

# Introducción a Javascript

Un lenguaje para dominarlos a todos



# Alejandro M. Alberto

- Web & Mobile Full Stack Developer
- En mis tiempos libres, intento dar algunas ponencias o contribuir al Open Source
- <https://twitter.com/skurt23>
- <https://linkedin.com/alejandromalberto>
- Keepcoder

# ¿Qué es Javascript?

- Es un lenguaje de programación interpretado
- Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico
- Se ha visto influido por lenguajes como:
  - Java
  - Perl
  - Self
  - Python
  - C
- Creado en 1995
- Utiliza camelCase para nombrar variables, funciones, etc...

A large yellow square containing the letters 'JS' in a bold, dark grey, sans-serif font, representing the JavaScript logo.

# Historia de Javascript

- Creado por Brendan Eich, desarrollador de Netscape, en 1995
- En sus inicios se llamó LiveScript, pero cambió su nombre a Javascript para obtener mayor popularidad porque en esa época el lenguaje por excelencia era Java
- La primera versión llegó con el navegador Netscape 2.0 y, con el lanzamiento de Netscape 3.0, Se lanzó la versión 1.1 de Javascript debido al éxito que había tenido
- Al mismo tiempo, Microsoft lanzó JScript, una copia de JavaScript con diferente nombre para su navegador Internet Explorer 3.0
- En 1997, el organismo ECMA(European Computer Manufactures Association) y el comité TC39 decidieron que la mejor decisión era estandarizar el lenguaje y surgió ECMAScript

# Versiones de Javascript

<b>Versión</b>	<b>Nombre oficial</b>	<b>Descripción</b>
1.0	ECMAScript 1 (1997)	Primera edición
2.0	ECMAScript 2 (1998)	Sólo cambios editoriales
3.0	ECMAScript 3 (1999)	Se incluye try/catch
4.0	ECMAScript 4	No llegó a ser publicada
5.0	ECMAScript 5 (2009)	Se añadió el modo 'strict'
5.1	ECMAScript 5.1 (2011)	Sólo cambios editoriales
6.0	ECMAScript 2015	Se añade let y const
7.0	ECMAScript 2016	Se añade el operador exponencial
8.0	ECMAScript 2017	Async/await
9.0	ECMAScript 2018	Se añade iteración asíncrona

# Javascript para front-end

- Se ejecuta en el navegador
- En sus inicios, el lenguaje estaba destinado a su uso para dotar de dinamismo a las webs escritas en HTML y CSS
- Se puede acceder a las vistas, suscribirse a eventos de las mismas, cambiar estilos, etc...
- Las APIs disponibles permiten acceder a un almacenamiento local o de sesión, localización, notificaciones, etc...
- En los últimos años, se está popularizando el uso de frameworks de Javascript, los cuáles se usan para crear single-page applications.
  - React
  - VueJS
  - Angular(JS, Typescript)

# Javascript para back-end

- Creado por Ryan Dahl en 2009
- Su nombre es NodeJS
- Ryan Dahl estuvo evangelizando la tecnología desde 2009 pero muy poca gente le apoyaba porque no querían tener los mismos errores de JS en un servidor
- Su gestor de paquetes en NPM, creado en 2010 por Isaac Schlueter
- Gracias en parte a NPM y su facilidad de uso fue ganando popularidad
- Hoy en día, un 90% de las herramientas para desarrollo web dependen de NodeJS y de su gestor de paquetes para poder ejecutarse



# Versiones de NodeJS

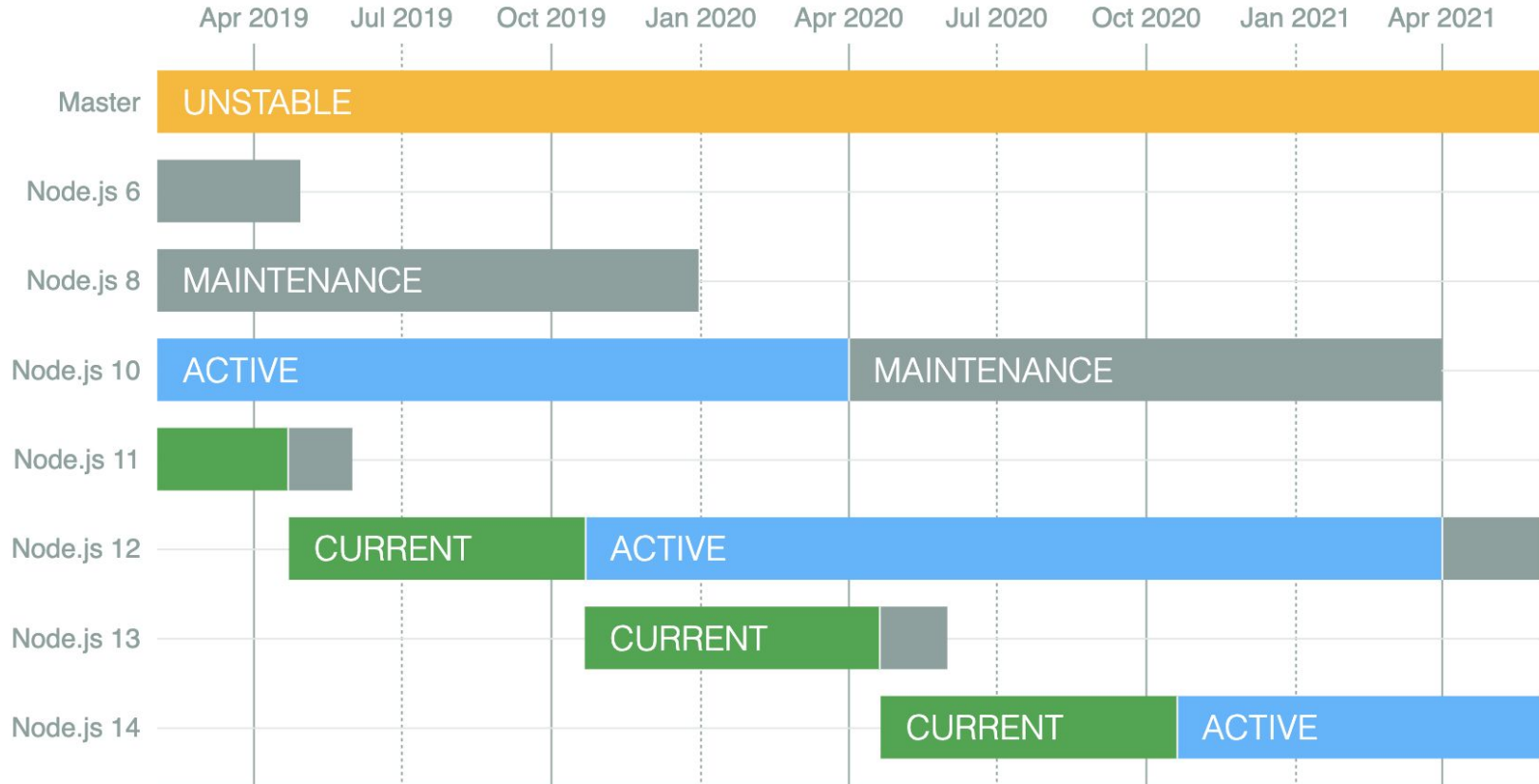
Release	Status	Codename	Initial Release	Active LTS Start	Maintenance LTS Start	End-of-life
v0.10.x	End-of-Life	-	2013-03-11	-	2015-10-01	2016-10-31
v0.12.x	End-of-Life	-	2015-02-06	-	2016-04-01	2016-12-31
4.x	End-of-Life	Argon	2015-09-08	2015-10-01	2017-04-01	2018-04-30
5.x	End-of-Life		2015-10-29			2016-06-30
6.x	End-of-Life	Boron	2016-04-26	2016-10-18	2018-04-30	2019-04-30
7.x	End-of-Life		2016-10-25			2017-06-30
9.x	End-of-Life		2017-10-01			2018-06-30
11.x	End-of-Life		2018-10-23			2019-06-01



# Versiones de NodeJS

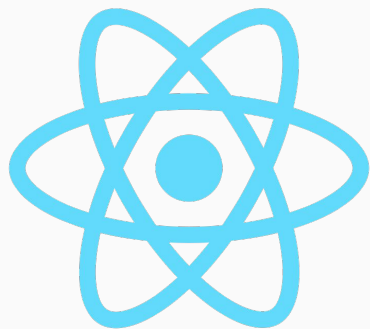
Release	Status	Codename	Initial Release	Active LTS Start	Maintenance Start	End-of-life
8.x	Maintenance LTS	Carbon	2017-05-30	2017-10-31	2019-01-01	December 2019 <sup>1</sup>
10.x	Active LTS	Dubnium	2018-04-24	2018-10-30	April 2020	April 2021
12.x	Current Release		2019-04-23	2019-10-22	April 2021	April 2022
13.x	Pending		2019-10-22			June 2020
14.x	Pending		April 2020	October 2020	April 2022	April 2023

# Versiones de NodeJS



# Javascript para desarrollo móvil nativo

- Nació en 2013, gracias a un hackathon interno de Facebook
- En enero de 2015 presentaron React Native en la React.js Con
- Después de React Native han creado más frameworks de Javascript para desarrollo móvil nativo
  - Nativescript (Vue.js y Angular)
  - Vue Native
- Anteriormente habían aparecido frameworks como Ionic que creaban una app móvil con Javascript pero sin ser nativas. Creaban un WebView, un elemento nativo que permite mostrar páginas webs, y renderizaban la aplicación dentro de este elemento.



# Javascript en el mundo laboral

- Actualmente, Javascript es uno de los lenguajes más usados y demandados en el mundo laboral
- Algunas empresas usan Javascript para todo, así facilitan que todos los desarrolladores puedan tocar cualquier aplicación y no tienen que estar cambiando de lenguaje
- Empresas como Google, Facebook, AirBNB, Github, Amazon, etc... usan JS como parte de su stack



**NETFLIX**

# Instalación de NodeJS

Nos dirigimos al sitio web de [NodeJS](https://nodejs.org) y hacemos click en el botón de descargar la versión LTS (10.16.0)



The screenshot shows the Node.js website's download page for macOS (x64). The navigation bar at the top includes links for INICIO, ACERCA, DESCARGAS, DOCUMENTACIÓN, PARTICIPA, SEGURIDAD, NOTICIAS, and FUNDACIÓN. The main content area states that Node.js is an execution environment for JavaScript built with the Chrome V8 JavaScript engine. Below this, the section is titled 'Descargar para macOS (x64)'. There are two green buttons: '10.16.0 LTS' with the subtext 'Recomendado para la mayoría' and '12.4.0 Actual' with the subtext 'Últimas características'. The '10.16.0 LTS' button is highlighted with a red rectangular border. At the bottom, there are links for 'Otras Descargas', 'Cambios', and 'Documentación del API' for both versions, followed by a link to 'Ó revise la Agenda de LTS.'

nodejs

INICIO | ACERCA | DESCARGAS | DOCUMENTACIÓN | PARTICIPA | SEGURIDAD | NOTICIAS | FUNDACIÓN

Node.js® es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome.

Descargar para macOS (x64)

**10.16.0 LTS**  
Recomendado para la mayoría

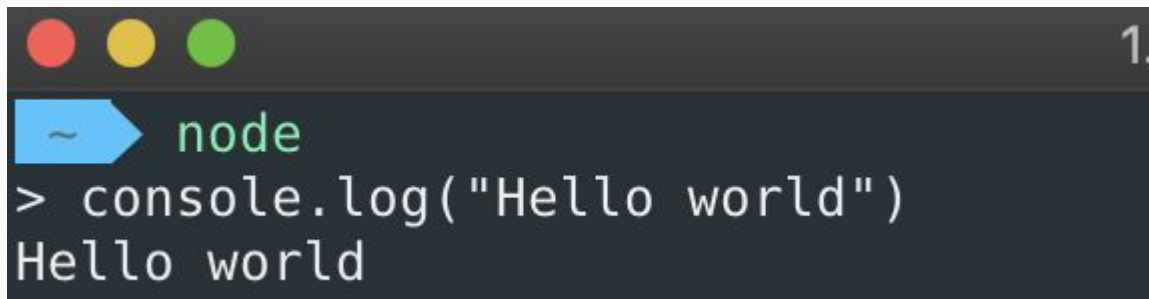
**12.4.0 Actual**  
Últimas características

Otras Descargas | Cambios | Documentación del API    Otras Descargas | Cambios | Documentación del API

Ó revise la Agenda de LTS.

# Hola mundo con NodeJS en terminal

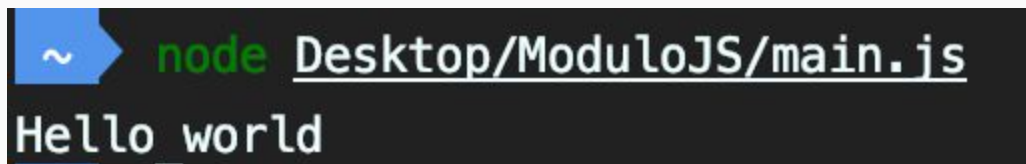
Una vez tenemos NodeJS instalado podemos escribir la palabra node en la terminal y se nos abrirá la consola de NodeJS donde podremos ejecutar código JS y ver su salida

A screenshot of a terminal window with a dark background. At the top, there are three colored window control buttons (red, yellow, green) and a small '1.' in the top right corner. The prompt is a blue arrow pointing right with a tilde '~' inside. The text 'node' is entered in green. Below that, the command '> console.log("Hello world")' is entered in white. The output 'Hello world' is displayed in white on the next line.

```
1.  
~ node  
> console.log("Hello world")  
Hello world
```

# Hola mundo desde un fichero externo

También podemos crear un fichero con extensión .js y lo ejecutaremos con el comando "node main.js". Después de ejecutarlo podremos ver la salida en la terminal y los distintos elementos que queramos imprimir por terminal



```
~ node Desktop/ModuloJS/main.js
Hello world
```

A terminal window with a dark background. A blue prompt character is followed by the command 'node Desktop/ModuloJS/main.js' in green and white text. Below the command, the output 'Hello world' is displayed in white text.

```
JS main.js x
1 console.log("Hello world");
```

A code editor window with a dark background. The title bar shows 'JS main.js' with a close button. The editor contains a single line of code: 'console.log("Hello world");' on line 1. A yellow sun icon is visible at the bottom of the editor.

# Variables

- Es un espacio en el sistema de almacenaje principal del ordenador
- Permite guardar un valor para, posteriormente, recuperarlo y usarlo cuando sea necesario
- Se puede almacenar cualquier tipo de dato primitivo o complejo:
  - Strings
  - Integers
  - Floats
  - Arrays
  - Diccionarios
  - Null
- Se puede crear una variable sin valor para, después de un tiempo almacenar un valor en ella
- Si creamos una variable y no le asignamos ningún valor. Esta variable tendrá como valor "undefined". Un concepto que veremos más adelante en el módulo
- Se declaran con la palabra reservada "let". Más adelante veremos qué son las palabras reservadas y cuáles son
- En versiones anteriores de Nodejs o ECMAScript se usaba la palabra "var" para declarar variables



# Variables

```
> let myVariable = "Hello";  
undefined  
> myVariable = "Hello World!";  
'Hello World!'  
> console.log(typeof myVariable);  
string  
undefined  
> let myUndefinedVariable;  
undefined  
> console.log(myUndefinedVariable);  
undefined  
undefined  
> myUndefinedVariable = 5;  
5  
> console.log(typeof myUndefinedVariable);  
number  
undefined
```

# Constantes

- Su principal diferencia con las variables es que no pueden cambiar su valor
- Puedes usar métodos sobre ella pero no podemos reasignarle un valor a esa variable
- Se crean con la palabras reservada "const" seguido del nombre que queremos asignarle
- En versiones anteriores de NodeJS y ECMAScript, las constantes se definían con la palabra "let"

```
> const myConstant = "Hello World"
undefined
> myConstant = "Hola mundo"
TypeError: Assignment to constant variable.
> console.log(myConstant);
Hello World
undefined
> myConstant.split(" ");
[ 'Hello', 'World' ]
```

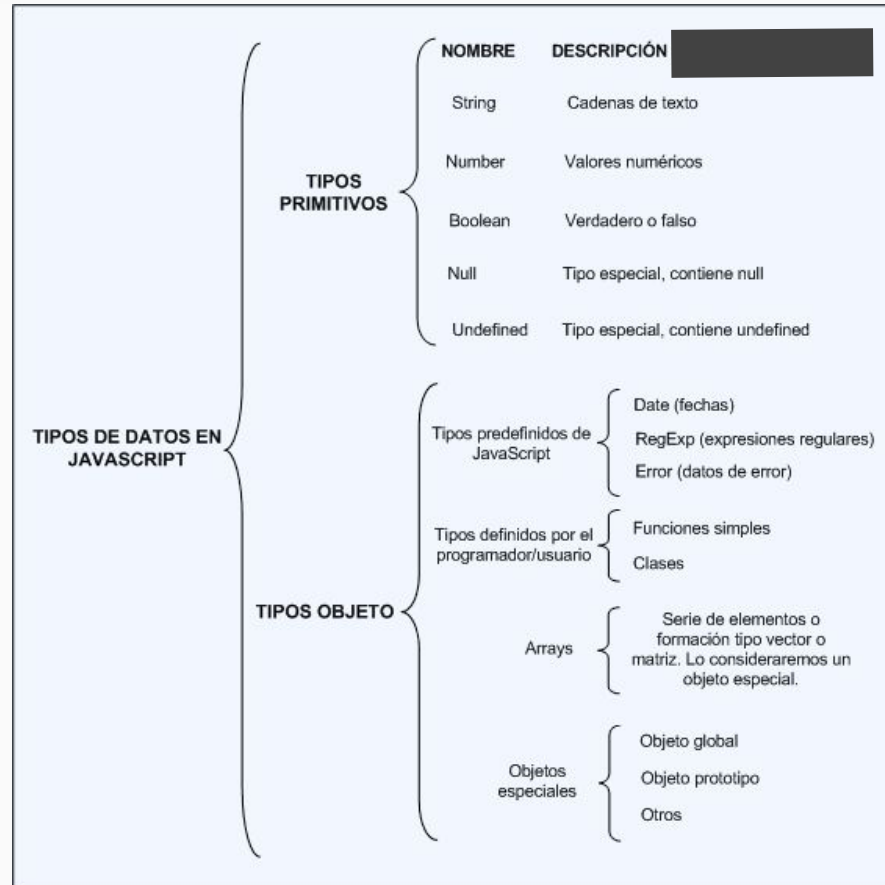
# Palabras reservadas

- Las palabras reservadas son palabras que no deben usarse como nombre de funciones o variables
- En caso de usarse, seguramente se produzcan errores en la ejecución del programa puesto que el lenguaje usa esas palabras para referirse a algún tipo o algún método
- Los valores true, false y null son palabras reservadas
- En este [enlace](#) pueden ver todas las palabras reservadas del lenguaje, como pueden ser:
  - await
  - break
  - case
  - if
  - for
  - in
  - const
  - let
  - while
  - void
  - ...

# Estructuras de datos

- Todos los lenguajes de programación contienen estructuras de datos, pero estas pueden diferir de un lenguaje a otro
- En Javascript salvo los tipos primitivos, todo lo demás son objetos
- En Javascript existen 6 estructuras de datos primitivas:
  - String
  - Number
  - Boolean
  - Symbol (ECMAScript 6)
  - Null
  - Undefined
- Existen otras estructuras de datos como:
  - Objetos
    - Array
    - Dicionarios
    - Tipos predefinidos por el lenguaje
      - Dates
      - RegExp
    - Tipos definidos por el desarrollador
      - Funciones

# Estructuras de datos



# Strings o cadenas de texto

- Es usado para representar datos textuales o cadenas de texto
- Es un conjunto de “elementos” de valores enteros, sin signo, de 16 bits
- Para definir las se pueden usar tres métodos:
  - Mediante el constructor:

```
let myString = new String("Keepcoding");
```

- Mediante la función global:

```
let myString = String("Keepcoding");
```

- Creando una string primitiva, se puede crear usando comillas dobles (“ ”) o comillas simples (‘ ’):

```
let myString = "Keepcoding";
```

# Strings o cadenas de texto

- La longitud de la cadena es el número de caracteres que contiene. Para comprobarlo podemos acceder a la propiedad "length"

```
myString.length // 10
```

- Para determinar la posición de cada carácter se empieza a contar de izquierda a derecha y a partir de la posición cero



- Podemos acceder a ellos mediante el método `charAt()` o llamando a nuestra variable con `[]` y entre los corchetes especificando la posición del carácter al que queremos acceder

```
myString.charAt(1) // "e"  
myString[1] // "e"
```

# Number

- Se usan para representar valores numéricos
- Pueden ser integers (enteros) o floats (números de coma flotante o decimales)
- [valor de doble precisión de 64-bits IEEE 754](#) (un número entre  $-(2^{53}-1)$  y  $2^{53}-1$ )
- De la misma forma que una string podemos crearlo mediante el constructor, la función global o de la forma primitiva
- Cabe destacar que cuando usamos el constructor realmente estamos creando un objeto Number y no un number primitivo. A medida que avancemos en el módulo se explicará esto con más detalle

```
new Number(value);  
let myNumber = new Number('123'); // myNumber === 123 es false  
let myPrimitiveNumber = Number('123'); // myPrimitiveNumber === 123 es true  
let myPrimitiveSecondNumber = 123; // myPrimitiveSecondNumber === 123 es  
true  
myNumber instanceof Number; // es true  
myPrimitiveNumber instanceof Number; // es false
```



# Numbers

- A la hora de crear números con ceros a la izquierda o a la derecha, podemos utilizar la letra “e” seguida del número de ceros que queremos
- En el caso de querer ceros a la derecha bastará con poner la letra “e” seguida de un número positivo
- Al contrario si queremos un número muy pequeño con ceros a la izquierda, el número que especifica la cantidad de 0 será un número negativo

```
let billion = 1000000000;  
let eBillion = 1e9; // 1000000000
```

## Conversión de tipos primitivos

Ahora que ya conocemos dos tipos primitivos, alguno puede que se haya preguntado si es posible pasar de un tipo de dato a otro. Para ello el lenguaje nos proporciona funciones que nos lo permiten.

- `parseInt(string)`
  - Usado para convertir una string en un número entero
- `parseFloat(string)`
  - Usado para convertir una string en un número decimal
- `.toString()`
  - Usado para convertir un número en una string

```
parseInt("2") // 2
parseFloat("3.5") // 3.5
let myNumber = 5
myNumber.toString() // "5"
```

# Boolean

- Una variable o constante booleana almacena dos posibles valores:
  - Verdadero (true)
  - Falso (false)
- Como hemos visto y explicado anteriormente se puede construir con el constructor del objeto:

```
let myBool = new Boolean(false) // myBool == false no es lo mismo
```

- En este caso al ser un objeto, al convertirse en un valor booleano obtendría el valor de true puesto que existe el objeto. Por ello es muy importante recordar que no es lo mismo un tipo objeto que un tipo primitivo en si.

```
let myBool = false; // myBool == false sí, es lo mismo
```

- En este caso, creamos un valor primitivo y su valor en booleano es el que hemos establecido, en este caso, false.

# Boolean

- En caso de que queramos convertir algún otro tipo a booleano, podemos hacerlo de la siguiente forma:

```
x = Boolean(expression);
```

- Los valores booleanos son usados normalmente para tratar con condiciones y ver cuando se producen ciertas situaciones para reaccionar a ellas. Un caso claro puede ser cuando manejamos una contraseña que tiene que tener 8 caracteres como mínimo, una letra y un número, para ello evaluaríamos la condición y cuando el resultado sea true tendremos una contraseña válida
- Es muy importante recordar que cualquier objeto convertido a boolean tendrá de valor **TRUE**, incluido una string cuyo valor sea "false" o un objeto Boolean con valor false

# Null

- Es un literal de javascript que representa un valor nulo o “vacío”
- En javascript, existe otro tipo de dato que es “undefined” que no se debe confundir con NULL.
- Null representa algo que tiene valor nulo o vacío, mientras que “undefined” representa algo que no tiene valor o que no ha sido inicializado.
- Para crearlo es muy sencillo:

```
let foo = null;
```

- Un caso práctico de su uso, puede ser cuando estamos comprobando datos que nos han pasado nos creamos una variable nula y en caso de que alguno de los datos no sea válido modificamos esa variable. Posteriormente, bastaría sólo con comprobar el valor de la variable para saber si hay algún dato erróneo o por el contrario está todo bien.

# Undefined

- Es una variable a la que no se le ha dado valor o no se ha inicializado en ningún momento
- Es un valor primitivo
- En caso de que no haya inicializado una variable, podrás comprobar que vale undefined comparando su tipo con la cadena "undefined". Pero es muy importante diferenciar eso del valor primitivo undefined, por lo que al comparar esa misma variable con undefined esa condición nunca se cumpliría

```
let foo; // typeof foo == "undefined" = true, foo == undefined = false
let bar = undefined // foo == undefined = true
```

- Uno de los errores más comunes en Javascript, tiene un mensaje de error similar a este:
  - "TypeError: Cannot read property "property" of undefined/null"
- Este error viene ocasionado, porque estamos llamando a alguna propiedad o método de algo que en el momento de ejecución tiene como valor null o undefined. La solución sería encontrar porqué al llegar a ese punto el objeto de la propiedad o método que queremos llamar tiene como valor null/undefined

# Arrays

- Es un objeto tipo lista de alto nivel
- Nos permite almacenar un conjunto de datos en la misma variable
- Estos datos no tienen porqué ser del mismo tipo, pueden ser todos de diferente tipo o como se desee
- Son objetos mutables, eso significa que podemos modificar sus valores

```
let myArray = ["Keepcoding", 5, true]
```

- Para acceder a esos elementos que almacenamos dentro de un array podemos usar [], igual que hacíamos anteriormente con los caracteres de una string, y dentro de los corchetes incluyendo el número de posición que ocupa en el array. El primer elemento del array ocupará la posición 0, el segundo la posición 1, etc...

```
let myArray = ["Keepcoding", 5, true]  
myArray[1] // 5
```

# Arrays

- Para modificar cualquier tipo de dato que tenemos en un array, se haría de la misma forma que para modificar una variable pero usando los `[]` para acceder al elemento que se quiere modificar

```
let myArray = ["Keepcoding", 5, true]
myArray[1] = 10
myArray[1] // 10
```

- Podemos guardar un valor en un array en la posición que nosotros queramos, incluso en alguna posición que no exista

```
colors = new Array();
colors[99] = "yellow"; // [undefined, undefined... "yellow"]
```

- Al añadir un elemento en la posición 99 de nuestro array, el array pasará a tener 100 valores y los valores que tiene que rellenar hasta llegar a la posición 99 son valores no definidos, es decir, undefined



# Diccionarios

- En javascript los diccionarios, realmente, son objetos.
- Forman una estructura de datos clave/valor, es decir, podemos almacenar los valores que queramos asignándoles una clave

```
let object = { property1 : valor1,    // propertyn puede ser un id,  
               2:          valor2,    // o un numero,  
               "propertyn": valorn }; // o una cadena
```

- Se puede acceder a esos valores, de la misma forma que un array pero sustituyendo la posición del elemento por su clave. También se pueden modificar sus valores de la misma forma.
- Cuando declaramos una key sin que sea un number o una string, internamente, el lenguaje lo transforma a string

```
let object = { property1 : "valor1",  
               2:          "valor2"};  
object["property1"] // "valor1"  
object["property1"] = "valor3"  
object["property1"] // "valor3"
```

# Date

- Permite trabajar con fechas y horas
- Es un objeto
- Se puede crear sin pasarle argumentos al constructor y creará un objeto Date con la fecha y la hora de hoy según la hora local

```
let myDate = new Date(); // 2019-06-24T18:44:49.789Z
```

- Si pasamos algún argumento, deben ser como mínimo 2
- Soporta UTC y horarios locales

```
let myDate = new Date(2019, 5, 17, 3, 24, 0); // 2019-06-17T03:24:00.000Z  
let mySecondDate = new Date("June 17, 2019 03:24:00");
```

# Operadores

- Javascript tiene varios tipos de operadores:
  - Operadores de asignación
  - Operadores de comparación
  - Operadores aritméticos
  - Operadores bit a bit
  - Operadores lógicos
  - Operadores de cadena de caracteres
  - Operadores condicionales
  - Operadores unarios
  - Operadores relacionales
- A continuación vamos a ver unas tablas donde entraremos en detalle con estos operadores y realizaremos algunos ejercicios prácticos con los más comunes.


# Operadores de asignación

Nombre	Operador abreviado	Significado
Operadores de asignación	<code>x = y</code>	<code>x = y</code>
Asignación de adición	<code>x += y</code>	<code>x = x + y</code>
Asignación de sustracción	<code>x -= y</code>	<code>x = x - y</code>
Asignación de multiplicación	<code>x *= y</code>	<code>x = x * y</code>
Asignación de división	<code>x /= y</code>	<code>x = x / y</code>
Asignación de resto	<code>x %= y</code>	<code>x = x % y</code>
Asignación de exponenciación	<code>x **= y</code>	<code>x = x ** y</code>
Asignación de desplazamiento a la izquierda	<code>x &lt;= y</code>	<code>x = x &lt;&lt; y</code>
Asignación de desplazamiento a la derecha	<code>x &gt;= y</code>	<code>x = x &gt;&gt; y</code>
Asignación de desplazamiento a la derecha sin signo	<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt;&gt; y</code>
Asignación AND binaria	<code>x &amp;= y</code>	<code>x = x &amp; y</code>
Asignación XOR binaria	<code>x ^= y</code>	<code>x = x ^ y</code>
Asignación OR binaria	<code>x  = y</code>	<code>x = x   y</code>

# Operadores de comparación

Operador	Descripción	Ejemplos devolviendo <code>true</code>
Igualdad ( <code>==</code> )	Devuelve <code>true</code> si ambos operandos son iguales.	<pre>3 == var1 "3" == var1 3 == "3"</pre>
Desigualdad ( <code>!=</code> )	Devuelve <code>true</code> si ambos operandos no son iguales.	<pre>var1 != 4 var2 != "3"</pre>
Estrictamente iguales ( <code>===</code> )	Devuelve <code>true</code> si los operandos son igual y tienen el mismo tipo. Mira también <code>Object.is</code> y <code>sameNeess</code> in JS.	<pre>3 === var1</pre>
Estrictamente desiguales ( <code>!==</code> )	Devuelve <code>true</code> si los operandos no son iguales y/o no son del mismo tipo.	<pre>var1 !== "3" 3 !== "3"</pre>
Mayor que ( <code>&gt;</code> )	Devuelve <code>true</code> si el operando de la izquierda es mayor que el operando de la derecha.	<pre>var2 &gt; var1 "12" &gt; 2</pre>
Mayor o igual que ( <code>&gt;=</code> )	Devuelve <code>true</code> si el operando de la izquierda es mayor o igual que el operando de la derecha.	<pre>var2 &gt;= var1 var1 &gt;= 3</pre>
Menor que ( <code>&lt;</code> )	Devuelve <code>true</code> si el operando de la izquierda es menor que el operando de la derecha.	<pre>var1 &lt; var2 "2" &lt; 12</pre>
Menor o igual que ( <code>&lt;=</code> )	Devuelve <code>true</code> si el operando de la izquierda es menor o igual que el operando de la derecha.	<pre>var1 &lt;= var2 var2 &lt;= 5</pre>

# Operadores aritméticos

Operador	Descripción	Ejemplo
Resto (%)	Operador binario correspondiente al módulo de una operación. Devuelve el resto de la división de dos operandos.	<code>12 % 5</code> devuelve <code>2</code> .
Incremento (++)	Operador unario. Incrementa en una unidad al operando. Si es usado antes del operando <code>(++x)</code> devuelve el valor del operando después de añadirle 1 y si se usa después del operando <code>(x++)</code> devuelve el valor de este antes de añadirle 1.	Si <code>x</code> es <code>3</code> , entonces <code>++x</code> establece <code>x</code> a <code>4</code> y devuelve <code>4</code> , mientras que <code>x++</code> devuelve <code>3</code> y, solo después de devolver el valor, establece <code>x</code> a <code>4</code> .
Decremento (--)	Operador unario. Resta una unidad al operando. Dependiendo de la posición con respecto al operando tiene el mismo comportamiento que el operador de incremento.	Si <code>x</code> es <code>3</code> , entonces <code>--x</code> establece <code>x</code> a <code>2</code> y devuelve <code>2</code> , mientras que <code>x--</code> devuelve <code>3</code> y, solo después de devolver el valor, establece <code>x</code> a <code>2</code> .
Negación Unaria (-)	Operación unaria. Intenta convertir a número al operando y devuelve su forma negativa.	<code>-"3"</code> devuelve <code>-3</code> . <code>-true</code> devuelve <code>-1</code> .
Unario positivo (+)	Operación unaria. Intenta convertir a número al operando.	<code>+"3"</code> devuelve <code>3</code> . <code>+true</code> devuelve <code>1</code> .
Exponenciación (**) 	Calcula la potencia de la base al valor del exponente. Es equivalente a <code>base<sup>exponente</sup></code>	<code>2 ** 3</code> devuelve <code>8</code> . <code>10 ** -1</code> devuelve <code>0.1</code> .

# Operadores de bit a bit

Operador	Uso	Descripción
<a href="#">AND bit a bit</a>	<code>a &amp; b</code>	Devuelve uno por cada posición de bit en la cuales los bits correspondientes de ambos operandos tienen valor uno.
<a href="#">OR bit a bit</a>	<code>a   b</code>	Devuelve uno por cada posición de bit en la cual al menos uno de los bits correspondientes de ambos operandos tiene valor uno.
<a href="#">XOR bit a bit</a>	<code>a ^ b</code>	Por cada posición de bit en la cual los bits correspondientes de ambos operandos son iguales devuelve cero y si son diferentes devuelve uno.
<a href="#">NOT bit a bit</a>	<code>~ a</code>	Invierte los bits del operando.
<a href="#">Desplazamiento a la izquierda</a>	<code>a &lt;&lt; b</code>	Desplaza <code>b</code> posiciones a la izquierda la representación binaria de <code>a</code> , el exceso de bits desplazados a la izquierda se descarta, dejando ceros a la derecha de los bits desplazados.
<a href="#">Desplazamiento a la derecha con propagación de signo</a>	<code>a &gt;&gt; b</code>	Desplaza <code>b</code> posiciones a la derecha la representación binaria de <code>a</code> , el exceso de bits desplazados a la derecha se descarta.
<a href="#">Desplazamiento a la derecha con relleno de ceros</a>	<code>a &gt;&gt;&gt; b</code>	Desplaza <code>b</code> posiciones a la derecha la representación binaria de <code>a</code> , el exceso de bits desplazados a la derecha se descarta, dejando ceros a la izquierda de los bits desplazados.

# Operadores lógicos

Operador	Uso	Descripción
AND Lógico ( & & )	<code>expr1</code> <code>&amp;&amp;</code> <code>expr2</code>	Devuelve <code>expr1</code> si puede ser convertido a <code>false</code> de lo contrario devuelve <code>expr2</code> . Por lo tanto, cuando se usa con valores booleanos, <code>&amp;&amp;</code> devuelve <code>true</code> si ambos operandos son <code>true</code> , en caso contrario devuelve <code>false</code> .
OR Lógico (    )	<code>expr1</code> <code>  </code> <code>expr2</code>	Devuelve <code>expr1</code> si puede ser convertido a <code>true</code> de lo contrario devuelve <code>expr2</code> . Por lo tanto, cuando se usa con valores booleanos, <code>  </code> devuelve <code>true</code> si alguno de los operandos es <code>true</code> , o <code>false</code> si ambos son <code>false</code> .
NOT Lógico ( ! )	<code>!expr</code>	Devuelve <code>false</code> si su operando puede ser convertido a <code>true</code> , en caso contrario, devuelve <code>true</code> .



# Operadores de cadena de caracteres

- Son operadores que se pueden utilizar con strings
- Se pueden usar:
  - Operadores de comparación (==, ===, !=,...)
  - Operador de concatenación (+)

```
let myString = "hello" + " ";  
myString += "world"; // "hello world"
```

# Operador unario

- Un operador unario es aquel que sólo necesita un operando
- Existen los siguientes operadores unarios:
  - delete
    - Permite eliminar un objeto, una propiedad de un objeto o un elemento de array
  - typeof
    - Devuelve una cadena de caracteres que contiene el tipo del elemento que evalúe
  - void
    - Especifica una expresión que se evaluará y no devolverá nada

# delete

- Permite eliminar una propiedad de un objeto, un valor de un array o un objeto
- Si finaliza con éxito establece el valor eliminado a undefined
- El operador devuelve true si el objeto ha sido eliminado con éxito o false en caso de que no haya podido ser eliminado

```
delete myObjecto;  
delete myObject.property;  
delete myArray[index];
```

# typeof

- Devuelve el tipo de la variable o constante a evaluar en forma de string

```
let myString = "string";  
let myNumber = 1;  
let today = new Date();  
typeof myString; // "string"  
typeof myNumber; // "number"  
typeof today; // "object"  
typeof myUndefined; // "undefined"  
typeof true; // "boolean"  
typeof null; // "object"
```

# void

- Especifica una expresión que será evaluada y no devolverá nada
- Su uso principalmente se extiende a web

```
void expresion  
void (expresion)
```

# Operadores relacionales

- Un operador relacional compara sus operandos y retorna un valor booleano. Existen dos:
  - in
    - Devuelve true si una propiedad se encuentra en el objeto

```
let myCar = {model: "Fiat Punto", year: 1999};  
"model" in myCar; // true  
let myString = new String("red");  
"length" in myString; // true  
"PI" in Math; // true  
let colors = ["blue", "yellow", "red", "black",  
"white"]:  
0 in colors; // true
```

- instanceof
  - Devuelve true si un objeto es de un tipo específico

```
let myArray = ["Keepcoding", 4, 6, 7, "Javascript"]  
myArray instanceof Array // true  
myArray instanceof String // false
```

## If....else....else if

- if se utiliza para ejecutar un código si la condición es verdadera
- else, por el contrario, es usado para ejecutar un código si la expresión es falsa
- No necesariamente hay que usar siempre if con else, if puede usarse sólo sin necesidad de evaluar cuando la condición es false

```
if (condition) {  
    console.log(condition);  
}  
  
if (condition) {  
    console.log(condition);  
} else {  
    console.log(condition);  
}
```

## If....else....else if

- Else if se puede usar para evaluar otra condición sin necesidad de estar declarando muchos if por el código

```
if (condition) {  
    console.log("condición 1");  
} else if (condition2) {  
    console.log("condición 2");  
} else if (conditionN) {  
    console.log("condición 3");  
} else {  
    console.log("ninguna de las 3 anteriores");  
}
```



## Operadores condicionales (ternarios)

- Es el único operador en Javascript que necesita tres operandos
- El funcionamiento es el mismo que un if...else, se evalúa una condición y si esta es true se retorna un valor o, si por el contrario es false, se retorna otro valor.

```
condition ? true : false
```

- Es muy habitual usar este operador para asignar un valor a una variable o a una constante en base a una condición

```
let age = 25;  
let adult = (age >= 18) ? "adult" : "child"; // "adult"
```

# Switch

- Permite evaluar una expresión e intentar asociar su valor a diferentes etiquetas de caso(case)
- Si encuentra alguna coincidencia ejecuta la sentencia correspondiente a ese case
- En caso de que no encuentre coincidencia, se declara una sentencia "default" que se ejecutará en este caso

```
let myCategory = "Ciencia Ficción";
switch (myCategory) {
  case "Ciencia Ficción":
    console.log("Ciencia Ficción")
    // break / continue;
  case "Comedia":
    console.log("Comedia")
    // break / continue;
  default:
    console.log("default")
}
```

## Break / Continue

- La sentencia break vista en el ejemplo anterior frena la ejecución de la sentencia
- Esto permite que una vez que se evalúa un case no se sigan ejecutando los demás
- Por otro lado, continue, no termina la ejecución:
  - En un switch, vuelve a la condición para evaluarla nuevamente.
  - En un for, como veremos próximamente, salta a la expresión actualizada

# Try/catch

- La sentencia try/catch es utilizada cuando alguna acción en nuestro código puede provocar una excepción
- Las excepciones son imprevistos que ocurren durante la ejecución de un programa, acciones que impiden que el programa siga con su ejecución normal
- En la sentencia try incluimos el código que puede lanzar una excepción
- Con la sentencia catch recibimos esa excepción y podemos reaccionar en base a ella

```
try {  
    monthName = getMonthName(myMonth);  
}  
catch (e) {  
    monthName = "unknown";  
    console.log(e);  
}
```

# Throw

- La expresión throw nos permite lanzar una excepción
- Como explicamos anteriormente, las excepciones son comportamientos inesperados en nuestra ejecución, por ello throw es utilizado comprobamos que algún valor no es el indicado y queremos reaccionar mostrando un error al usuario o cualquier otra acción para evitar que el programa deje de funcionar sin razón
- Puedes lanzar cualquier expresión:
  - String
  - Boolean
  - Number
  - Object
  - etc...

```
throw "Error";  
throw 42;  
throw true;
```

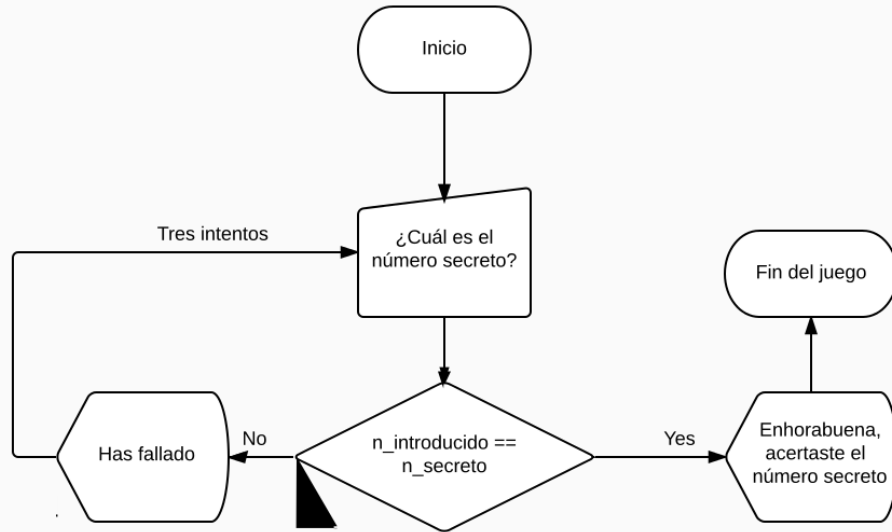
# Finally

- El bloque finally se utiliza al final de la ejecución de try/catch
- En caso de que ocurra una excepción o no, el bloque finally permite realizar una acción independientemente de lo que pase
- En caso de que finally devuelva algún valor, siempre la sentencia try/catch/finally devolverá este valor, ignorando cualquier valor retornado en la sentencia try o la sentencia catch

```
openMyFile();  
try {  
    writeMyFile(theData);  
} catch(e) {  
    handleError(e);  
} finally {  
    closeMyFile();  
}
```

# Bucles

- Un bucle es una sentencia que ejecuta repetidas veces un fragmento de código
- Un ejemplo puede ser el parchís, cuando un jugador tira el dado mueve su ficha el número de veces que haya obtenido al tirar el dado. La acción de mover la ficha puede interpretarse como un bucle en el que en cada repetición del fragmento de código, la ficha avanza una casilla hasta alcanzar el número obtenido en el dado



# For

- El bucle for en javascript es similar al de Java
- Repite la expresión deseada hasta que la condición se evalúa como false

```
for (expresionInicial; condicion; expresionIncremento)
{
    // code
}
```

- La expresión inicial puede ser la declaración de una variable o el uso de una expresión con cualquier grado de complejidad
- Mientras la condición sea true el bucle seguirá ejecutándose
- Después de evaluar la condición y que su valor siga siendo true, se ejecuta el código del bucle
- Después de la ejecución del código, se ejecuta la expresión de incremento y el bucle regresa a evaluar la condición



## Ejemplo

```
let myArray = ["1", "2", "3", "4"]
for (let i = 0; i < myArray.length; i++) {
    myArray[i] = parseInt(myArray[i])
}
myArray // [1, 2, 3, 4]
typeof myArray[0] // "number"
```

- En el ejemplo anterior, nuestro bucle se ejecutará 4 veces. Esto es debido a que después de cada iteración incrementamos el valor de i en 1 y después de 4 ejecuciones su valor será 4 por lo que la condición evaluará a false
- Ayudándonos del valor de i podemos acceder uno a uno por los elementos del array y realizar la acción que deseemos con ese valor

## forEach

- Es un método de los Arrays y no una sentencia de control de flujo como el for visto anteriormente
- Su funcionamiento es exactamente igual, repite un fragmento de código tantas veces como elementos contenga el array
- Itera por cada elemento del array y te lo proporciona para que puedas realizar lo que quieras con él. También nos permite acceder al index del elemento o número de iteración del bucle

```
let myArray = ["1", "2", "3", "4"]  
myArray.forEach( // fragmento de código a repetir 4 veces )
```

## For...of

- Esta sentencia crea un bucle sobre objetos iterables(Array, objetos, Map, Set)

```
let myObject = ["2", "5"]
for (let value of myObject) {
  console.log(value) // "2" en la primera iteración y "5" en la segunda
}
```

- Funciona exactamente igual que el forEach, con la diferencia de que for...of sí es una sentencia y no un método como el anterior.
- Podemos acceder al index usando el método entries() propio de los Arrays

## For...in

- Es otro bucle sobre objetos iterables pero en esta ocasión cuenta con una pequeña diferencia
- Al contrario que el bucle for...of o forEach, for...in itera sobre las propiedades de un objeto o sobre los índices de un array
- En cada iteración podremos acceder uno a uno a las distintas propiedades que tenga un objeto o a los índices de un array

```
let myArray = ["2", "5"]
for (let value in myArray) {
  console.log(value) // 0 en la primera iteración y 1 en la segunda
}
```

```
let myObject = {
  name: "Keepcoding",
  module: "Javascript"
}
for (let value in myObject) {
  console.log(value) // "name" en la 1º iteración y "module" en la 2º
}
```

# Do/While

- La sentencia do/while ejecuta un fragmento de código hasta que la condición sea evaluada a false
- En este caso el fragmento de código a ejecutar, se encuentra en la sentencia do mientras que la condición se especifica con la sentencia while

```
do {  
    // code  
}  
while (condition);
```

- Después de tantos bucles, toca responder a dos preguntas.
  - ¿Es posible crear un bucle infinito?
  - Si es así, ¿qué pasaría si ejecutamos un bucle infinito en nuestro código?

# While

- Crea un bucle que ejecuta una sentencia mientras cierta condición sea true

```
while (condition) {  
    console.log("Keepcoding")  
}
```

# Funciones

- Es uno de los pilares de la programación en Javascript
- ¿Recuerdan cuando he hablado de métodos? Pues los métodos son funciones pertenecientes a algún objeto
- Es un conjunto de sentencias que se ejecutan para realizar una tarea o calcular un valor
- Esto permite poder agrupar nuestro código en pequeñas funciones para repetir el menor código posible y re-utilizarlo tantas veces como queramos
- La declaración de una función consta de tres partes:
  - Nombre
    - Es el nombre con el que nos referiremos a nuestra función
  - Argumentos
    - Son distintos elementos que recibe la función para poder ejecutarse y realizar su tarea
  - Sentencias
    - Es el código que se ejecutará al ejecutar esa función
- Las funciones pueden devolver cualquier tipo de valor o pueden no devolver nada
- Cabe destacar que el conjunto de sentencias de nuestra función podremos acceder a variables y funciones que se encuentra fuera de nuestra función pero todo lo que declaremos dentro de la función SÓLO será accesible desde dentro del conjunto de sentencias de esa función

# Funciones

```
function myFunction(argument1, argument2, argument3, ...) {  
    if (argument1 > argument 2) {  
        return argument3  
    }  
}
```

```
function myFunction() {  
    if (argument1 > argument 2) {  
        console.log(argument3)  
    }  
}
```



# Funciones

- Podemos crear funciones de forma simplificada usando funciones anónimas

```
let multiply = function(number) {return number * number};
```

- Cuando no sabemos cuántos argumentos de entrada vamos a recibir en la función, podemos usar arguments
- arguments es un array que contiene los argumentos que hayan pasado a la función y que no estén declarados en la misma

```
function myConcat(separator) {  
  let result = "";  
  for (let i = 1; i < arguments.length; i++) {  
    result += arguments[i] + separator;  
  }  
  return result;  
}
```

# Funciones

- Para ver cómo funciona la función anterior necesitamos saber como ejecutar las funciones

```
myConcat( " , " , "red" , "orange" , "blue" );
```

- Al ejecutar la función veremos cómo, a pesar de no haber declarado todos los argumentos, hemos podido iterar a través de arguments y obtenerlos todos
- En el caso de que la función a ejecutar devuelva algún valor que queramos almacenar, bastaría con igualar la ejecución de la función a una nueva variable o a una ya existente
- Es muy importante tener en cuenta que para ejecutar una función es obligatorio abrir los paréntesis porque, de no hacerlo, estaremos referenciando la función pero no ejecutándola

# Arrow functions

- Son funciones anónimas que nos permiten declarar funciones de forma más sencilla y con menos código
- Podemos definir un conjunto de sentencias o, en el caso de que sólo se necesite una sentencia, podemos devolver directamente esa sentencia como veremos a continuación

```
let myFunction = (argument) => {  
  console.log(argument)  
}
```

```
let myFunction = (number) => number + number  
myFunction(2) // 4
```

# Asincronía

- La definición oficial explica que es una acción que no tiene lugar en total correspondencia temporal con otra acción
- En programación podemos decir que es una acción que se ejecuta en un hilo aparte mientras sigue la ejecución de nuestro código
- La razón de ejecutar código en otro hilo y hacerlo asíncrono es porque pueden ser tareas pesadas que pueden bloquear la ejecución de nuestro código o porque ejecutamos alguna sentencia que nos va a responder pero no podemos saber cuando
- Los ejemplos más habituales son peticiones http, escritura y lectura de ficheros, etc...

# Funciones asíncronas con callbacks

- Es una función que se pasa a otra como argumento para, posteriormente, ejecutarla en el interior de esta última para completar algún tipo de acción

```
function sayHi(name) {  
  alert('Hola ' + name);  
}  
  
function myCallbackFunction(callback) {  
  let name = prompt('Por favor ingresa tu nombre.');
```

```
  callback(name);  
}  
  
myCallbackFunction(sayHi);
```

# Hoisting

- El hoisting fue pensado como la forma de referirse a cómo funcionan los contextos en Javascript
- En muchos sitios podrán ver que el hoisting consiste en que la declaración de funciones y de variables sean movidas físicamente al principio del código
- Esto no es realmente así, lo que sucede es que esas declaraciones son asignadas en memoria durante el tiempo de compilación pero no son movidas físicamente
- Una de las ventajas de esto es que podemos ejecutar una función antes de declararla, pues que esta se almacenará en memoria en tiempo de compilación
- El hoisting sólo es válido es declaraciones no en iniciaciones, por ello, si ejecutamos una sentencia que llama a una variable que se inicializa posteriormente a esa sentencia, la variable valdrá undefined

```
let a = 5;  
let b;  
console.log(x + y); // "5 undefined"  
y = 10;
```

# Bindings

- En javascript, la palabras reservada `this` sirve para hacer referencia a variables o funciones dentro de un contexto
- El problema es que a veces necesitamos usar otro contexto al ejecutar una función para su correcto funcionamiento
- Para ello existen los bindings
- Crear un binding es ejecutar una función llamada `bind` sobre la función a la que queremos aplicar el contexto de `this` adecuado

# Bindings

```
this.x = 9;  
let module = {  
  x: 81,  
  getX: function() { return this.x; }  
};  
  
module.getX(); // 81  
  
var getX = module.getX;  
getX(); // 9, porque en este caso, "this" apunta al objeto global  
  
// Crear una nueva función con 'this' asociado al objeto original 'module'  
let boundGetX = getX.bind(module);  
boundGetX(); // 81
```



# Programación orientada a objetos

- Su uso se popularizó a principios de la década de los 90
- Es un paradigma de programación que vino a innovar la forma de obtener resultados. Cada objeto ofrece una función en especial en la que modifican unos datos de entrada para dar unos datos de salida específicos
- Utiliza la abstracción para crear objetos del mundo real
- En la actualidad, la mayoría de lenguajes permiten la programación orientada a objetos
- Cada objeto tiene su identidad, es decir, tiene alguna función o característica que lo diferencia del resto de objetos
- Una de sus desventajas es que a veces resulta complicado cambiar la forma de pensar para identificar exactamente cómo se debería comportar un objeto, a su vez, un programador puede no comprender el objeto de la misma forma que otro porque identifica algo que no tiene presente la otra persona o porque piensa que algo sobra
- Una de sus ventajas es la legibilidad del código y la posibilidad de reutilizar código y no replicar por toda la aplicación constantemente

# Programación orientada a objetos en Javascript

- Antes de EcmaScript 2015, el lenguaje no tenía una declaración de clase como en otros lenguajes como puede ser Java
- En su lugar, Javascript utiliza funciones como clases
- Para crear nuestro objeto bastaría con crearnos una función con el nombre del objeto
- Esta función haría la labor de constructor, por lo que la lógica de la inicialización de nuestro objeto debe estar dentro de él, así como la asignación de propiedades
- Usamos this para la asignación de propiedades para hacer referencia al contexto de cada objeto que creamos y que no haya interferencias entre los objetos y sus propiedades

```
function Persona(primerNombre) {  
    this.primerNombre = primerNombre;  
}  
  
let person = new Persona('Pedro');  
person.primerNombre // "Pedro"
```

# Programación orientada a objetos en Javascript

- A partir de EcmaScript 2015, podemos definir las clases con la palabra class
- La asignación de propiedades se haría en el interior de la sentencia de la clase, así como los métodos que pertenezcan al objeto
- Todas las clases tienen una función constructor para la inicialización de la clase y la asignación de las propiedades

```
class Polygon {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
}  
  
let myPolygon = new Polygon(5, 10);  
myPolygon.x // 5
```

# Métodos de clase

- Como hemos dicho anteriormente, la programación orientada a objetos se basa en simular el comportamiento de los objetos en la vida real
- Todos los objetos pueden realizar algunas acciones que pueden ser representadas mediante funciones que ejecutan un conjunto de sentencias para obtener el resultado de esa acción
- En el caso de querer crear un objeto Perro, este tendría el método ladrar puesto que esta es una acción que realiza los perros en la vida real
- Antes de EcmaScript 2015, se utiliza prototype para poder crear métodos de un objeto
- prototype representa al objeto prototipo de Object

```
Person.prototype.walk = function() {  
    alert("Estoy caminando!");  
};  
Person.prototype.sayHi = function(){  
    alert("Hola, Soy" + this.name);  
};
```

# Métodos de clase

- A partir de EcmaScript 2015, para crear métodos de clase bastaría con añadir una función con el nombre del método dentro de la sentencia de la clase

```
class Dog {  
  constructor(name) {  
    this.name = name  
  }  
  speak() {  
  
    console.log(this.name +  
      ' ladra. ');  
  }  
}
```

# Herencia

- La herencia en los objetos nos permite heredar las propiedades y los métodos de otra clase y extenderla o sobrescribir los métodos del objeto padre en nuestro objeto
- Una gran ventaja es la posibilidad de reutilizar código común entre clases
- Por ejemplo, estamos creando una app sobre animales. Tenemos que crear 10 animales y para ello, podríamos crear una clase Animal que tenga las propiedades y métodos comunes entre todos, posteriormente podemos crear una clase según sea un mamífero, anfibio, etc... con sus propiedades y métodos y, por último, crear una clase para cada animal con las propiedades específicas de cada una y sus métodos
- A partir de EcmaScript 2015, bastaría con usar la palabra `extends` para heredar de una clase

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  speak() {  
    console.log(this.name + ' hace un  
ruido.');  }  
}  
  
class Dog extends Animal {  
  speak() {  
    console.log(this.name + ' ladra.');  }  
}
```

# Herencia

- La herencia antes de EcmaScript 2015 se realiza llamando al constructor de la clase padre dentro del constructor de la clase hija y, posteriormente, inicializar las propiedades específicas de nuestro objeto

```
function Person(name) {  
    this.name = name;  
}  
Person.prototype.walk = function() {  
    alert("Estoy caminando!");  
};  
Person.prototype.sayHi = function(){  
    alert("Hola, Soy" + this.name);  
};  
function Student(name, module) {  
    Person.call(this, name);  
  
    this.module = module;  
};
```



# Importación

- La sentencia import sirve para poder importar un objeto, una variable o una función en otro fichero js y poder utilizarlo
- Esto nos permite poder estructurar nuestro código de una forma más eficiente, pudiendo además evitar duplicar código
- Antiguamente para importar algo se usa la función require():

```
const fs = require('fs');
```

- De esta forma, importamos un módulo entero y todo lo que éste exporte
- Pero, ¿para qué importar un módulo entero si podemos importar sólo lo que necesitamos?
- Para esto tenemos la nueva sintaxis de imports

## Importación del contenido de todo un módulo

```
import * as foo from '/modules/my-module.js';
```

- De esta forma se importaría todo el contenido del módulo como anteriormente con `require()`
- Al usar `*` especificamos que queremos todo lo que ese módulo exporte
- La palabra reservada `as` nos permite acceder a todo lo que hemos importado con el nombre que le especifiquemos

## Importación de uno o varios miembros de un módulo

```
import {foo} from '/modules/my-module.js';  
import {foo, bar} from "my-module.js";  
import {reallyUglyModuleExportName as bestName}  
  from '/modules/my-module.js';  
import {  
  reallyUglyModuleExportName as bestName,  
  anotherUglyModuleName as best  
} from '/modules/my-module.js';
```

- Haciendo uso de los corchetes podemos importar los miembros que queramos del módulo
- Como anteriormente, con `as` podemos “renombrar” esos módulos para acceder a ellos de una forma más fácil

## Importar un módulo para efectos secundarios

```
import './modules/my-module.js';
```

- De esta forma importamos el módulo para efectos secundarios, es decir, esta forma de import ejecuta el código del módulo pero no importa ningún valor para re-utilizarlo

# Importación de elementos por defecto

```
import myDefault from '/modules/my-module.js';
```

- De esta forma importamos el objeto, función o variable que el módulo exporte por defecto
- También es posible usar la importación de elementos por defecto junto con las importación de todo un módulo o de sus miembros, siempre y cuando realicemos la importación por defecto en primer lugar

```
import myDefault, * as myModule from '/modules/my-module.js';  
import myDefault, {foo, bar} from '/modules/my-module.js';
```

# Exportación

- La sentencia export se usa al crear módulos en Javascript para exportar funciones, objetos, etc...
- Existen dos tipos de exports
  - Exports nombrados
  - Exports por defecto

```
export const foo = Math.sqrt(2);  
export default function() {}
```

# Métodos más usados en Strings

- `split(separador)`
  - Divide una string en una array separando la cadena por el separador indicado como parámetro del método
- `trim()`
  - Elimina los espacios en blanco en ambos extremos de un string
- `join(uniión)`
  - Une todos los objetos de una lista en una cadena. Esta unión la realiza con el parámetro especificado en la función
- `concat(string1, string2, string3, ...)`
  - Encadena dos o varias strings en una sola string
- `includes(search)`
  - Indica si una cadena de texto se encuentra dentro de otra cadena de texto
- `replace(replaceString)`
  - Busca unos caracteres en una cadena y los sustituyes por los especificados

# Métodos más usados en Numbers

- `toFixed(valor)`
  - Formatea un número de forma que lo devuelve con el número de decimales que hayamos especificado como parámetro de entrada de la función
- `isNaN()`
  - Especifica si el número es NaN
- `isFinite()`
  - Determina si el valor es finito



# Métodos más usados en Arrays

- `filter(function)`
  - Este método nos permite filtrar en un array, para ello recibe como parámetro de entrada una función que recibe un objeto de nuestro array cada vez que se ejecuta y comprueba una condición para según devuelva true o false incluir el elemento en nuestro array filtrado
- `splice()`
  - Cambia el contenido de un array eliminando un elemento, añadiéndolo o modificándolo
- `slice()`
  - Devuelve una copia de nuestro array, pudiendo especificar desde qué posición queremos empezar a realizar la copia y donde queremos detenerla
- `reverse()`
  - Devuelve el array del revés
- `sort(function)`
  - Como en el caso de filter, sort recibe una función que, en este caso, recibe dos parámetros de entrada que son los elementos que compara cada vez que se ejecuta esa función. Esta función debe devolver una condición que si devuelve true lo colocará antes o después en el array

# Métodos más usados en Arrays

- `reduce(function)`
  - Aplica una función a cada valor de un array para reducir el array a un único valor. Recibe 4 parámetros de entrada:
    - `valorAnterior`
    - `valorActual`
    - `index`
    - `array`
- `map(function)`
  - Devuelve un nuevo array con los valores resultantes de la ejecución de la función que recibe un elemento del array en cada iteración.

# Métodos más usados en Dates

- `now()`
  - Devuelve el número de milisegundos transcurridos desde las 00:00:00 UTC del 1 de enero de 1970
- `parse()`
  - Transforma una cadena con una fecha al número de milisegundos transcurridos desde 1970
- `getDate()`
  - Devuelve el día del mes de la fecha especificada
- `getFullYear()`
  - Devuelve el año de la fecha en cuestión
- `getDay()`
  - Devuelve el día de la semana en el que se encuentra el día de la fecha especificada
- `getUTCMonth()`
  - Devuelve el mes de la fecha especificada. Este método devuelve un número entero entre 0 y 11, siendo 0 el mes de enero y 11 el mes de diciembre

# Extras

- [NVM](#)
- [Awesome NodeJS](#)
- [Typescript](#)

¡Muchas gracias por su  
atención!