JavaScript

Ángel Villalba Fdez-Paniagua

Table of Contents

1. Introducción

JavaScript fue creado en 10 días en el año 1995 por *Brendan Eich*, que trabajaba en Netscape. Al principio, no se llamaba JavaScript, sino que su nombre era **Mocha**, un nombre que le eligió *Marc Andreessen* (fundador de Netscape). En septiembre de ese mismo año, se le cambia el nombre por primera vez a **LiveScript**. Un poco más tarde en ese mismo año, Nescape y Sun Microsystems firman una alianza para desarrollar el nuevo lenguaje de programación, ahora con el nombre de JavaScript, el cual se lo dan como una estrategia de marketing, ya que en aquella época **Java** estaba de moda en el mundo informático y añadiendole Java al nombre, lo que intentaban era atraer a programadores de Java para que probaran este lenguaje.

La primera versión de JavaScript es un exito y *Microsoft* decide sacar **JScript** con su navegador Internet Explorer 3. JScript era una copia de JavaScript a la que le habían cambiado el nombre para evitar problemas legales. Y viendo esto Netscape decidió estandarizar el lenguaje JavaScript.

En los años 1996 y 1997 JavaScript es llevado a **ECMA** para labrarse una especificación estándar, que otros proveedores de navegadores prodrían implementar basandose en el trabajo realizado en Netscape. Este trabajo lleva a la liberación oficial de **ECMAScript** que es el nombre de la norma oficial, siendo JavaScript la implementación más conocida.

El proceso de las normas continuó con los lanzamientos de ECMAScript 2 (en 1998) que traía unas pequeñas modificaciones para adaptar el lenguaje a una ISO (Organización Internacional para la Estandarización) y ECMAScript 3 (en 1999) que es la línea base para el JavaScript moderno.

En 2005, las comunidades se pusieron a trabajar para revolucionar lo que podría hacerse con JavaScript. Este esfuerzo de las comunidades se desató cuando en 2005, *Jesse James Garrett* publicó un libro en el que se usaba el término **AJAX** y describió un conjunto de tecnologías, de las cuales JavaScript era la columna vertebral, que se usaba para crear aplicaciones web en las que los datos pueden ser cargados desde el servidor, evitando la necesidad de cargar la página completa y como resultado: aplicaciones más dinámicas. Esto dio lugar a un periodo de renacimiento del uso de JavaScript encabezado por librerías de código abierto y las comunidades que se formaron a su alrededor, con algunas librerías como **JQuery**, **Prototype**…

En julio de 2008 se decide cambiar el nombre de ECMAScript 3.1 a ECMAScript 5 e impulsar el lenguaje.

Todo esto nos trae a la actualidad, con JavaScript que ha entrado en un nuevo y emocionante ciclo de evolución, innovación y normalización, con nuevos desarrollos como la plataforma **nodejs**, que nos permite usar JavaScript en el lado del servidor, y las **APIs de HTML5** que nos ofrecen funcionalidades com acceder a la ubicación del dispositivo, usar sockets…

Y en el año 2015 sale la **ECMAScript 6** que trae muchos cambios significativos al lenguaje.

2. ¿Qué es JavaScript?

Básicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario hacer nada con estos programas, ni siquiera hay que compilarlos. Las aplicaciones escritas en JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos

intermedios.

Entonces, JavaScript es uno de los lenguajes más usados en el mundo, es el único lenguaje de programación que entienden los navegadores, y se puede usar para hacer las páginas web dinámicas, para desarrollar videojuegos, aplicaciones móviles…

Podemos incluir scripts en nuestras páginas web de alguna de las siguientes formas:

- Escribiendo el código JavaScript en archivos externos a nuestras páginas HTML
- Incluyendo scripts en las páginas HTML
- Incluyendo JavaScript dentro de las propias etiquetas HTML

2.1. JavaScript en archivos externos

El código JavaScript se puede escribir en un archivo externo de tipo JavaScript (con la extensión `.js`) que los documentos HTML enlazan con la etiqueta `<script src='archivo.js'>`. Se pueden enlazar todos los archivos JavaScript que se necesiten y cada documento HTML puede enlazar tantos archivos JavaScript como vaya a utilizar. La mayor ventaja de esta forma de usar el código JavaScript, es la reutilización.

/main.js

```
console.log('Código JS en un archivo externo');
```

/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <script src="main.js"></script>
</head>
<body>
</body>
</html>
```

2.2. JavaScript en la página HTML

El código JavaScript se encierra entre las etiquetas `<script>⋯</script>` dentro del archivo HTML, y estas etiquetas se pueden incluir en cualquier parte del documento HTML, aunque se recomienda hacerlo dentro de la cabecera del documento, es decir, entre las etiquetas `<head>⋯</head>`.

Este método se usa cuando vamos a definir un bloque pequeño de código JavaScript, o cuando se quieren incluir códigos específicos en un determinado documento. La principal desventaja es que no se puede reutilizar el código.

/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <script>
    console.log('Código JS en el HTML');
  </script>
</head>
<body>
</body>
</html>
```

2.3. JavaScript en las etiquetas HTML

Esta última forma es la menos usada, ya que consiste en crear bloques de código JavaScript dentro de las etiquetas HTML del documento. Este método solo se usa para definir eventos en las etiquetas.

La mayor desventaja de esta forma es que ensucia innecesariamente el código HTML del documento y hace más dificil el mantenimiento del bloque de código.

/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
</head>
<body>
  <h1 onclick="console.log('Código JS en la etiqueta h1')">Título de la página</h1>
</body>
</html>
```

3. Sintaxis

La sintaxis de JavaScript son aquellas reglas que hay que seguir al escribir el código y que este funcione correctamente. La sintaxis de JavaScript es muy parecia a la de otros lenguajes como Java. Las reglas que hay que tener en cuenta son las siguientes:

- Tipo de las variables: en la mayor parte de los lenguajes a las variables se les indica que tipo de valor van a almacenar, pero en JS no hay que hacerlo, y de esta forma, podemos hacer que una variable contenga distintos tipos de datos a lo largo de la ejecución del script.
- `;` al final de cada instrucción: en la mayor parte de lenguajes, cada instrucción acaba en `;`. En JS no es necesario ponerlo `var j = 0`, aunque se recomienda hacerlo `var j = 0;`.
- No se tienen en cuenta los espacios y los saltos de línea: da igual si añadimos muchos espacios, o algunos saltos de línea para dejar el código más legible, porque JS ignora aquellos espacios que

sobran.

- Es *case sensitive*: en JS una variable llamada *coche* es totalmente distinta a una variable llamada *Coche*, y por lo tanto si no usamos la correcta, el script no funcionará como se espera. No es lo mismo `new Array()` que `new array()`, donde el primero crea un array, mientras que el segundo no.
- Se pueden incluir comentarios: podemos añadir comentarios en nuestro código para aclarar lo que hacen las instrucciones. Podemos incluir comentarios de una sola línea `// ⋯` o de múltiples líneas `/* ⋯ */`.

4. Variables

Las variables nos permiten almacenar datos. Y estos datos pueden cambiar su valor mientras se ejecuta el script.

Las variables en JS se declaran usando la palabra reservada `var` seguida del nombre que le vamos a asignar a la variable. Este nombre nos va a permitir usar la variable en otros sitios del script.

```
var numero;
```

En caso de que el nombre de la variable esté compuesto por más de una palabra, esta se suele escribir en **camelcase**, es decir la primera palabra en minúsculas, y el resto de palabras que van a continuación con la primera letra en mayúscula.

```
var unNumero;
```

Las variables no saben de que tipo son hasta que se les asigna un valor, hasta entonces son de tipo **undefined**. A las variables se les asignan valores de la siguiente forma.

```
var unaVariable = 'Hola mundo!';

unaVariable = 10;
```

Se pueden declarar e inicializar varias variables en una misma línea de código de la siguiente forma:

```
var alto, ancho, area;
alto = 10;
ancho = 5;
area = alto * ancho;

var precio = 1.5, cantidad = 5;
var total = cantidad * precio;
```

Los nombres de las variables deben cumplir las siguientes reglas:

- El nombre de la variable solo puede estar formado por letras, números y los simbolos `$` y `_`.
- El primer carácter del nombre de la variable no puede ser un número, ni un carácter de los que no están permitidos.
- El nombre no puede ser una palabra reservada del lenguaje: for, if, while, var, throw,

4.1. Ejercicio 1

- ¿Cuáles de las siguientes variables está declaradas bien?

```
var una_variable;
var una-variable;
var 1_variable;
var $1variable;
var una;variable;
```

5. Tipos de valores

JavaScript distingue entre los siguientes tipos de valores: strings, números, booleanos, nulos, indefinidos, arrays y objetos.

5.1. String

Los datos de tipo **string** representan cadenas de texto. Estas cadenas tienen que ir entre comillas (simples o dobles) y en una línea.

```
var saludo = 'Hola mundo!';
```

En caso de necesitar mostrar las comillas en el string, hay que usar las comillas que no hemos usado para inicializar la variable o *escaparlas* usando las **backslash** (que le indican al interprete que el siguiente caracter es parte del string).

```
var unTexto = "Esto es un 'texto'";
var otroTexto = 'Esto es \'otro\' texto';
```

5.2. Number

Los datos de tipo **number** representan valores númericos, tanto valores enteros como valores con decimales.

```
var altura = 180;
```

5.3. Boolean

Los datos de tipo **boolean** solo pueden tener uno de los siguientes dos valores, **true** (verdadero) o **false** (falso).

```
var estaEncendido = false;
```

5.3.1. Valores Truthy y Falsy

Debido a la **conversión de tipos**, cada valor en JavaScript puede ser tratado como si fuera un *Boolean*. **Falsy** son los valores que son tratados como si fueran `false`, mientras que **Truthy** son aquellos que son tratados como si fueran `true`.

Table 1. Valores Falsy

Valor
El booleano *false*
El número *0*
Un string vacío '' o con espacios en blanco
El valor indefinido *undefined*
El valor nulo *null*

Table 2. Valores Truthy

Valor
Cualquier otro valor que no sea *Falsy*

5.4. Null

Null es un valor primitivo que representa la ausencia de valor. Cuando se usa en una expresión booleana, actua como *false* y de esta forma podemos comprobar que una variable tiene valor o no.

```
var nula = null;

if (!nula) {
  console.log('Nula es una variable ' + nula)
}
```

5.5. Undefined

Undefined es un valor primitivo que representa que un valor es desconocido, por ejemplo cuando no se le ha asignado un valor a una variable. También actua como *false* cuando se usa en una expresión booleana.

```
var indefinida;

if (!indefinida) {
  console.log('Indefinida es una variable ' + indefinida)
}
```

5.6. Ejercicio 2

- Completar los tests de Jasmine sobre Truthy y Falsy

5.7. Array

Un **array** es un tipo especial de variable que no almacena un valor, sino que almacena una lista de valores. Se suele usar para trabajar con listas o con un conjunto de valores que están relacionados entre ellos. Al crear el array no es necesario indicar cual es el tamaño que va a tener la lista.

Los arrays se pueden crear de las siguientes dos formas:

```
var dias = ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo'];

var colores = new Array('Blanco', 'Negro', 'Rojo', 'Azul', 'Verde');
```

Se recomienda usar la primera forma (**array literal**) a la hora de crear arrays, a no ser que queramos crear un array vacío con una longitud muy alta, en el que vendría mejor usar la segunda forma (**array constructor**):

```
var arrayVacioDeCienElementos = new Array(100); // En lugar de [,,,,,,,,,,,,...,]
```

Los arrays guardan los elementos empezando por la posición 0. Y para poder acceder a los elementos tenemos que poner entre corchetes la posición en la que se encuentra el elemento.

```
var dias = ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo'];

var martes = dias[1];
var domingo = dias[6];
```

Para modificar un elemento, solo hay que indicar el elemento que queremos (indicando la posición entre corchetes) e igualarle el nuevo valor que se le va a dar.

```
var colores = new Array('Blanco', 'Negro', 'Rojo', 'Azul', 'Verde');

colores[4] = 'Amarillo';
colores[5] = 'Naranja';
```

6. Operadores

Los operadores realizan cambios sobre los datos y los podemos clasificar en los siguientes grupos:

Table 3. Operadores aritméticos

Operador	Ejemplo
+ (Suma)	5 + 10 = 15
- (Resta)	8 - 3 = 5
* (Multiplicación)	2 * 4 = 8
/ (División)	9 / 4 = 2.25
% (Módulo)	10 % 3 = 1
++ (Incremento)	a=1; a++; ⇒ a=2;
— (Decremento)	a=5; a--; ⇒ a=4;

Table 4. Operadores de strings

Operador	Ejemplo
+ (Concatenación)	'5' + '10' = '510'
+ (Concatenación)	221 + 'B Baker Street' = '221B Baker Street'

Table 5. Operadores relacionales

Operador	Ejemplo
< (Menor que)	3 < 7 (true)
> (Mayor que)	8 > 9 (false)
<= (Menor o igual que)	2 <= 2 (true)
>= (Mayor o igual que)	9 >= 4 (true)
== (Igual a)	3 == 2 (false)
!= (Distinto de)	4 != 5 (true)

Table 6. Operadores lógicos

Operador	Ejemplo
&& (AND)	10 > 3 && false (false)
|| (OR)	10 > 3 || false (true)
! (NOT)	!(10 > 3) (false)

Table 7. Operadores de asignación (a = 5, b = 4)

Operador	Ejemplo
= (Igual)	a = b (4)
+= (Suma)	a += b => a = a + b (9)
-= (Resta)	a -= b => a = a - b (1)

Operador	Ejemplo
*= (Multiplicación)	a *= b => a = a * b (20)
/= (División)	a /= b => a = a / b (1.25)

Operador de tipo (typeof)

Este operador nos dice el tipo de una variable o valor.

```
// typeof [variable o valor]
var a = 6;
typeof a; // => number
```

7. Consola y Popup en JavaScript

JavaScript nos permite mostrar mensajes en pantalla a través de la *consola* o de un *popup*. Además con los popups podemos hacer que el usuario interaccione con el programa.

7.1. Consola

Los navegadores disponen de una consola sobre la que escribir mensajes que aparecerán en el navegador. Esta es la forma más fácil de mostrar mensajes, y podemos mostrar mensajes de distintos tipos:

- console.log('mensaje'): muestra mensajes generales
- console.warn('mensaje'): muestra mensajes de alerta
- console.error('mensaje'): muestra mensajes de error
- console.dir('mensaje'): muestra las propiedades de un objeto

7.2. Popup

JavaScript también permite mostrar popups en el navegador para mostrar mensajes al usuario o para permitir a este interactuar con el programa. Los distintos tipos de popups que podemos mostrar son:

- alert('mensaje'): muestra un popup con un mensaje de alerta. Este popup tiene un botón de *aceptar* y devuelve un valor `undefined`.
- confirm('mensaje'): muestra un popup con un mensaje (suele ser una pregunta). Este popup tiene dos botones, uno es *cancelar* que en caso de pulsarlo devuelve `false` y el otro es *aceptar* y en este caso devuelve `true`.
- prompt('mensaje'): muestra un popup con un mensaje (se suele pedir que se introduzca una respuesta) y un campo de texto. Este popup tiene dos botones, uno es *cancelar* que en caso de pulsarlo devuelve `null` y el otro es *aceptar* y en este caso devuelve el texto que se haya introducido en el campo.

7.3. Ejercicio 3

- Escribir un script que le haga un par de preguntas al usuario y muestra un mensaje por consola con las respuestas.

8. Lógica condicional e iterativa

Al mirar el diagrama de flujo de cualquier aplicación podemos ver que esta puede tomar distintos caminos, es decir, que puede ejecutar código distinto en diferentes situaciones. Esto se debe a que la aplicación se tiene que comportar de forma distinta dependiendo de las acciones que vaya realizando el usuario en la aplicación. Para decidir que camino hay que coger vamos a usar los siguientes conceptos:

- Evaluaciones: analizan los valores en un script para determinar si se cumple lo esperado (`nota >= 5`).
- Condiciones: dependiendo del resultado de las evaluaciones, decide que parte del código va a ejecutar.
- Bucles: hay veces que es necesario ejecutar la misma instrucción una serie de veces.

En el código vamos a tener muchos lugares donde usaremos condiciones para saber que intrucciones son las que hay que ejecutar.

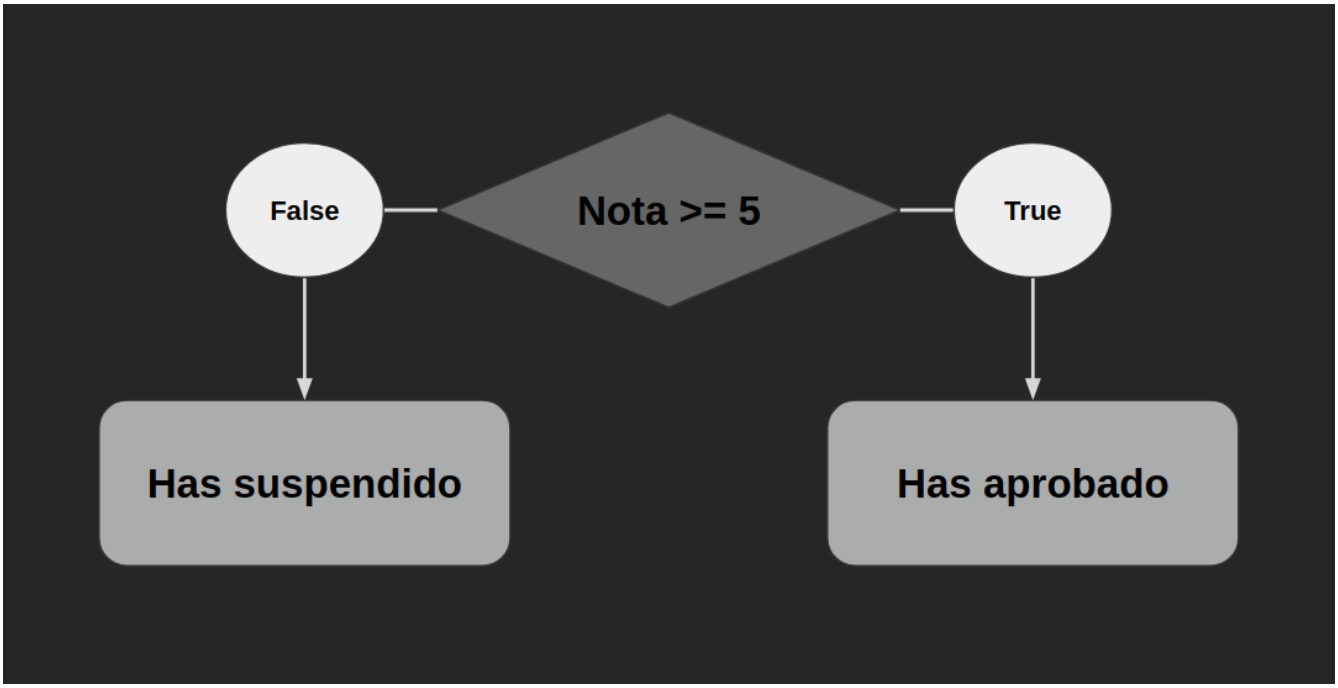

Figure 1. Diagrama de flujo

La evaluación de una condición nos ayuda a tomar las decisiones, y normalmente se realiza comparando dos valores con alguno de los operadores relacionales (`!=`, `===`, `<`, `>=` ...) que devuelven un valor booleano.

8.1. Condicionales

Una vez que ya tenemos el valor que nos devuelven estos operadores, vamos a usar las instrucciones condicionales (`if`, `if ⋯ else` ...) para elegir el código que debe de ejecutarse.

8.1.1. IF

La instrucción condicional `if` comprueba una condición, y si esta condición resulta ser `true`, entonces se ejecutará el bloque de código perteneciente al `if`. En caso de ser `false`, el bloque de código del `if` se salta y la ejecución sigue a partir de donde termina ese bloque.

```
if (nota >= 5) {
  console.log('Has aprobado');
}
```

8.1.2. Ejercicio 4

- Crear el juego de Fizz-Buzz
  - Consiste en preguntar un número al usuario
  - Si el número es múltiplo de 3 se muestra el mensaje *Fizz* junto al número
  - Si el número es múltiplo de 5 se muestra el mensaje *Buzz* junto al número

8.1.3. IF - ELSE

La instrucción condicional `if ⋯ else` también va a comprobar una condición. En este caso, si la condición resulta ser `true` se va a ejecutar el bloque de código del `if`, mientras que si la condición es `false`, se ejecutará el bloque del `else`.

```
if (nota >= 5) {
  console.log('Has aprobado');
} else {
  console.log('Has suspendido');
}
```

8.1.4. Ejercicio 5

- Preguntar al usuario si quiere que le contemos un chiste
  - Si la respuesta es *si* le mostramos un chiste en la consola
  - Si la respuesta es *no* no le mostramos un mensaje al aburrido

8.1.5. IF - ELSE IF

Si necesitamos tener en nuestro código más de un camino, podemos añadir más instrucciones `if` justo después de la instrucción del `else`. De esta forma, si la primera condición no se cumple,

comprobará la del segundo `if` y así sucesivamente hasta el último `else`.

```
if (nota < 5) {
  console.log('Has suspendido');
} else if (nota < 6) {
  console.log('Suficiente');
} else if (nota < 8) {
  console.log('Bien');
} else if (nota < 10) {
  console.log('Notable');
} else {
  console.log('Sobresaliente');
}
```

8.1.6. SWITCH

La instrucción `switch` va a comparar el valor que recibe, con cada uno de los valores que hay en los casos (instrucción `case`), y va a ejecutar el bloque de código correspondiente al caso cuyo valor coincida con el valor que se le pasa al `switch`. El `switch` tiene un caso especial (`default`) cuyo bloque de código se va a ejecutar cuando ninguno de los casos coincida con el valor que se está comparando. Al final de cada `case`, se pone la palabra instrucción `break` que le dice al interprete de JavaScript que ya ha terminado de ejecutar el `switch` y que puede seguir ejecutando el código a partir de su bloque.

```
switch (nota) {
  case 5:
    console.log('Suficiente');
    break;
  case 6:
    console.log('Bien');
    break;
  case 7:
    console.log('Bien alto');
    break;
  case 8:
    console.log('Notable');
    break;
  case 9:
      console.log('Notable alto');
      break;
  case 10:
    console.log('Sobresaliente');
    break;
  default:
    console.log('Suspenso');
    break;
}
```

8.1.7. Operador ternario ?

El operador ternario (**?**) actua como si fuera un `if - else`, solo que en este caso se usa como parte de una expresión en la que una instrucción `if - else` no sería muy práctica.

```
// [condicion] ? [código si la condición es true] : [código si la condición es false]
var resultado = (5 < 3) ? 'Es menor' : 'Es mayor'; // resultado = 'Es mayor'
```

8.2. Bucles

Los bucles van a comprobar una condición. Si esta condición es `true`, el bloque de código se va a ejecutar. Una vez ejecutado el bloque de código, se vuelve a comprobar la condición y si sigue siendo `true` se vuelve a ejecutar otra vez, y así sucesivamente hasta que la condición sea `false` que será cuando deje de ejecutar ese bloque de código correspondiente al bucle.

8.2.1. FOR

El `for` se usa para cuando necesitamos ejecutar un bloque de código un número de veces conocido. Este el bucle más común a la hora de usarse. En este bucle la condición es el número de veces que se tiene que repetir el bloque de código. El valor que se usa en la condición tendremos que inicializarlo, e ir incrementandolo/decrementandolo para que no se quede en un bucle infinito al final de la ejecución del bloque.

```
for (var i = 0; i < 5; i++) {
  console.log('Iteración ' + (i + 1));
}
```

8.2.2. Ejercicio 6

- Haz un programa que cuente hacia atrás 30 números.

8.2.3. Ejercicio 7

- Mejora el ejercicio de Fizz Buzz de antes para que muestre todos los números hasta llegar al que se ha introducido.

8.2.4. FOR IN

El `for in` es una versión del bucle `for` que se usa para iterar sobre Arrays u Objetos. En cada iteración se va a ir guardando en la variable la posición (si se está usando para iterar sobre un objeto, se va a guardar la *clave*).

```
for (var j in [1, 2, 3, 4]) {
  console.log('Posición del array en esta iteración' + j);
}
```

8.2.5. WHILE

En caso de no conocer el número de veces que se tiene que ejecutar el bloque de código, deberíamos usar el `while`, en el que la condición puede ser otra expresión que no sea un contador (como en el `for`) y el código se repetirá hasta que esta condición sea `false`.

```
var num = prompt('Dame un número... Introduce -1 si quieres terminar.');
while (num != -1) {
  console.log('Has introducido el número ' + num);
  num = prompt('Dame un número... Introduce -1 si quieres terminar.');
}
```

8.2.6. Ejercicio 8

- Crea un programa que muestre los 10 primeros números que son múltiplos de 7.

8.2.7. Ejercicio 9

- Mostrar en la consola la canción de Homer Simpson, cojo un muelle y lo tiro por el retrete... hasta llegar a tantos muelles como haya escrito el usuario en un popup.

8.2.8. DO - WHILE

Este bucle es muy parecido al `while`. La única diferencia entre ellos, es que el bloque de código correspondiente a este bucle, se va a ejecutar la primera vez aunque la condición sea `false`. En este caso, primero se ejecuta el bloque, luego comprueba la condición, y a partir de aquí todo funciona igual que el `while`.

```
do {
  var num = prompt('Dame un número... Introduce -1 si quieres terminar.');
  console.log('Has introducido el número ' + num);
} while (num != -1);
```

8.2.9. Ejercicio 10

- Adivina el número al azar
  - El programa saca un número al azar entre 0 y 50
  - El usuario introduce el número que piensa que ha salido
  - Si el usuario acierta se le indica que ha ganado, el número de turnos jugados, y se le pregunta si quiere volver a jugar
  - Si el usuario no acierta:
    - Se le indica si el número que ha dicho es mayor o menor
    - Se le vuelve a pedir un número

9. Funciones

Cuando las aplicaciones se van haciendo las aplicaciones más grandes, es importante organizar el código para que quede limpio y no repetirlo agrupando los bloques de código según la funcionalidad. Para hacer todo eso, vamos a usar las funciones.

Las **funciones** nos permiten agrupar una serie de instrucciones que se encargan de realizar una tarea específica. Si esta tarea se repite en distintas partes de nuestro código podemos usar las funciones para evitar repetir código en todos estos sitios. Estas funciones no se ejecutan directamente cuando se ejecuta el script, sino que se van a ejecutar cuando se realiza una llamada a la función.

Para crear una función se usa la palabra reservada `function` seguida del nombre de la función y el cuerpo de esta (entre llaves `{}`).

```
function saludar() {
  alert('Hola');
}
```

Una vez que una función está declarada, para ejecutar el código que contiene solo hay que llamarla de la siguiente forma:

```
saludar();
```

Estas funciones pueden recibir datos externos para poder realizar las tareas, y estos datos son los **parámetros** que se le pasan entre los paréntesis. Estos parámetros se comportan como **variables** dentro de la función.

```
function calcularArea(ancho, alto) {
  return ancho * alto;
}
var ancho = 5;
var area = calcularArea(ancho, 10);
```

En JavaScript, a las funciones que reciben unos parámetros se les puede pasar más de los que espera recibir. En este caso podemos acceder a los parámetros que se pasan de más en la variable **arguments**. Esta variable la tienen todas las funciones y en ella se encuentran todos los parámetros que recibe la función.

```
function devolverParametro(param1) {
  console.log('Argumentos: ', arguments);
  return param1;
}
var p = devolverParametro('Un texto', 2, true);
```

También podemos no pasarle parámetros aunque esté esperando recibirlos, en este caso estos parámetros que no reciban el valor cuando se llama a la función tendrán el valor **undefined**.

```
function devolverParametro(param1) {
  return param1;
}
var p = devolverParametro();
```

9.1. Ejercicio 11

- Juego Semi-Blackjack
  - Gana quien se acerque más a 20 puntos
  - En cada turno se pregunta si se quiere un número
    - Si es que si, se saca un número al azar entre 1 y 10 y se suma a los puntos que se llevan
    - Si es que no, la máquina saca números hasta pasarse de 17
  - Si te pasas de 20 puntos, pierdes

9.2. Funciones recursivas

Una **función recursiva** es aquella que se llama a sí misma para realizar unos calculas y evitar usar lo bucles. Estas funciones tienen que estar bien pensadas para que no se conviertan en un bucle infinito, tenemos que darle unos casos básicos en los que se terminará de llamar a si misma y devolverá el resultado.

```
function factorial(num) {
  if (num == 1) {
    return 1;
  } else {
    return factorial(num-1) * num;
  }
}

var factorial5 = factorial(5);
```

9.3. Ejercicio 12

- Crear una función que devuelva los primeros 30 números de la serie de fibonacci.
  - Los dos primeros números de la serie de fibonacci son 0 y 1.
  - El resto de números se obtiene sumando los dos anteriores (1, 2, 3, 5, 8...).

9.4. Expresión de funciones y funciones anónimas

Las **expresiones de funciones** son aquellas funciones que se encuentran donde el interprete espera encontrar una expresión, y en estos casos se suelen usar las **funciones anónimas** que son aquellas que no tienen un nombre, y se van a ejecutar en cuanto el interprete se las encuentra.

```
var calcularArea = function(ancho, alto) {
  return ancho * alto;
}

var area = calcularArea(5, 10);
```

9.5. Funciones autoejecutables

Estas funciones no tienen nombre, y se van a ejecutar una vez que el interprete se las encuentre en el script. El interprete tiene que tratar estas funciones como expresiones, por lo que se meten entre parentesis, donde se declara una función anónima y se llama añadiendo justo despues los parentesis (con los argumentos necesarios, en caso de necesitarlos).

```
var saludar = (function() {
    console.log('Hola');
}());
```

Puede ser que esa función necesite parámetros para que realice la tarea correctamente, y en ese caso, los parametros se le pasan en los paréntesis que realizan la llamada a la función.

```
var saludar = (function(nombre) {
    console.log('Hola ' + nombre);
}('Robb'));
```

Estas funciones se suelen usar aquellas veces que el código solo necesita ejecutarse una vez, por ejemplo, para asignar los *listeners* y los *manejadores de eventos* a los elementos cuando estos tienen que ejecutar algún código si el usuario realiza una acción (como pulsar un botón). También evita que haya conflictos entre variables con el mismo nombre que se encuentran en distintos scripts.

10. Scopes

El lugar donde podemos usar las variables depende del lugar en el que las hemos declarado, y esto se conoce como **scope** de variables.

10.1. Variables locales

Cuando declaramos una variable (usando la palabra *var*) dentro de una función, la variable se crea de forma local a la función, por lo que no se podrá acceder al valor de esa variable desde el exterior

de la función. Cada vez que se ejecuta la función se crea la variable, y cuando se termina de ejecutar se destruye esa variable por lo que no se guardará el valor que tenía para las próximas ejecuciones.

```
function suma(n1, n2) {
  var resultado = n1 + n2;
  return resultado;
}

function resta(n1, n2) {
  var resultado = n1 - n2;
  return resultado;
}

suma(5, 5);
console.log(resultado);
resta(6, 2);
```

10.2. Variables globales

Al definir una variable sin la palabra `var`, esta variable será global, y puede darse el caso de que conlleve comportamientos inesperados (conflicto de nombres de variables) por lo que hay que poner la palabra `var` siempre que no vayamos a declarar una variable global. También se toma como variable global aquella que se declara con la palabra `var`, y se declara fuera de cualquier función.

```
function suma(n1, n2) {
  num = n1;
  return num + n2;
}
function muestraConsola(mensaje) {
  console.log('Mensaje: ' + mensaje + ', global = ' + global);
}

muestraConsola('1 + 2 = ' + suma(1, 2));
muestraConsola('num -> ' + num);
var global = 5;
muestraConsola('4 + 6 = ' + suma(4, 6));
num = 10;
muestraConsola('num -> ' + num);
```

11. Objetos

Los **objetos** agrupan un conjunto de variables y funciones que representan cualquier cosa que podamos necesitar, por ejemplo, un coche, una persona, una serie...

Las variables que se definen en el objeto se llaman **propiedades** y nos dan información sobre el objeto, por ejemplo la marca del coche, o el nombre de una persona.

Mientras que las funciones se llaman **métodos** y representan tareas asociadas a ese objeto, por ejemplo mostrar la velocidad a la que va el coche, o cambiar el valor que indica si una serie ha finalizado o no. Y para acceder a las propiedades del objeto dentro de un método se usa la propiedad `this` del objeto.

11.1. Creando objetos de forma literal

Un objeto se crea añadiendo estas claves-valores entre llaves como se puede ver a continuación:

```
var serie = {
  nombre: 'Vikings',
  temporadas: 5,
  episodios: 69,
  episodiosVistos: 45,
  episodiosPorVer: function() {
    return this.episodios - this.episodiosVistos;
  }
}
```

Una vez que hemos creado un objeto, podemos acceder a sus propiedades y métodos de varias formas.

11.2. Acceso a propiedades: Notación de punto

Está es la forma que se suele usar para acceder a las propiedades y métodos de los objetos. Solo hay que poner un `.` seguido del nombre de la propiedad/método al que queramos acceder detrás del objeto.

```
var numTemp = serie.temporadas;
var episodiosSinVer = serie.episodiosPorVer();
```

11.3. Acceso a propiedades: Usando corchetes

Está forma de acceso se suele usar cuando el nombre de la propiedad es un *número* (deberiamos de evitar poner números como nombres de las propiedades), o cuando vamos a usar el valor de una variable como nombre de la propiedad. En este caso hay que poner el nombre de la propiedad a la que queremos acceder entre corchetes.

```
var numEpisodios = serie['episodios'];
```

11.4. Creando objetos con el constructor

Otra forma de crear los objetos es usando `new Object()`, lo que crearía un objeto en blanco. Una vez creado el objeto, se le pueden añadir las propiedades y métodos usando la notación de punto o los corchetes (para acceder a la propiedad/método) y dandole un valor (al no estar definida esa propiedad, se añade con el valor dado).

```
var serieDirk = new Object(); // equivalente a -> var serieDirk = {};
serieDirk.nombre = "Dirk Gently's Holistic Detective Agency";
serieDirk['temporadas'] = 2;
serieDirk['episodios'] = 18;
serieDirk.episodiosVistos = 16;
serieDirk.episodiosPorVer = function() {
  return this.episodios - this.episodiosVistos;
};
```

11.5. Actualizando los objetos

Para actualizar el valor de las propiedades de los objetos, solo hay que darle un nuevo valor a estas (tanto con usando la notación de punto, o los corchetes).

```
serieDirk.episodiosVistos = 17;
serieDirk['episodiosVistos'] = 18;
```

Si se intenta actualizar una propiedad que no existe en el objeto, esta se creará con el valor que se le está asignando.

```
serieVikings.finalizada = false;
serieDirk.finalizada = false;
```

11.6. Eliminando propiedades

Se pueden eliminar las propiedades de un objeto usando **delete** seguido de la propiedad a eliminar.

```
delete serieVikings.finalizada;
```

En caso de querer unicamente limpiar el valor de la propiedad, sirve con asignarle un string vacio.

```
serieDirk.finalizada = '';
```

11.7. Creando varios objetos

Hasta ahora hemos creado dos objetos que representan lo mismo pero con distintos valores. Se pueden usar funciones como plantillas (clases) para crear estos objetos, a las cuales le vamos a pasar como parámetros los valores con los que vamos a inicializar sus propiedades. El nombre de las funciones constructoras empiezan en mayúscula por convención, de esta forma sirve para recordar que hay que usar la palabra **new** a la hora de crear los objetos.

```
function Serie (nombre, temporadas, episodios, episodiosVistos) {
  this.nombre = nombre;
  this.temporadas = temporadas;
  this.episodios = episodios;
  this.episodiosVistos = episodiosVistos;
  this.episodiosPorVer = function () {
    return this.episodios - this.episodiosVistos;
  }
}
```

La palabra **this** indica que la propiedad o el método pertenece al objeto que crea esta función. Se usa en lugar de usar el nombre del objeto como se ha visto anteriormente.

Para crear instancias de los objetos usando estas funciones constructoras, tenemos que ponerlas después de la palabra **new**.

```
var breakingBad = new Serie("Breaking Bad", 5, 62, 54);
var sonsOfAnarchy = new Serie("Sons of Anarchy", 7, 92, 92);
```

11.8. Ejercicio 13

- Crea un array con unos cuantos objetos *Serie* como los que hemos visto en el anterior ejemplo
- Añadeles una propiedad *finalizada* de tipo booleana
- Crea una función que devuelva un array con aquellos que no han finalizado

11.9. Arrays

Los **arrays** son un tipo especial de objeto que guarda un conjunto de pares clave-valor, pero en este caso la clave es la posición en la que se encuentran los elementos en la lista.

Se pueden combinar con los objetos para crear estructuras de datos más complejas, por ejemplo los arrays pueden guardar una serie de objetos, y los objetos pueden guardar arrays. En los objetos el orden en que aparecen las propiedades no es importante, pero en los arrays, la posición indica el orden de las propiedades.

```
sonsOfAnarchy.aparicionPersonajesTemporadas = {
  'jax': [1, 2, 3, 4, 5, 6, 7],
  'chibs': [1, 2, 3, 4, 5, 6, 7],
  'nero': [5, 6, 7]
};

breakingBad.datosTemporadas = [
  { anyo: 2008, notaMedia: 8.8 },
  { anyo: 2009, notaMedia: 8.5 },
  { anyo: 2010, notaMedia: 9.1 }
];
```

11.10. Ejercicio 14

- Crea una función que dado un array, devuelva el mismo array, pero con los números pares cambiados por el texto 'PAR'.

12. Objetos intrínsecos

Los navegadores traen una serie de objetos intrínsecos que representan cosas como la ventana del navegador, o la página web que se está mostrando. Estos objetos nos van a permitir crear páginas web interactivas.

Los objetos que nosotros creamos, los diseñamos para ajustarlos a nuestras necesidades, mientras que los objetos intrínsecos, tienen definida una funcionalidad que es usada por muchos scripts. Estos objetos nos van a permitir acceder a información como el ancho de la ventana del navegador, las coordenadas donde se encuentra el ratón, el contenido de la página web... Y a estas propiedades podemos acceder al igual que hacemos con nuestros objetos.

Estos objetos los podemos dividir en tres grupos:

- **Browser Object Model**: este grupo contiene objetos que nos van a dar información sobre la ventana del navegador, el historial, la pantalla del dispositivo...
- **Document Object Model**: este grupo contiene objetos que representan el contenido de la página actual. Se crea un objeto por cada elemento dentro de la página (cabecera, footer, botón, campo de texto...).
- **Objetos Javascript Globales**: este grupo contiene objetos que representan cosas que vamos a necesitar usar en JavaScript como obtener la fecha y hora, u obtener el valor de constantes matemáticas...

13. Browser Object Model

En este grupo de objetos, el que está por encima del resto es el objeto **window** que es el que representa la ventana actual del navegador. Este objeto tiene otros objetos hijos, que agrupan información sobre otras características del navegador.

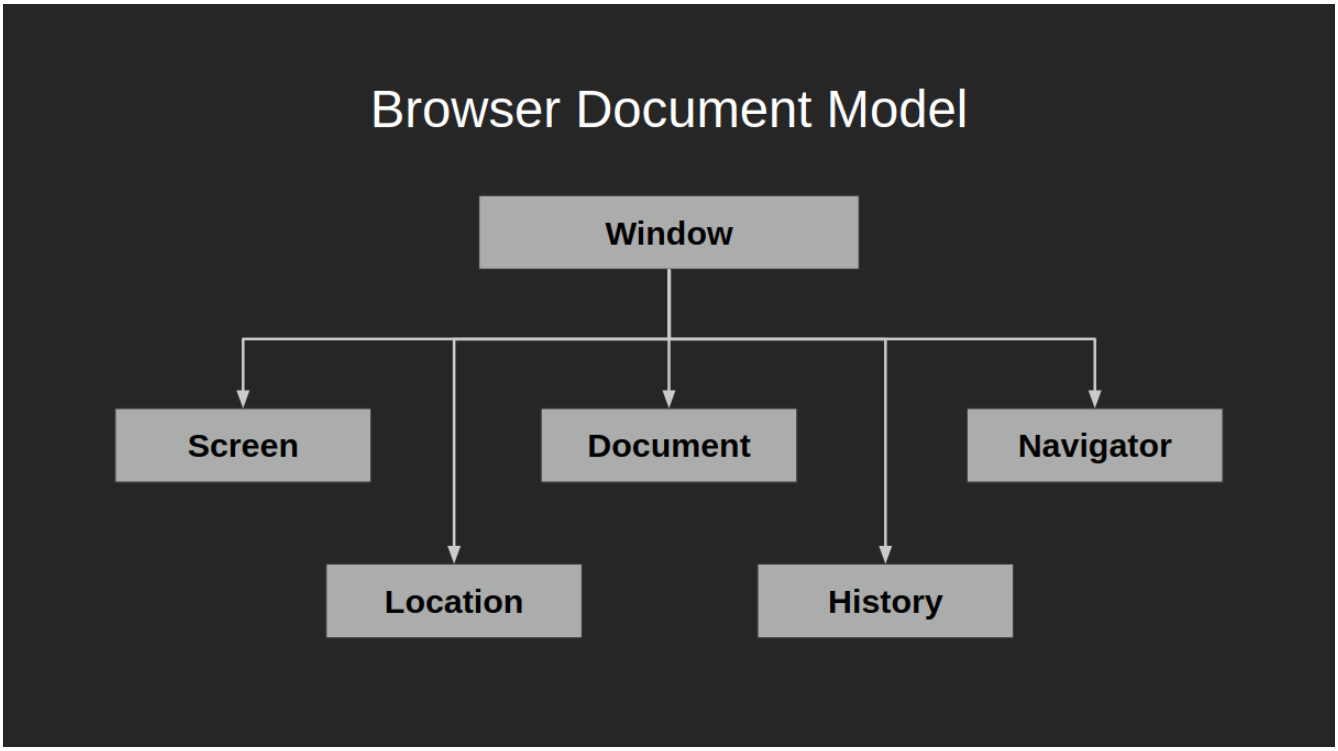

Figure 2. Browser Object Model

13.1. Window

El objeto **window** representa la ventana actual del navegador y este objeto tiene unas propiedades y métodos que se pueden ver a continuación:

Table 8. Propiedades de window

Propiedad	**Descripción**
window.innerHeight	Altura de la ventana
window.innerWidth	Ancho de la ventana
window.pageXOffset	Distancia que se ha hecho scroll horizontalmente en la ventana
window.pageYOffset	Distancia que se ha hecho scroll verticalmente en la ventana
window.screenX	Coordenada X a la que se encuentra el navegador de la esquina superior izquierda de la pantalla
window.screenY	Coordenada Y a la que se encuentra el navegador de la esquina superior izquierda de la pantalla

Table 9. Métodos de window

Método	**Descripción**	**Ejemplo**
window.alert(msg)	Muestra un popup con el mensaje *msg* y un botón OK	window.alert("Hola mundo");

Método	Descripción	Ejemplo
window.confirm(msg)	Muestra un popup con el mensaje *msg* y dos botones, *Aceptar* (devuelve *true*) y *Cancelar* (devuelve *false*).	var res = window.confirm("Quieres continuar?");
window.prompt(msg)	Muestra un popup con el mensaje *msg*, un campo de texto y dos botones, *Aceptar* (devuelve el texto introducido) y *Cancelar* (devuelve *null*).	var res = window.prompt("Cómo te llamas?");
window.open(url)	Abre una pestaña nueva en el navegador con la URL *url*.	window.open("https://www.google.es");
window.print()	Abre la ventana de impresión del navegador con la página actual.	window.print();
window.scrollBy(x, y)	Hace scroll horizontal (*x* pixeles) y vertical (*y* pixeles).	window.scrollBy(0, 1000);

13.2. History

El objeto **history** es uno de los hijos del objeto **window**, lo que quiere decir que es un objeto que se encuentra dentro de otro objeto. Este objeto nos va a dar información sobre el historial del navegador y metodos para cambiarlo.

Table 10. Propiedades de History

Propiedad	Descripción
window.history.length	Devuelve el número de páginas almacenadas en la pila de navegación

Table 11. Métodos de History

Método	Descripción	Ejemplo
window.history.back()	Vuelve una página atrás. Es como pulsar el botón de *retroceder* en el navegador	window.history.back();
window.history.forward()	Va una página adelante. Es como pulsar el botón de *avanzar* en el navegador	window.history.forward();
window.history.go(numPaginas)	Va *numPaginas* hacia delante (valor positivo) o hacia atrás (valor negativo)	window.history.go(-2);

13.3. Location

El objeto **location** es otro de los hijos del objeto **window** y este se encarga de darnos información sobre la URL de la página actual.

Table 12. Propiedades de Location

Propiedad	Descripción
window.location.href	Devuelve la URL de la página actual
window.location.port	Devuelve el puerto
window.location.host	Devuelve el host
window.location.pathname	Devuelve el pathname
window.location.protocol	Devuelve el protocolo
window.location.search	Devuelve los parámetros de búsqueda de la página

Table 13. Métodos de Location

Método	Descripción	Ejemplo
window.location.reload()	Recarga la página actual	window.location.reload();
window.location.replace(url)	Reemplaza la página actual por la de la URL *url*	window.location.replace(url);

13.4. Navigator

El objeto **navigator** también es hijo del objeto **window** y nos proporciona información sobre el navegador y acceso al hardware del dispositivo como puede ser a la batería o la localización.

Table 14. Propiedades de Navigator

Propiedad	Descripción
window.navigator.appName	Devuelve el nombre oficial del navegador (puede devolver un valor erroneo)
window.navigator.appVersion	Devuelve la versión del navegador (puede devolver un valor erroneo)
window.navigator.language	Devuelve el lenguaje del navegador elegido por el usuario
window.navigator.languages	Devuelve un array de lenguajes que va a usar el navegador por orden de preferencia
window.navigator.onLine	Devuelve un booleano indicando si el navegador se encuentra trabajando en línea
window.navigator.platform	Devuelve la plataforma en la que el navegador se encuentra trabajando (puede devolver un valor erroneo)
window.navigator.vendor	Devuelve el nombre del fabricante del navegador actual

Table 15. Métodos de Navigator

Método	Descripción	Ejemplo
window.navigator.getBattery()	Devuelve una promesa que cuando se cumple nos da información de la bateria	window.navigator.getBattery();
window.navigator.geolocation	Nos da acceso a la localización del dispositivo	window.navigator.geolocation.getCurrentPosition(function(position) { … });
window.navigator.vibrate(patron)	Hace que aquellos dispositivos que lo soportan vibren siguiente el patrón de vibraciones y pausas. Este valor puede ser un número (tiempo que va a vibrar el dispositivo, solo una vez) o un array de números (alterna entre tiempo que vibra y tiempo que está en pausa)	window.navigator.vibrate();

13.5. Screen

El objeto **screen** también es hijo del objeto **window** y nos da información sobre la pantalla del dispositivo en que se ha abierto el navegador.

Table 16. Propiedades de Screen

Propiedad	Descripción
window.screen.width	Devuelve el ancho de la pantalla (no del navegador)
window.screen.height	Devuelve la altura de la pantalla (no del navegador)
window.screen.orientation	Devuelve información sobre la orientación de la pantalla

13.6. Document

El objeto **document** es otro de los hijos del objeto **window** y nos va a este nos va a permitir interactuar con la página web que se ha cargado, permitiendo modificarla mediante código. Este objeto lo veremos más adelante.

14. Global JavaScript Object

Los objetos globales son un grupo de objetos individuales que se encargan de algunas de las distintas partes del lenguaje JavaScript. Estos objetos los podemos dividir en:

- Aquellos que representan tipos de datos básicos (String, Boolean y Number).
- Aquellos que ayudan con algunos conceptos usados en el mundo real (Date, Math y Regex).

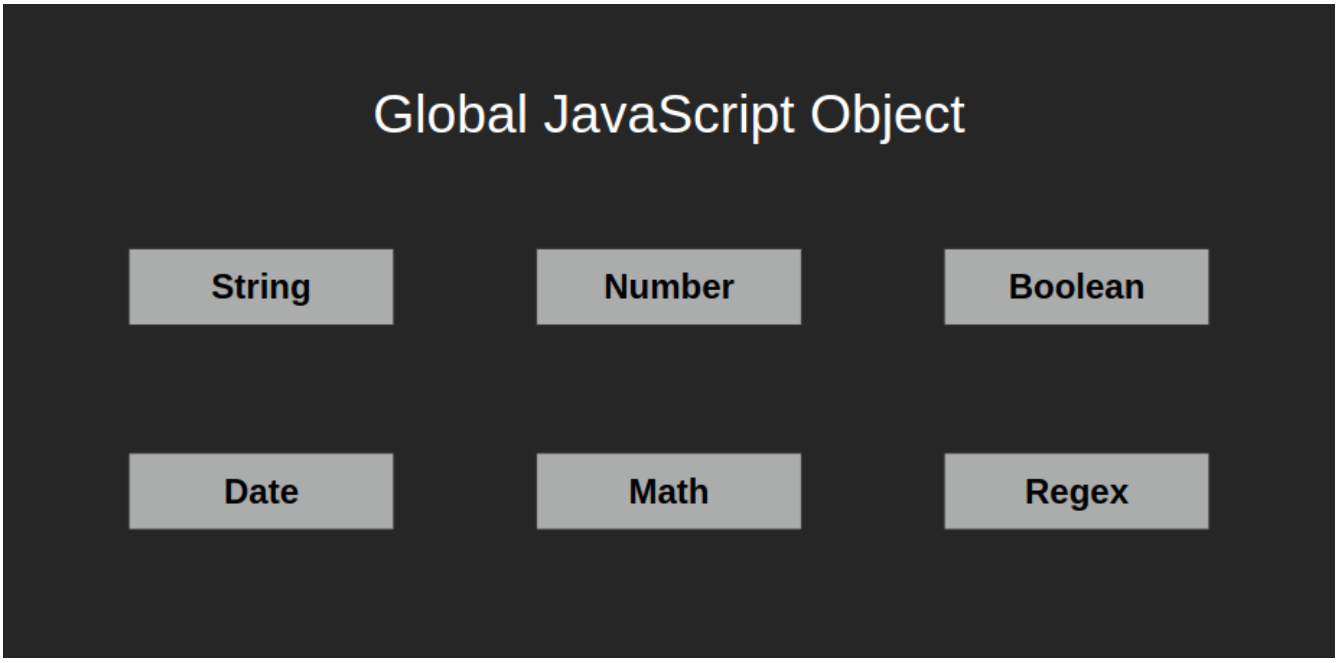

Figure 3. Global JavaScript Object

14.1. String

Siempre que tenemos un string podemos acceder a las siguientes propiedades y métodos para trabajar con el.

Table 17. Propiedades de String

Propiedad	**Descripción**
length	Devuelve la longitud del String

Table 18. Métodos de String

Método	**Descripción**	**Ejemplo**
toUpperCase()	Convierte el string a mayúsculas	"hola mundo".toUpperCase()
toLowerCase()	Convierte el string a minúsculas	"HOLA MUNDO".toLowerCase()
charAt(index)	Devuelve el caracter que se encuentra en la posición *index* del String	"Hola mundo".charAt(2)
indexOf(char)	Devuelve la posición en la que se encuentra el primer caracter *char* del String	"Hola mundo".indexOf('m')
lastIndexOf(char)	Devuelve la posición en la que se encuentra el último caracter *char* del String	"Hola mundo".lastIndexOf('m')

Método	Descripción	Ejemplo
substring(index1, index2)	Devuelve el String que hay entre las dos posiciones dadas como parámetros, donde la primera posición (*index1*) se incluye, y la segunda (*index2*) no se incluye	"Hola mundo".substring(0, 6)
split(char)	Separa el String en un Array con los substring separados por el caracter *char*	"Hola a todos".split(' ')
trim()	Elimina los espacios del inicio y final del String	" Hola a todos ".trim()
replace(str1, str2)	Reemplaza el *str1* por el *str2* en el String. Por defecto solo reemplaza la primera aparición de *str1* en el String.	"Hola a todos".replace('Hola', 'Adios')

14.2. Ejercicio 15

- Crear una función que te dice si un string es *palindromo* o no.
  - Una cadena de texto es palindroma si se lee igual de izquierda a derecha que de derecha a izquierda.

14.3. Ejercicio 16

- Crear una función que te dice si todas las letras del string que recibe están en *mayúsculas*, *minúsculas* o son *mixtas*.

14.4. Number

A continuación se muestran los métodos que se pueden usar con los números.

Table 19. Métodos de Number

Método	Descripción	Ejemplo
Number.isInteger(num)	Comprueba si el número *num* es un entero (*true*) o no (*false*)	Number.isInteger(2.4)
Number.isNaN(num)	Comprueba si el parámetro *num* es un NaN (*true*) o no (*false*)	Number.isNaN("un string")
toFixed(numDecimales)	Redondea un número decimal a tantos *numDecimales* decimales y lo devuelve como String	2.43467.toFixed(3)
toPrecision(num)	Devuelve un String con tantos números como *num* le llega como parámetro	2.43467.toPrecision(3)

Método	Descripción	Ejemplo
toExponential(num)	Devuelve un String representando el número en una notación exponencial	0.0000000025.toExponential(3)

14.5. Array

Para trabajar con los Arrays, podemos usar las siguientes propiedades y métodos.

Table 20. Propiedades de Array

Propiedad	Descripción
length	Devuelve la longitud del Array

Table 21. Métodos de Array

Método	Descripción	Ejemplo
pop()	Elimina el último elemento del Array y lo devuelve	[1, 3, 5, 7, 2].pop()
push(it1, it2, ...)	Añade uno o varios elementos al final del array y devuelve la nueva longitud	[1, 3, 5, 7, 2].push(4, 3)
shift()	Elimina el primer elemento del Array y lo devuelve	[1, 3, 5, 7, 2].shift()
unshift(it1, it2, ...)	Añade uno o varios elementos al principio del array y devuelve la nueva longitud	[1, 3, 5, 7, 2].unshift(4, 11)
reverse()	Da la vuelta a todos los elementos del Array	[1, 3, 5, 7, 2].reverse()
sort()	Ordena los elementos del Array	[1, 3, 5, 7, 2].sort()
join(str)	Une todos los elementos del Array separandolos por el *str* y los devuelve en forma de String	[1, 3, 5, 7, 2].join(', ')
splice(pos, numElem, el1, el2, ...)	Desde la posición *pos* elimina *numElem* elementos y añade los elementos *el1, el2, ...*	[1, 3, 5, 7, 2].splice(1, 1, 10, 11)
slice(pos1, pos2)	Devuelve un Array con los elementos del Array entre *pos1* y *pos2*	[1, 3, 5, 7, 2].slice(1, 3)
indexOf(item)	Devuelve la posición del primer elemento *item* que se encuentra en el Array. Si no existe ese *item* en el Array, devuelve -1	[1, 3, 5, 7, 2].indexOf(3)
lastIndexOf(item)	Devuelve la posición del último elemento *item* que se encuentra en el Array. Si no existe ese *item* en el Array, devuelve -1	[1, 3, 5, 7, 2].lastIndexOf(0)

Método	Descripción	Ejemplo
forEach()	Ejecuta una función por cada elemento del Array. Esta función recibe como parámetros el elemento y la posición en la que se encuentra	[1, 3, 5, 7, 2].forEach(function(elem, pos) {…})
map()	Devuelve un nuevo Array con los elementos que devuelve la función que recibe como parámetro	[1, 3, 5, 7, 2].map(function(elem, pos) {…})
filter()	Devuelve un nuevo Array con aquellos elementos que cumplen con la condición establecida en la función que recibe como parámetro	[1, 3, 5, 7, 2].filter(function(elem, pos) {return true;})
concat(arr1, arr2, …)	Devuelve un Array con la unión de varios Arrays	[1, 3, 5, 7, 2].concat([2, 5], [1, 8])

14.6. Ejercicio 17

- Crea una función que devuelva el número de temporadas en las que han coincidido Nero y Chibs del ejemplo visto anteriormente.

14.7. Ejercicio 18

- Crea una función que funcione como el método *filter*
  - Recibe como primer parámetro un array
  - Recibe como segundo parámetro una función
  - Devuelve un array con aquellos elementos que cumplen la condición que hay en la función

14.8. Date

Para poder trabajar con las fechas, tenemos que crear una instancia del objeto **Date** que nos dará la fecha actual. Esta fecha se obtiene del reloj del dispositivo, por lo que cada usuario obtendra la fecha actual correspondiente a su zona horaria.

```
var hoy = new Date();
```

También podemos crear la fecha que queremos representar pasandole al constructor del objeto unos parámetros con los siguientes formatos.

```
var fecha1 = new Date(1992, 5, 10);
var fecha2 = new Date(1992, 5, 10, 20, 14, 38);
var fecha3 = new Date(1992, 5, 10);
```

Table 22. Métodos de Date

Método	Descripción	Ejemplo
getDate()	Devuelve el día del mes (1-31)	fecha.getDate()
setDate()	Establece el día del mes (1-31)	fecha.setDate(2)
getMonth()	Devuelve el mes (0-11)	fecha.getMonth()
setMonth()	Establece el mes (0-11)	fecha.setMonth(6)
getFullYear()	Devuelve el año (4 dígitos)	fecha.getFullYear()
setFullYear()	Establece el año (4 dígitos)	fecha.setFullYear(2015)
getDay()	Devuelve el día de la semana (0-6)	fecha.getDay()
getHours()	Devuelve la hora (0-23)	fecha.getHours()
setHours()	Establece la hora (0-23)	fecha.setHours(20)
getMinutes()	Devuelve los minutos (0-59)	fecha.getMinutes()
setMinutes()	Establece los minutos (0-59)	fecha.setMinutes(13)
getSeconds()	Devuelve los segundos (0-59)	fecha.getSeconds()
setSeconds()	Establece los segundos (0-59)	fecha.setSeconds(5)
getMilliseconds()	Devuelve los milisegundos (0-999)	fecha.getMilliseconds()
setMilliseconds()	Establece los milisegundos (0-999)	fecha.setMilliseconds(154)
getTime()	Devuelve el número de milisegundos que han pasado desde el 01/01/1970 (si la fecha es anterior a esta se devuelve un número negativo)	fecha.getTime()
setTime()	Establece el número de milisegundos que han pasado desde el 01/01/1970 (si la fecha es anterior a esta hay que establecer un número negativo)	fecha.setTime(711064800000)
toDateString()	Devuelve un String con solo la fecha	fecha.toDateString()
toTimeString()	Devuelve un String con solo la hora	fecha.toTimeString()
toString()	Devuelve la fecha y hora como String	fecha.toString()

Método	Descripción	Ejemplo
getTimezoneOffset()	Devuelve en minutos la diferencia de la zona horaria para la configuración	fecha.getTimezoneOffset()

No hay métodos que nos devuelvan los nombres de los días o de los meses porque estos pueden variar entre los distintos lenguajes. Para obtenerlos, podemos usar un array para guardar los nombres y así poder acceder a ellos.

14.9. Math

El objeto **Math** tiene propiedades y métodos que podemos usar para realizar operaciones matemáticas.

Table 23. Propiedades de Math

Propiedad	Descripción
Math.PI	Devuelve el valor del número PI

Table 24. Métodos de Math

Método	Descripción	Ejemplo
Math.round(num)	Redondea un número al entero más cercano	Math.round(2.6)
Math.sqrt(num)	Devuelve la raíz cuadrada de un número	Math.sqrt(9)
Math.ceil(num)	Redondea un número al entero superior más cercano	Math.ceil(2.2)
Math.floor(num)	Redondea un número al entero inferior más cercano	Math.floor(2.9)
Math.random()	Devuelve un número al azar entre 0 (incluido) y 1 (no incluido)	Math.random()
Math.min(n1, n2, ...)	Devuelve el número mínimo de los números que se le pasan como argumentos	Math.min(3, 6)
Math.max(n1, n2, ...)	Devuelve el número máximo de los números que se le pasan como argumentos	Math.max(3, 6)
Math.abs(num)	Devuelve el absoluto de un número	Math.abs(-3)
Math.sin(num)	Devuelve el seno de un número (en radianes)	Math.sin(90)
Math.cos(num)	Devuelve el coseno de un número (en radianes)	Math.cos(90)

Método	Descripción	Ejemplo
Math.tan(num)	Devuelve la tangente de un número (en radianes)	Math.tan(90)

14.10. JSON

El objeto **JSON** tiene unos métodos que nos van a permitir trabajar con objetos JSON y objetos JavaScript.

Table 25. Métodos de JSON

Método	Descripción	Ejemplo
JSON.parse(objJson)	Convierte un objeto JSON en un objeto JavaScript	JSON.parse("{\"nombre\":\"Ramsay\",\"apellido\":\"Bolton\"}");
JSON.stringify(obj)	Convierte un objeto JavaScript en un objeto JSON	JSON.stringify({ nombre: 'Ramsay', apellido: 'Bolton' })