes.length; i++){  saludarEstudiantes(estudiantes[i]);  }  //La nueva forma  for(var estudiante of estudiantes){  saludarEstudiantes(estudiante);  } |

**16.- LOOPS: WHILE**

|  |
| --- |
| var estudiantes = ["Maria", "Sergio", "Rosa", "Daniel"];  function saludarEstudiantes(estudiantes){  console.log('Hola, ' + estudiantes);  }  while (estudiantes.length > 0){  console.log(estudiantes);  var estudiante = estudiantes.shift();  saludarEstudiantes(estudiante);  } |
| (4) ["Maria", "Sergio", "Rosa", "Daniel"]  Hola, Maria  (3) ["Sergio", "Rosa", "Daniel"]  Hola, Sergio  (2) ["Rosa", "Daniel"]  Hola, Rosa  ["Daniel"]  Hola, Daniel |

**Objects:**

**17.- Objects:**

* JS es un lenguaje que está diseñado en un paradigma de objetos.

|  |
| --- |
| var miAuto = {  marca: "Toyota",  modelo: "Corolla",  año: 2020  } |
| console.log(miAuto.marca) //Toyota  console.log(miAuto.marca) //Toyota  console.log(miAuto.marca) //Toyota |

* Se pueden agregar propiedades que van a ser una

|  |
| --- |
| var miAuto = {  marca: "Toyota",  modelo: "Corolla",  año: 2020,  detallesDelAuto: function () {  console.log(`Auto ${this.modelo} ${this.año}`  );  } |

* ¿Quién es this?
  + Es una variable que hace referencia al objeto. En este caso: this = miAuto.

**18.- Función Constructora:**

Para evitar crear muchos objetos con los mismos atributos se usan los constructores

|  |
| --- |
| function auto (marca, modelo, annio){  this.marca = marca;  this.modelo = modelo;  this.annio = annio;  }  console.(auto) |
| var autoNuevo = new auto(“Tesla”, “Model 3”, 2020); |
| var autoNuevo2 = new auto(“Tesla”, “Model X”, 2018); |
| var autoNuevo3 = new auto(“Toyota”, “Corolla”, 2020); |

**Métodos de Arrays**

**19.- Métodos de recorridos de Arrays:**

|  |
| --- |
| **var** artículos = [  { nombre: “Bici”, costo: 3000 },  { nombre: “Tv”, costo: 2500 },  { nombre: “Libro”, costo: 320 },  { nombre: “Celular”, costo: 10000 },  { nombre: “Laptop”, costo: 20000 },  { nombre: “Teclado”, costo: 500 },  { nombre: “Audifonos”, costo: 17000 }  ]; |
| .**filter :**  Crea un nuevo array con los elementos del array que se pasó como parámetro y que cumplan con la condición definida |
| var articulosFiltrados = articulos.filter(function(articulo){  return articulo.costo <= 500  }); |
| 0: { nombre: “Libro”, costo: 320 },  1: { nombre: “Teclado”, costo: 500 }, |
| **.map :** crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos. |
| var nombreArticulos = artículos.map(function(articulo){  return articulo.nombre  }) |
| [“Bici” , “Tv”, “Celular”, “Laptop”, “Teclado”, “Audifonos” ] |

**20.- Recorriendo Arrays con .find() . forEach() y .some()**

|  |
| --- |
| **var** artículos = [  { nombre: “Bici”, costo: 3000 },  { nombre: “Tv”, costo: 2500 },  { nombre: “Libro”, costo: 320 },  { nombre: “Celular”, costo: 10000 },  { nombre: “Laptop”, costo: 20000 },  { nombre: “Teclado”, costo: 500 },  { nombre: “Audifonos”, costo: 17000 }  ]; |
| **.find():** Devuelve el primer valor que coincide de la colección. Una vez que coincida con el valor en los resultados, no verificará los valores restantes en la colección de matriz. |
| var encuentraArticulo = artículos.find(function(articulo){  return articulo.nombre === “Laptop”  }); |
| { nombre: “Laptop”, costo: 20000 }, |
| **foreach() :** Ejecuta lo que le definamos una vez por cada elemento de nuestro array |
| artículos.forEach(function(articulo){  console.log(articulo.nombre);  }); |
| bici  Tv  Libro  Celular  Laptop  Teclado  Audifonos |
| **some() :** Comprueba si al menos un elemento del array cumple con la condición que le damos. True - False |
| var articulosBaratos = artículos.some(function(articulo){  return articulo.nombre <= 700;  }); |
| articulosBaratos;  true; |
| **filter() :** Devuelve todos los elementos del array que cumplan con la condición dada |

**21.- Eliminando elementos de un Array:**

|  |
| --- |
| **.push():** El método .push() nos permite agregar uno o más elementos al final de un array.  A continuación, veremos un ejemplo aplicado con un array que contiene números: |
|  |
| Como podemos ver, al momento de ejecutar la función se agregan los números 6 y 7 al array.  Ahora revisemos un ejemplo con strings: |
|  |
| Como podemos ver, agregamos dos cadenas de strings al ejecutar la función donde tenemos txtArray.push(). Es decir, indico el array al que voy agregar elementos, uso el método .push(), y dentro de .push() indico los elementos que quiero agregar al string. Finalmente, el console.log() lo uso para revisar en la consola si esto sucedió o no. |
| **.shift():** |
| Ahora pasemos a la otra cara de la moneda donde necesitamos eliminar un elemento del array. .shift() eliminar el primer elemento de un array, es decir, elimina el elemento que esté en el índice 0. |
|  |
| Como vemos, luego de aplicar .shift() se eliminó exitosamente el primer elemento del array. ¿Y si quisiéramos eliminar el último elemento? Pasemos al bonus track de esta clase 🙌🏼. |
| **Bonus Track** |
| Si ya entendiste cómo funciona .shift() aplicar .pop() te será pan comido 🍞. El método .pop() eliminará el último elemento de un array. En este sentido, si tenemos un array de 5 elementos, pop() eliminará el elemento en el índice 4. Usemos el mismo ejemplo pero usando este método. |
|  |

**Próximos pasos con JavaScript**

**22.- Continúa con el Curso Práctico de JavaScript:**

* **Curso de JavaScript Engine (V8) y el Navegador**
  + https://platzi.com/clases/javascript-navegador
* **Curso de ECMAScript 6+**
  + https://platzi.com/clases/ecmascript-6/
* **Curso Práctico de JavaScript | Platzi**
  + https://platzi.com/clases/javascript-practico