**¿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.  
**Débilmente tipado**  
Se pueden hacer operaciones entre tipos distintos de datos (enteros con strings, booleanos con enteros, etc). Ejemplo:

4 + "7"; // 47

4 \* "7"; // 28

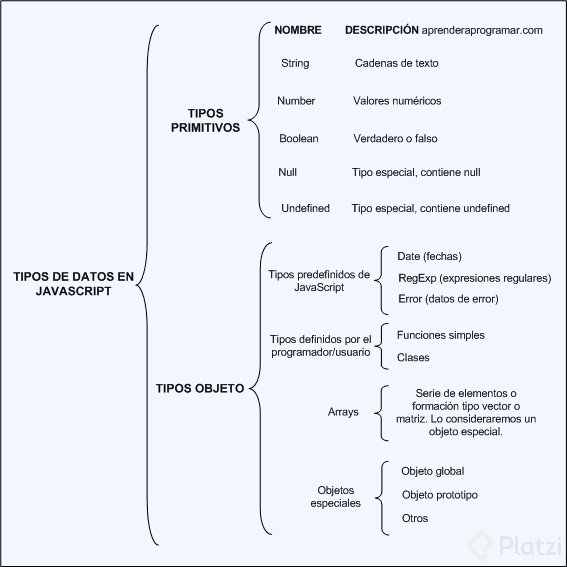
2 + **true**; // 3

**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 Javascript un lenguaje interpretado?**  
Si, y la razón es que le navegador lee linea por linea nuestro código el cuál le indica lo que tiene que hacer, sin la necesidad de compilar. Todo esto es controlado por el motor de Javascript V8 del navegador  
**Javascript es Basckwards Compatible**  
Todas las funciones nuevas que salen de Javascript no dañarán el trabajo ya hecho, pero no se podrá utilizar en nuestro entorno de trabajo inmediatamente. Para solucionar esto está **Babel** que permite utilizar las nuevas características del lenguaje pero lo transforma a una versión que el navegador pueda entender

1.- JavaScript tiene una **comunidad enorme** de desarrolladores que te pueden ir ayudando a generar diferentes cosas.

1. 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 mas rápida, eficiente y robusta (Angular, View, React,entre otros)
2. 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.
3. Puedes construir **aplicaciones de escritorio** con JavaScript, usando un framework llamado Electron, pueden correr en Mac o Windows.
4. 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.



VALORES

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”}.

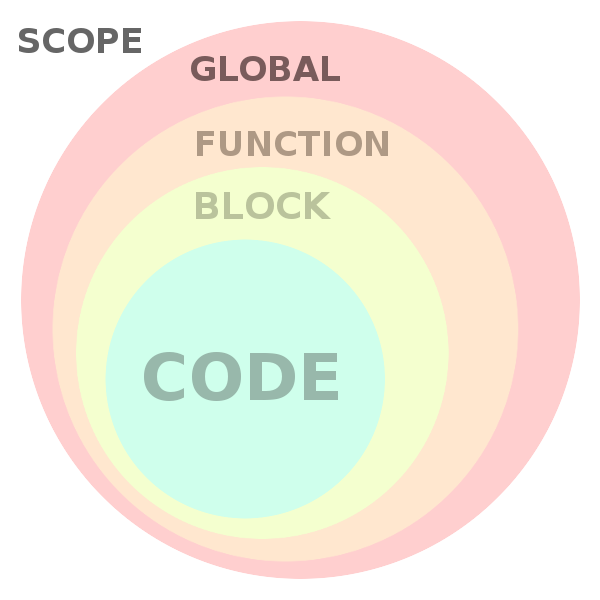
Dentro de JavaScript tenemos tres formas de declarar una variable las cuales son: **var**, **const** y **let.**

1. **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:

**o 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.  
**o 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  
**o 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,**

1. **const:** sirve para declarar variables que nunca van a ser modificadas:  
   **o 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’  
   o **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’  
   o **No es inmutable:** osea no puede cambiar con objetos:
2. **Let:** Son variables que pueden ser modificadas, se pueden cambiar:  
   **o 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’  
   o **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’  
   o **Su contexto de es bloque:** Solo funciona dentro de un bloque {}, fuera de ello no.



**Scope Global** : todo el mundo.  
**Scope Local** : Las Vegas, y lo que pasa en las vegas, se queda en Las Vegas.

Global -> Universo , seria como el todo , las variables son accesibles desde cualquier parte del código.  
Local -> Cada Planeta , un mundo distinto , las variables son accesibles en ese mundo , no es visible para los demás mundos.

Resumen : lo que es global se puede acceder desde cualquier parte de tu código , y lo pueden acceder cada mundo , lo que es local solo lo puede acceder cada mundo.😁

Realice un resumen sobre lo que yo entendí de lo que es el *hoisting*, aunque no sé que tan correcto sea lo que yo aprendi, igual se los dejo esperando les sea de ayuda a entender el concepto de mejor manera:

El **Hoisting** es un proceso del compilador de JavaScript, que consiste en que la *declaracion de las variables* y las *funciones* son llevadas al inicio del codigo, sin importar su posicion, para su procesamiento, sin embargo, la inicializacion de las variables no es llevada al inicio del codigo para su compilacion, sino solo su declaracion, lo cual suele dar espacio a errores cuando se declara una variable sin inicializarla y se procesa en el codigo antes de haber llegado a su inicializacion, 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 despues de su ejecución.

Aqui un ejemplo de esto:

saludo();

**function** **saludo**() {

console.log("Hola " + nombre);

}

**var** nombre = "Aaron";

El output de este codigo seria el siguiente:

Hola undefined

Debido a que como lo hemos dicho, la variable se tomo 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 ingreso a la funcion, y ya despues de correr la funcion se le asigno el valor.

Este comportamiendo se puede entender facilmente 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 mueven 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 cometio el error de usar la variable antes de inicializarla, pues sin problema el compilador le asigna memoria, pero no el valor hasta despues.

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

Tenemos el codigo asi:

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";

1. 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 funcion ya fue ejecutada.

***Es por eso que se tiene como buena practica 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 declaracion de variables con **let** y **const**, que son dos nuevas variables que no activan hoisting.

¿Qué es Hoisting?  
En JavaScript, las declaraciones (por ejemplo, de variables o funciones) se mueven al principio de su scope o ámbito. Este comportamiento se conoce como hoisting y es muy importante tenerlo en cuenta a la hora de programar para prevenir posibles errores.

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

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.

<h1>OPERADORES: ASIGNACION, LOGICOS Y ARITMETICOS</h1>

**Operador de asignacion**

| **Simbolo** | **Descripcion** |
| --- | --- |
| = | operador de asignacion |

**Operadores de comparacion**

| **Simbolo** | **Descripcion** |
| --- | --- |
| == | igual que |
| === | estrictamente igual que |
| > or >= or >== | mayor o mayor igual que |
| < or <= or <== | menor o menor igual que |
| != or !== | diferente que |

**Operadores aritmeticos**

| **Simbolo** | **Descripcion** |
| --- | --- |
| + | operador suma este se utiliza para concatener dos cadenas de texto. |
| - | operador resta |
| \* | operador de multicacion |
| / | operador de division |
| % | operador de modulo |
| \*\* | operador de potenciacion |

tambien se les conoce como operadores binarios. por que toman dos valores y generan un resultado.

**Operadores logicos**

| **Simbolo** | **Descripcion** |
| --- | --- |
| ! | NOT niega un valor |
| && | AND |
| || | OR |
|  |  |

Un **Array** es un tipo de estructura de datos, objeto. Puede guardar datos distintos dentro, guarda los datos en forma de lista.  
**.lenght** devuelve la longitud del array.  
**.push()** agrega elementos al final de array.  
**.pop()** elimina un elemento del array.  
**.unshift()** agrega un elemento al array, pero lo agrega en primer lugar.  
\*\*.shift() \*\*elimina el elemento que está en el inicio del array.  
\*\*.indexOf \*\* devuelve la posición de un elemento del array.

Sólo una pequeña observación, un array **no guarda “datos distintos dentro”**(**por ejemplo**: no se puede guardar string y enteros en un mismo array), sino **obligatoriamente todos sus elementos deben ser del mismo tipo**, saludos 😄

Ojo que .pop no elimina cualquiera elemento del array , siempre el ultimo

-----------------------------------------------------------

Los bucles pueden ejecutar un bloque de código varias veces. JavaScript admite diferentes tipos de bucles:

* for - recorre un bloque de código varias veces
* for/in - recorre las propiedades de un objeto
* for/of - recorre los valores de un objeto iterable
* while - recorre un bloque de código mientras se cumple una condición específica
* do/while - también recorre un bloque de código mientras se cumple una condición específica

A modo de recopilación:

* **find()** : Devuelve el primer elemento del array que cumpla con la condición dada
* **foreach()** : Ejecuta lo que le definamos una vez por cada elemento de nuestro array
* **some()** : Comprueba si al menos un elemento del array cumple con la condición que le damos
* **filter()** : Devuelve todos los elementos del array que cumplan con la condición dada  
  Acá te dejo la documentación de cada uno: **[find()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/find" \t "_blank)** - **[foreach()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/forEach" \t "_blank)** - **[some()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/some" \t "_blank)** - **[filter()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/filter" \t "_blank)**

Los cursos que recomienda al final.

* [Fundamentos de JavaScript](https://platzi.com/clases/fundamentos-javascript/)
* [V8 y el navegador](https://platzi.com/clases/javascript-navegador/)
* [ECMAScript 6+](https://platzi.com/clases/ecmascript-6/)