JavaScript

Hour of Code 2018

Universidad de Valladolid

27 Oct 2018

Qué es Javascript

Qué es Javascript

Javascript (también llamado JS) es un lenguaje de programación originado en 1995.

Qué es Javascript

Javascript (también llamado JS) es un lenguaje de programación originado en 1995.

Es conocido principalmente por su uso en desarrollo web, permitiendo que un usuario pueda interaccionar con una pagina web.

Qué es Javascript

Javascript es un lenguaje...

- interpretado

Qué es Javascript

Javascript es un lenguaje...

- interpretado
- que usa tipado débil

Qué es Javascript

Javascript es un lenguaje...

- interpretado
- que usa tipado débil
- dinámico

Javascript es interpretado

Javascript es interpretado

Un lenguaje **interpretado** es aquel que es traducido a código máquina a medida que se ejecuta, mientras que un lenguaje **compilado** es aquel que se traduce a código máquina antes de ejecutarse.

Javascript es interpretado

Un lenguaje **interpretado** es aquel que es traducido a código máquina a medida que se ejecuta, mientras que un lenguaje **compilado** es aquel que se traduce a código máquina antes de ejecutarse.

Javascript es un lenguaje interpretado.

Javascript usa tipado débil

Javascript usa tipado débil

Un lenguaje que usa **tipado débil** no necesita declarar el tipo de sus variables explícitamente, mientras que un lenguaje que usa **tipado fuerte** necesita declarar el tipo que va a almacenar una variable de antemano.

Javascript usa tipado débil

Un lenguaje que usa **tipado débil** no necesita declarar el tipo de sus variables explícitamente, mientras que un lenguaje que usa **tipado fuerte** necesita declarar el tipo que va a almacenar una variable de antemano.

Javascript utiliza tipado débil.

Javascript es dinámico

Javascript es dinámico

Un lenguaje **dinámico** comprueba los tipos durante la ejecución del programa, mientras que un lenguaje **estático** los comprueba antes de la ejecución.

Javascript es dinámico

Un lenguaje **dinámico** comprueba los tipos durante la ejecución del programa, mientras que un lenguaje **estático** los comprueba antes de la ejecución.

Javascript es un lenguaje dinámico.

C: Compilado, tipado fuerte y estático

C: Compilado, tipado fuerte y estático

```c
#include <stdbool.h>
void main(){

    // Hay que declarar el tipo de la variable.
    int numero = 42;
    bool booleano = true;

    // El programa detecta el error aqui antes de ser
     ejecutado.
    float error = numero + booleano;

    // Esto tampoco estaria permitido.
    booleano = 3.49;
}
```

Este programa ha de ser compilado con un compilador.

El mismo ejemplo, con Javascript

El mismo ejemplo, con Javascript

```
// No se declara el tipo de las variables.
let numero = 42;
let booleano = true;

// Al ser dinamico, esto se permite.
booleano = "Ahora soy un string!";

// E incluso esto...
let caos = numero + booleano - false;
```

Cómo ejecutar Javascript

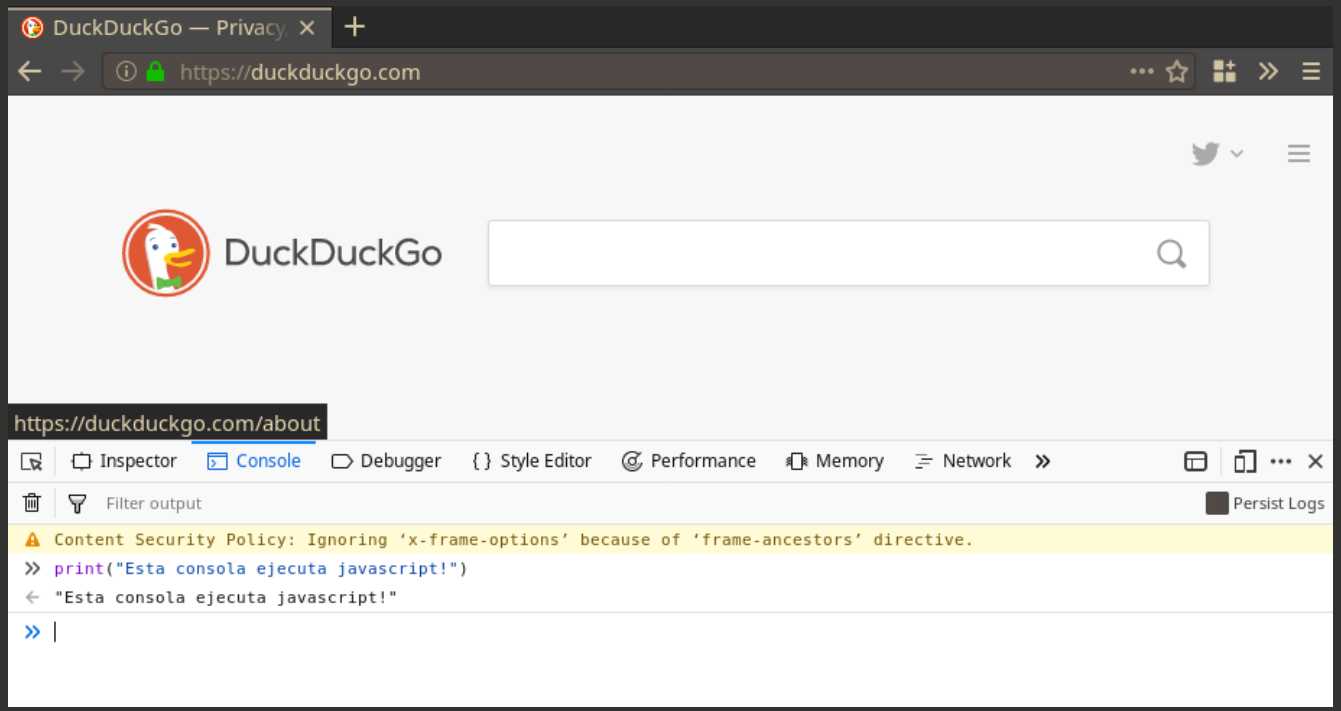
Cómo ejecutar Javascript

Hay varias maneras de ejecutar Javascript. La mas accesible es desde vuestro navegador de preferencia (Firefox, Chrome...)

Cómo ejecutar Javascript

Hay varias maneras de ejecutar Javascript. La mas accesible es desde vuestro navegador de preferencia (Firefox, Chrome...)

Abrid las herramientas de desarrollador (F12), y buscad la opción de consola.



Como ejecutar Javascript

Para este taller, vamos a utilizar JS Bin, un editor online de Javascript.

`https://jsbin.com/?js,console`

ECMAScript

Antes de empezar con Javascript, deberíais saber esto...

ECMAScript es un estándar de lenguajes de programación, y Javascript es una implementación de dicho estándar.

ECMAScript se va actualizando con el tiempo con nuevas funcionalidades, y versiones antiguas de algunos navegadores pueden no estar actualizadas para soportar nuevas versiones de ECMAScript.

ECMAScript

En este taller, todo lo que os enseñe esta basado en ECMAScript 6 (También llamado ECMAScript 2015, o ES6) y esta soportado por todos los navegadores actuales.

En caso de que tengáis que soportar navegadores antiguos (Internet Explorer 11 y menor, por ejemplo), algunas partes de este taller no funcionarán.

Variables y constantes

Hay varias maneras de crear variables en JavaScript:

- `var` para declarar una variable
- `let` para declarar una variable
- `const` para declarar una constante

Var y let

La diferencia entre `var` y `let` es simple.

- `var` define las variables a un nivel funcional (pertenecen a la función en la que se definen o, en caso de que no pertenezcan a ninguna, son globales)
- `let` define las funciones a un nivel de bloque (pertenecen al bloque en el que se definen o, en caso de que no pertenezcan a ninguno, son globales)

Mostraremos ejemplos de esto mas adelante, cuando entendáis la sintaxis de JavaScript. En caso de duda, utilizad `let` para evitar posibles problemas.

Const

se utiliza para definir constantes. Esto implica que en la mayoría de los casos, una variable constante:

- no puede ser reasignada
- no puede ser modificado*

Intentar hacer cualquiera de estas cosas a una constante va a resultar en un error.

Tipos de datos

En Javascript, tenemos los siguientes tipos de datos:

Primitivos:

- Number
- String
- Boolean
- Null
- Undefined
- Symbol

No primitivos:

- Object

Numbers

Un **number** (Numero) es un tipo de dato numérico.

A diferencia de otros lenguajes, no hay distintos tipos dependiendo de si el numero es decimal o no, o de cuanto espacio se quiera reservar (short, int, long, double, float...)

En Javascript, solo hay un único tipo, number. Este es similar a un double en otros lenguajes de programación (64 bits, punto flotante)

Numbers

```
let meaningOfLife = 42;
typeof(meaningOfLife); // "number"
```

Numbers

```
let meaningOfLife = 42;
typeof(meaningOfLife); // "number"

let floatingPoint = 3.141592;
typeof(floatingPoint); // "number"
```

Numbers

```
let meaningOfLife = 42;
typeof(meaningOfLife); // "number"

let floatingPoint = 3.141592;
typeof(floatingPoint); // "number"

// Tambien puedes utilizar notacion cientifica
let exponentialNotation = 123e5; // 123 * (10^5) =
    12300000
```

Numbers

```
let meaningOfLife = 42;
typeof(meaningOfLife); // "number"

let floatingPoint = 3.141592;
typeof(floatingPoint); // "number"

// Tambien puedes utilizar notacion cientifica
let exponentialNotation = 123e5; // 123 * (10^5) =
    12300000

// O hexadecimal (0x) y octal (0)
let hexadecimalNum = 0xFF; // 255
let octalNum = 0147 // 103
```

Numbers: Casos especiales

```
// Y que pasa si hago esto?
let divisionByZero = 1/0;
```

Numbers: Casos especiales

```
// Y que pasa si hago esto?
let divisionByZero = 1/0; // Infinity
```

Numbers: Casos especiales

```
// Y que pasa si hago esto?
let divisionByZero = 1/0; // Infinity
let negativeDivisionByZero = -1/0; // -Infinity
```

Numbers: Casos especiales

```
// Y que pasa si hago esto?
let divisionByZero = 1/0; // Infinity
let negativeDivisionByZero = -1/0; // -Infinity

// Y un numero extremadamente grande?
```

Numbers: Casos especiales

```
// Y que pasa si hago esto?
let divisionByZero = 1/0; // Infinity
let negativeDivisionByZero = -1/0; // -Infinity

// Y un numero extremadamente grande?
let giganticNumber = 1e9999; // Infinity
let giganticNegativeNumber = -1e9999; // -Infinity

// Infinity e -Infinity son de tipo number!
typeof(Infinity); // "number"
typeof(-Infinity); // "number"

// Y esto de aqui?
let divisionByTomato = 4 / "tomato";
```

Numbers: Casos especiales

```
// Y que pasa si hago esto?
let divisionByZero = 1/0; // Infinity
let negativeDivisionByZero = -1/0; // -Infinity

// Y un numero extremadamente grande?
let giganticNumber = 1e9999; // Infinity
let giganticNegativeNumber = -1e9999; // -Infinity

// Infinity e -Infinity son de tipo number!
typeof(Infinity); // "number"
typeof(-Infinity); // "number"

// Y esto de aqui?
let divisionByTomato = 4 / "tomato"; // "NaN"
```

Numbers: Casos especiales

NaN es un número especial. Como habréis notado, Javascript no suele devolver errores ante casos extraños.

Ya entraremos en detalle acerca de como Javascript maneja estos casos, pero por ahora centrémonos en NaN.

Numbers: Casos especiales

NaN es un número especial. Como habréis notado, Javascript no suele devolver errores ante casos extraños.

Ya entraremos en detalle acerca de como Javascript maneja estos casos, pero por ahora centrémonos en NaN.

NaN (Not a Number; No es un número) es un valor numérico que Javascript devuelve siempre que no encuentre un valor legal en una operación aritmética (como una string, con excepciones que ya mencionaremos)

Numbers: Operaciones

```
let num1 = 9, num2 = 3;

// Operador de suma: +
let sumNums = num1 + num2; // 12

// Operador de resta: -
let substractNums = num1 - num2; // 6

// Operador de producto: *
let multiplyNums = num1 * num2; // 27

// Operador de division: /
let divideNums = num1 / num2; // 3

// Operador de resto: %
let reminderOfNums = num1 % num2; // 0
```

Numbers: Operaciones

```
// Operador de incremento: ++
num1++; // num1 = 10;

// Operador de decremento: --
num2--; // num2 = 2;
```

Cuidado! Operaciones con NaN

Cualquier operación que tenga NaN en alguno de sus operandos devolverá NaN.

```
let someOperation = 46 + (38 % NaN) * 27; // NaN
```

Numbers: Punto flotante

Recordad que en Javascript, **todos** los números son de punto flotante. Esto puede dar lugar a imprecisiones, y debéis de tener cuidado con estas.

```
0.1 + 0.2; // 0.30000000000000004

0.1 + 0.2 == 0.3; // false
```

Strings

Un **String** es una cadena de caracteres usada para representar texto.

Puedes usar single quotes (’), double quotes (") o backticks (‘) para delimitar un string, aunque estos últimos son un caso especial que ya mencionaremos.

```
let singleQuotes = ’Soy una cadena de caracteres!’;
let doubleQuotes = "Yo tambien!";
let backTicks = ‘Y yo!‘;

typeof(singleQuotes); // "string"
```

Strings

En caso de que queráis utilizar alguno de los caracteres delimitantes en una string, tenéis dos formas de hacerlo:

- Utilizar un delimitador distinto:

```
let doubleQuoteDelimited = "Uso ' y `!"

let singleQuoteDelimited = 'Y yo uso " y `!'

let backtickDelimited = `Tengo " y ' sin problemas
    !`
```

- Escapar el caracter con \:

```
let stringWithEscapedChar = "Esto \" funciona!"
```

Strings: Concatenación

Las strings solo tienen un operador: concatenación (+)

```
let firstName = "Alberto";
let lastName = "Gonzalez";

let fullName = firstName + " " + lastName;
// fullName = "Alberto Gonzalez"
```

Cuidado! No confundáis los operadores

Tanto el operador de concatenación como el de adición utilizan el mismo símbolo (+), pero son dos operadores completamente distintos. Ya entraremos en detalle acerca de esto mas adelante.

Strings: template strings

En caso de que quieras tener un string con valores de variables, usar template strings puede ser muy útil para facilitar la tarea.

Esto solo funciona si el string usa backticks como delimitador (‘)

```
let firstName = "Alberto";
let lastName = "Gonzalez";

let fullName = `${firstName} ${lastName}`;
// fullName = "Alberto Gonzalez"
```

Boolean

Un **Boolean** (Booleano) es un tipo de dato que sólo tiene dos valores: true (verdadero) o false (falso).

```
let webDevRocks = true;
let javascriptIsGarbage = false;

typeof(webDevRocks); // "boolean"
typeof(javascriptIsGarbage); // "boolean"
```

Boolean: Operadores booleanos

Hay varios operadores booleanos: los operadores de comparación. Estos operadores comparan dos argumentos y siempre devuelven un booleano.

Comparadores normales

- Igualdad (==)
- No igualdad (!=)
- Mayor que (>)
- Menor que (<)
- Mayor o igual que (>=)
- Menor o igual que (<=)

Comparadores estrictos

- Igualdad (===)
- No igualdad (!==)

Boolean: Operadores booleanos

```
4 == 7; // false
2 != 5; // true

3 > 4; // false
3 < 4; // true

2 > 2; // false
2 >= 2; // true
```

Boolean: Operadores booleanos

Javascript hace varias conversiones de tipos por detrás cuando trabajas con tipos distintos.

```
3 > "5" // 3 > 5, false
4 == "4" // 4 == 4, true
```

Para evitar conversiones implícitas, utiliza los comparadores estrictos.

```
4 == "4" // 4 == 4, true
4 === "4" // false

2 != "2" // 2 != 2, false
2 !== "2" // true
```

Siempre que puedas, utiliza el comparador estricto para evitar problemas.

Null

null es un tipo de dato que sólo tiene un valor: null.

null se usa para indicar que algo no tiene ningún valor; que esta vacío.

```
let void = null;

typeof(void); // "object" (por motivos de legacy)
```

Undefined

undefined es un tipo de dato que solo tiene un valor: undefined.

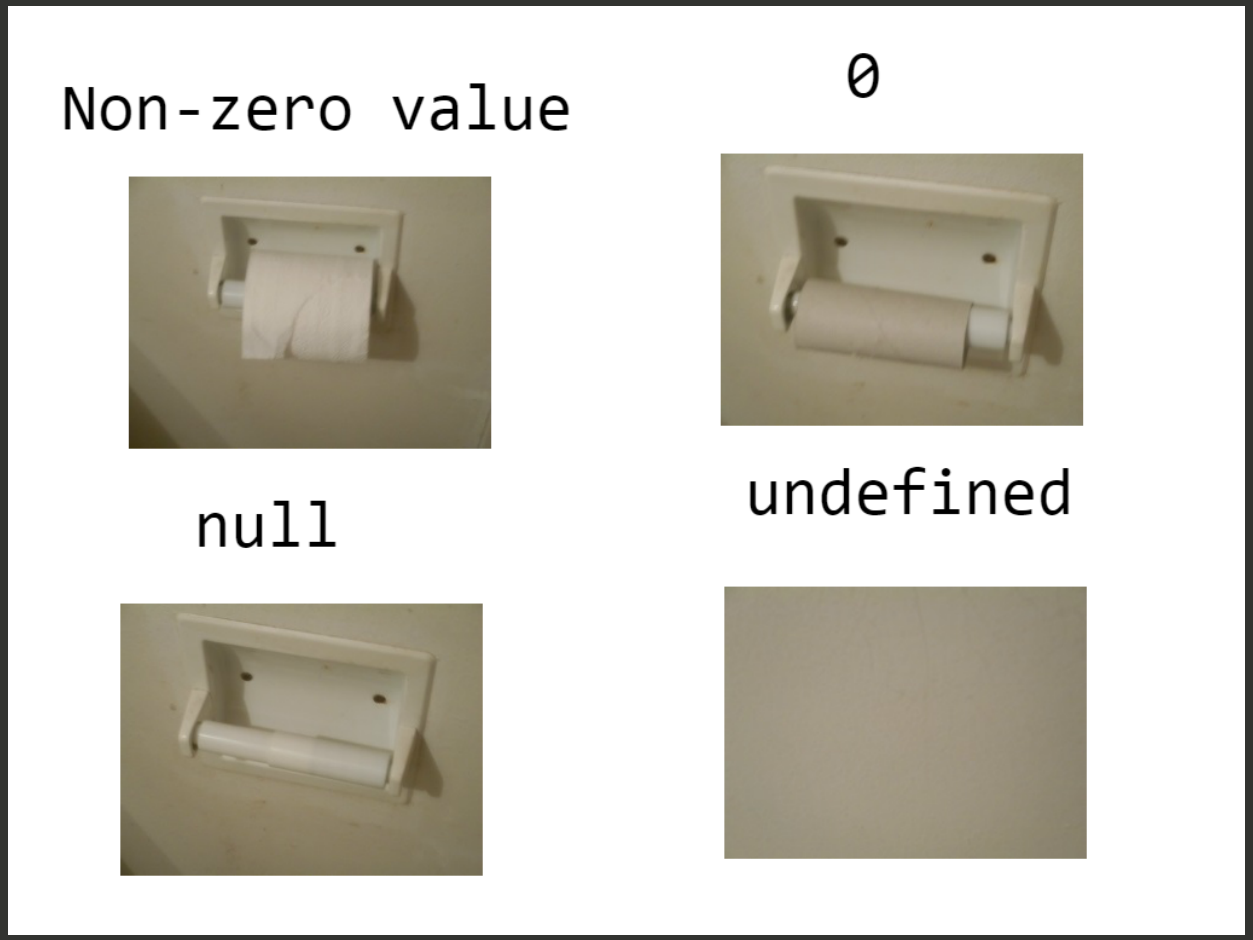
Undefined se usa para indicar que algo no tiene ningún valor asignado.

```
let something;

typeof(something); // "undefined"
```

Diferencia entre null y undefined

null y undefined son dos cosas distintas, pese a parecer idénticas. Aquí tenéis un ejemplo visual.



Diferencia entre null y undefined

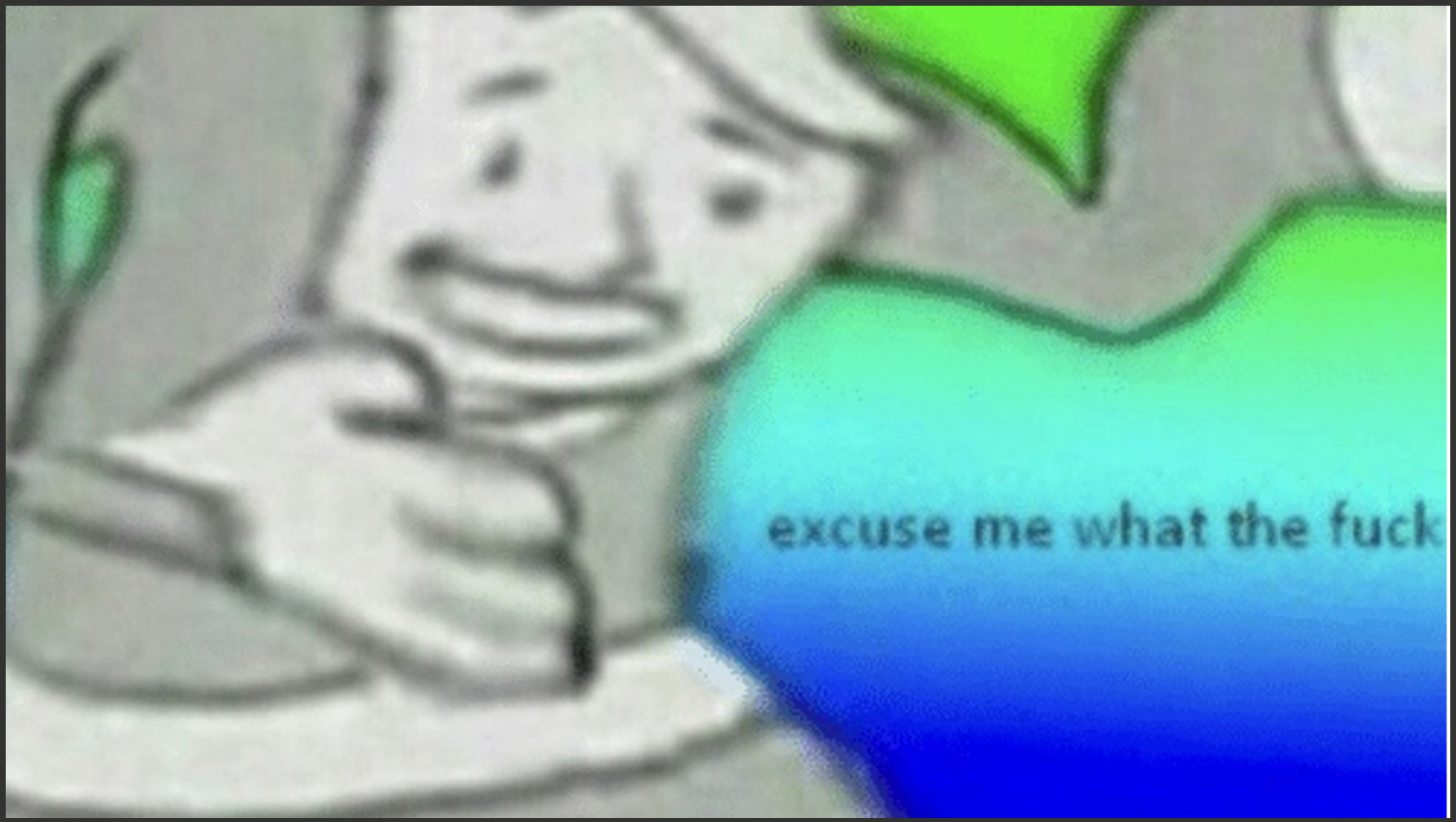
- Undefined indica que algo simplemente no esta ahí, y es el valor que toma cualquier variable por defecto.
- Null indica que el valor de algo es nulo, y has de asignarlo explícitamente a algo.

Conversiones implícitas

Javascript hace conversiones entre tipos por detrás cuando trabajas con tipos distintos.

```
1 + "10" // "110"
1 + 10 // 11
"1" - 10 // "-9"
3 - "Tomato"  // NaN
"b"+"a"+(+"a")+"a" // "baNaNa"
1 / 0 + " and beyond!" // "Infinity and beyond!"
```

Conversiones implícitas



Conversiones implícitas

Si estás sumando algo:

- Si alguno de los operandos es una string, el otro se convierte en string y se concatenan

Si estas restando, multiplicando o dividiendo algo:

- Si alguno de los operandos es una string, el otro se intenta convertir a número (En caso de que no se pueda, se convierte en NaN)

Tipos de datos no primitivos

Sólo hay un tipo de dato no primitivo: El tipo Object (objeto)

Un object es una colección de propiedades. Se definen utilizando llaves: { y }

Cada propiedad tiene una "llave" (identificador) asignada, siguiendo el formato ***llave : propiedad***, y se separan con comas unos pares llave : propiedad de otros.

Objects

```
market_stock = {"apples" : 4, "color" : "red"}

market_stock.apples // 4
market_stock.color // "red"
market_stock.precio // undefined

market_stock.apples-- // market_stock.apples = 3
market_stock.price = 4.95

market_stock // {apples: 3, color: "red", price:
    4.95}
```

Arrays

Un **array** es un tipo de objeto que te permite guardar múltiples valores distintos en una sola variable.

Una array se define con un par de corchetes [], separando los elementos con comas.

```
let someArray = [1, "manzana", 9, true];
typeof(someArray); // "object"
```

Arrays

Puedes acceder a un elemento específico de un array especificando su índice (numeración basada en 0)

```
let someArray = [1, "manzana", 9, true];

someArray[0]; // 1
someArray[1]; // "manzana"
someArray[3]; // true
```

Estructuras de control

Javascript tiene las siguientes estructuras de control.

- if / else if / else
- while / do while
- for
- switch
- for each
- with

If / Else if / Else

La sentencia **if** te permite ejecutar una parte del código sólo si una condición se cumple de antemano.

Una condición es cualquier sentencia que devuelva un valor booleano, o pueda convertirse en uno.

Si no se cumple dicha condición, puedes comprobar por condiciones adicionales usando **else if**, o puedes ejecutar código si no se cumple ninguna condición usando **else**.

If / Else if / Else

```
let someNumber = 5;

if (someNumber < 3) {
    console.log("el numero es menor que 3.");
} else if (someNumber < 10) {
    console.log("el numero esta entre 3 y 9.");
} else {
    console.log("el numero es mayor a 9.");
}
```

While / Do while

La sentencia **while** te permite repetir la ejecución de una parte del código mientras se cumpla una condición.

La sintaxis de un bucle while es la siguiente:

while (condicion) { código }

Una sentencia **do... while** es idéntica a un while, pero esta garantiza que el código se ejecutara al menos una vez, incluso si la condición no se cumple la primera vez.

do { código } while (condicion)

While / Do while

```
let counter = 5;

while(counter < 13){
    console.log(counter++);
}

do {
    console.log("Condicion imposible!");
} while (counter < 0)
```

For

La sentencia **for** te permite repetir la ejecución de una parte del código un numero determinado de veces.

La sintaxis de un bucle for es la siguiente:

for (contador; condición; paso) { código }

El contador es cualquier variable que se use para llevar la cuenta del numero de iteraciones del bucle.

La condición es que se ha de cumplir para que el bucle siga repitiéndose: Cuando la condición deja de cumplirse, el bucle para.

El paso indica de que manera quieres modificar el contador tras cada iteración del bucle.

For

```
for(counter = 10; counter > 0; counter--){
    console.log("Lanzamiento en " + counter);
}
console.log("Despegue!");
```

Switch

La sentencia **switch** te permite evaluar una expresión, y ejecutar un determinado código en base a dicha expresión.

La sintaxis de una sentencia switch es la siguiente:

```
switch (expresion) {
    case (condicion):
        code
        break;
    case (condicion):
        code
        break;
    default:
        code
}
```

Switch

```
let fruit = "apple"
switch (fruit) {
    case banana:
        console.log("Me quedan 3 platanos.")
        break;
    case apple:
        console.log("Me quedan 9 manzanas.")
        break;
    default:
        console.log("No me queda ninguna " + fruta)
}
```

FOR EACH

En JavaScript existen diferentes tipos de bucles **for each**, cada uno de los cuales tiene un comportamiento diferente:

- for...of
- for...in

FOR...OF

La sentencia **for...of** crea un bucle que itera a través de los elementos de objetos iterables (incluyendo Array), ejecutando las sentencias de cada iteración con el valor del elemento que corresponda.

La sintaxis de un bucle **for...of** es la siguiente:

```
for (variable of iterable) {
  code
}
```

FOR...OF

```
let iterable = [10, 20, 30]; //Array de numeros

for (let valor of iterable) {
  valor += 1;
  console.log(valor);
}
// 11
// 21
// 31
```

FOR...IN

La sentencia **for...in** itera sobre todos los nombres de las propiedades de un objeto, en un **orden arbitrario**. Para cada uno de los elementos, se ejecuta la sentencia especificada.

La sintaxis de un bucle **for...in** es:

```
for (variable in objeto) {
    code
}
```

FOR...IN

```
let prices = {"apple" : 1.25, "banana" : 2.30, "
    orange" : 4.5}; // Objeto

for (let fruit in prices) {
    console.log(fruit + " = " + prices[fruit]);
}

// banana = 2.30  <- Orden arbitrario
// apple = 2.30
// orange = 4.5
```

Funciones

Una **función** es una pieza de código que realiza una función determinada. Una función puede tener de 0 a múltiples parámetros de entrada, y puede devolver hasta un parámetro de salida.

```
function (entrada) {
    code;
    return salida;
}
```

Funciones

Hay multiples maneras de definir una función. Una de ellas es usando la palabra clave **function**:

```
function sumNumbers(num1, num2) {
    let total = num1 + num2;
    return total;
}
```

Otra manera es asignando la funcion a una variable o constante (se recomienda que sea una constante):

```
const sumNumbers = function (num1, num2) {
    let total = num1 + num2;
    return total;
}
```

Funciones

La ultima manera de definir una funcion es usando una función flecha:

```
const sumNumbers = (num1, num2) => {
    let total = num1 + num2;
    return total;
}
```

Da igual como definas la funcion, al final todas se emplean de la misma manera:

```
sumNumbers(4, 8); // 12
```

Funciones: Argumentos

Como dije antes, una funcion puede no tener argumentos de entrada ni de salida.

Si una función no tiene ningún argumento de salida, la función devuelve undefined por defecto.

```
function alertMe(){
    alert("ALERTA!");
}

alertMe(); // "undefined"
```

Funciones: Argumentos

Si pasas mas argumentos a una función de los que necesita, esta ignorara los restantes.

```
function multiply(num1, num2){
    return num1 * num2;
}

producto(9, 2, "patata"); // 18
```

Si pasas menos argumentos a una función de los que necesita, estos valen undefined por defecto.

```
function multiply(num1, num2){
    return num1 * num2;
}

producto(9); // 9 * undefined = "NaN"
```

Funciones: Argumentos

En caso de que queráis tratar con un numero variable de argumentos, el objeto `arguments` contiene todos los parametros pasados a una función, así como su orden.

```
function dummy(){
    for (let argument of arguments){
        console.log(`got arg ${argument}`);
    }
}

dummy(3, "bomb", false, [1, "cat"]);
```

Web Dev 101