Objetos predefinidos de Javascript

Objetos predefinidos de JavaScript

Javascript no es un lenguaje orientado a objetos basado en clases, sino en prototipos, ya que, los objetos pueden crearse no a partir de clases, sino a partir de otros objetos que funcionan a modo de «plantilla».

ECMAScript 2015 introdujo *class* para poder trabajar con clases al modo de la orientación a objetos clásica

Un objeto es una forma de guardar propiedades y métodos de forma organizada. Las propiedades son valores y los métodos son acciones.

Entre los objetos predefinidos de que dispone Javascript, el más importante es **Window**, ya que el es el objeto global en el que se ejecuta el programa.

WINDOW es el objeto global de Javascript cuando lo ejecutamos en un navegador y representa por tanto la ventana del navegador y el contenido que se carga en ella. En otros entornos de ejecución de Javascript (como en node.js), en los que no existe por tanto navegador, el objeto global es **Global**. Ambos coinciden con **GlobalThis**, por lo que puede usarse en cualquier entorno este objeto, que equivaldrá a Window en entorno de navegador y a Global en otros entornos.

Para acceder a las propiedades y métodos de un objeto usamos la notación punto. Por ejemplo, así podemos acceder al método **alert()** del objeto **Window**:

```
window.alert();
```

Sin embargo, como hemos visto, podemos escribir directamente **alert()**, ya que al ser Window el objeto global, de él dependen el resto de objetos y métodos, por lo que no es necesario escribirlo para acceder a sus métodos y propiedades.

Los cuadros de diálogo **alert()**, **prompt()** y **confirm()**, que hemos venido usando, son por tanto métodos del objeto window. Otros métodos globales que usamos con frecuencia son **isNaN(numero)** para comprobar si el parámetro es un número (devuelve false si es un número); **parseInt(cadena)**, que convierte la cadena en un número entero y **parseFloat(cadena)**, que convierte la cadena en un número de coma flotante.

Otro método global es **open()**, que permite abrir nuevas ventanas que, a su vez, tienen un objeto **Window** con sus correspondientes propiedades y métodos a los que podremos acceder, siempre que esté en el mismo dominio:

```
ventana =window.open("http://www.elmundo.es","nombre", "width=500,height=300,top=100,left=100");

ventana.location="https://www.elpais.es";
```

También por supuesto tenemos el método **close()** para cerrar ventanas, el método **print()** para imprimir, **focus()** para llevar el foco a una ventana, **blur()** para quitar el foco. **scrollTo(x,y)** para desplazar la ventana a una posición concreta y **scrollBy(x,y)** para desplazar la ventana una cantidad de píxeles concreta. Lo mismo sucede con **resizeTo(x,y)** y **resizeBy(x,y)**, si bien, estos métodos solo pueden usarse con ventanas abiertas con el método open y que tengan el mismo origen (y preferentemente que al abrirse se ha añadido a las opciones el valor "resizable"):

```
ventana =window.open("http://www.elmundo.es","nombre",
"width=500,height=300,top=100,left=100,resizable=yes");
```

La variable que usamos para la ventana, conviene declararla antes, así podemos comprobar si la variable tiene un valor indefinido en caso de no haberle aún asignado valor (es decir, cuando no se ha abierto). También podemos comprobar si se ha cerrado preguntando por la propiedad **closed**:

ventana.closed tendrá valor true si está abierta y false si está cerrada.

Los métodos **setTimeout()**, **setInterval()**, **clearTimeout()** y **clearInterval()** también pertenecen al objeto global. **setTimeout()** nos permite programar una instrucción para ejecutarse transcurrido un tiempo expresado en milisegundos, por ejemplo: *setTimeout (miFuncion,1000*). Si queremos impedir la ejecución podemos usar clearTimeout(). Por su parte **setInterval()** funciona de forma similar pero repite las acciones iterativamente hasta que se interrumpe usando **clearInterval()**, por ejemplo:

miIntervalo=setTimeout (miFuncion,1000); clearInterval(miIntervalo);

Para animaciones es preferible usar el método **requestAnimationFrame()** mejor que setInterval(), que no necesita expresar el tiempo, ya que utiliza la frecuencia de refresco de una animación: **requestAnimationFrame(miFuncion)**

Las variables globales declaradas con *var* son también propiedades del objeto Window, por lo que podríamos acceder a su valor mediante *window.nombreVariable*, algo que no sucede si son declaradas con let.

Pero Window tiene también propiedades con información sobre la ventana a las que podemos acceder igualmente con la notación punto, por ejemplo:

innerHeight e **innerWidth** para saber la altura y anchura de la ventana del navegador

outerHeight y **outerWidth** para saber la altura y anchura incluyendo las barras de herramientas y de desplazamiento.

Window también dispone de las propiedades **localStorage** y **sessionStorage** para almacenar información del usuario. Para ello cuenta con los métodos **setItem()** y **getItem()** para almacenar y recuperar pares de datos (nombre, valor).

Así con el método **setItem()** podemos guardar cualquier información:

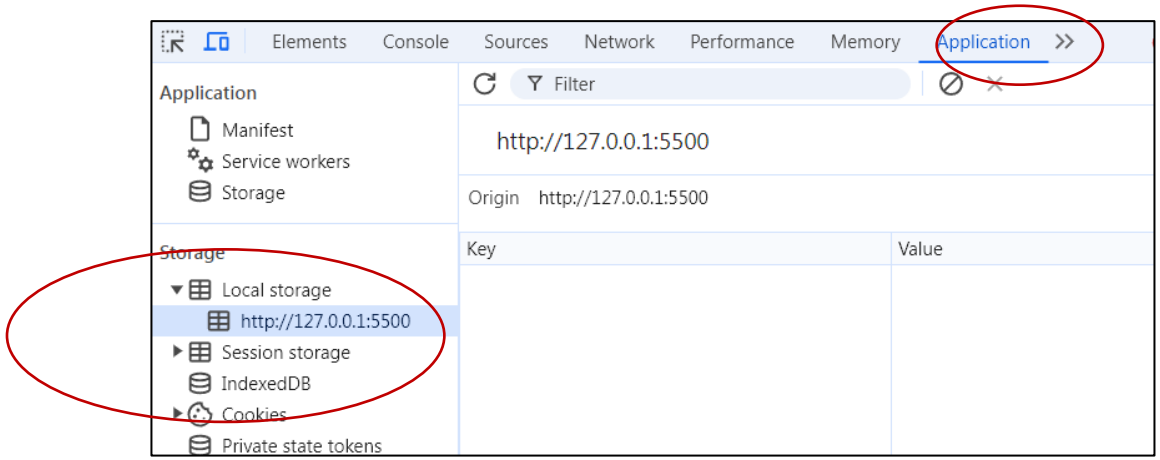
localStorage.setItem("nombreValor",valor)

Y con el método **getItem()** podemos recuperar el valor almacenado:

localStorage.getItem("nombreValor")

LocalStorage mantiene la información, aunque se cierre el navegador, mientras **sesionStorage** solo se almacena mientras se mantenga abierta la sesión del navegador, es decir hasta que este se cierra.

Podemos gestionar los valores que una página guarda en nuestro almacenamiento, desde el panel de herramientas de desarrollador del navegador en la pestaña Performance:



Para eliminar los datos guardados podemos usar los métodos **localStorage.clear()** y **sessionStorage.clear()**

Sobre el objeto Window podemos también controlar determinados eventos como la carga completa de la página (**load**), cuando se abandona la página (**unload**), el paso previo al abandono de la página, cuando la ventana se redimensiona (**resize**), se usa la barra de scroll (**scroll**) la ventana recibe el foco (**focus**) o lo pierde (**blur**), cuando hay conexión a internet o no la hay (**online** y **offline**), cuando se navega hacia atrás o adelante a través del historial (**popstate**), cuando se cambia el marcador (#) de la URL (**hashchange**)

Por último Window también tiene un evento **error** que se desencadena cuando se produce un error en Javascript que no esté controlado por el propio programa: **window.onerror=()=> alert("se ha producido un error")** y un evento de comunicación entre ventanas que se desencadena cuando recibe mensajes (**message**)

Para capturar el evento añadimos **on** al nombre del evento y lo asignamos a una función:

```
window.onresize=function(){alert(“la ventana se ha redimensionado)};

window.onload=nombreFuncion; /*Sin paréntesis para evitar que se ejecute*/
```

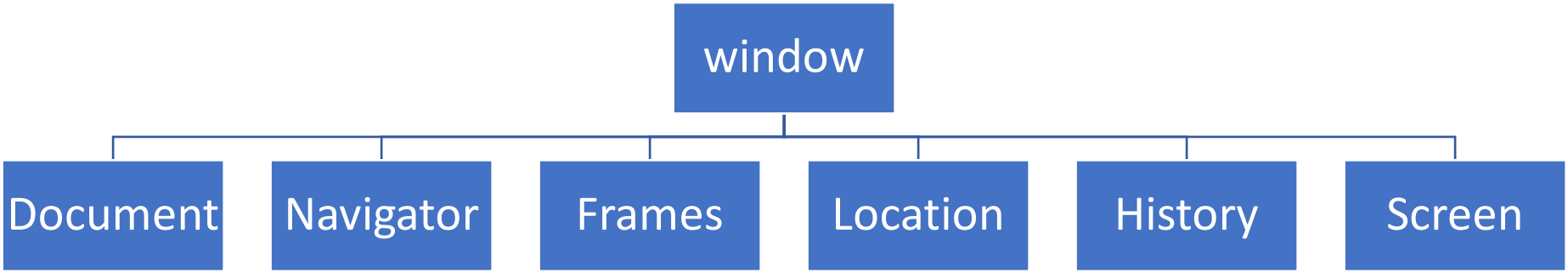
O bien podemos usar el método **addEventListener** del objeto **Window**, en cuyo caso usamos el nombre del evento sin **on**. Este evento tiene como parámetros el evento y la función que queremos desencadenar, que puede ser una función anónima:

```
window.addEventListener("popstate", function() {

alert("No nos dejes por favor")

});
```

O una función clásica:

```
window.addEventListener("resize",nombreFuncion);
```

Del objeto **Window** a su vez dependen otros objetos con sus propios métodos y propiedades, entre los que destacamos los siguientes:



DOCUMENT guarda el contenido completo de la página web cargada en el navegador y nos permitirá por tanto acceder al **Modelo de Objeto del Documento (DOM)**.

Antes de proceder a acceder a los elementos del DOM debemos comprobar que estos efectivamente se han cargado, para lo que usamos el evento **load** del objeto **Window**:

```
window.onload=function(){

//Acciones para manipular el DOM

}
```

Una vez cargado podemos acceder al contenido HTML de cualquier elemento usando su propiedad **innerHTML**, que nos permite leer el contenido existente usando el ID del elemento HTML:

```
let contenido=idDelElemento.innerHTML;
```

Y como es una propiedad tanto de lectura como de escritura, podemos igualmente cambiar su valor:

```
idDelElemento.innerHTML="Nuevo contenido para el elemento";
```

La propiedad **innerHTML** incluye el contenido HTML del elemento por lo que podemos incluir también etiquetas HTML:

```
idDelElemento.innerHTML="<b>Esto estará en negrita</b>";
```

Mientras que, si lo que queremos es acceder únicamente al nodo texto del elemento, podemos usar la propiedad **innerText**:

```
const ContenidoTexto=idDelElemento.innerText;
```

El objeto **Document** almacena las imágenes, formularios, etc. Por ejemplo, podemos acceder al valor de un campo de formulario:

document.nombreFormulario.nombreCampo.value;

O bien, usar el array FORMS del objeto **Document** para acceder por orden (empezando por 0 como todos los arrays) a los formularios del documento:

document.forms[0].nombreCampo.value;

En el caso de SELECT, como está formado por un array de opciones, debemos acceder al índice seleccionado:

document.forms[0].nobreSelect.options[document.forms[0].nombreSelect.selectedIndex].value

Para las imágenes podemos acceder a su propiedad SRC tanto para leerla como para modificarla. Para ello podemos acceder igualmente al array que contiene todas las imágenes del documento:

document.images[0].src

O bien directamente a través del ID del elemento:

idImagen.src

Dada la importancia del objeto **Document** para la manipulación del **DOM** lo estudiaremos más detalladamente en el tema dedicado al **DOM**.

NAVIGATOR almacena la información del navegador, por lo que podemos acceder a todas sus características:

- *userAgent* es la información completa sobre el navegador, incluyendo nombre, versión, motores, sistema operativo sobre el que se ejecuta, etc.
- *appCodeName* (propiedad obsoleta) es el nombre del Navegador, *appVersion* la versión del navegador y *platform* la información del sistema operativo
- *language* es el idioma en el que está configurado el navegador
- maxTouchPoints es el número máximo de puntos de contacto simultáneos que puede gestionar el navegador cuando se ejecuta en pantallas táctiles. SI el valor es 0 la pantalla no es táctil
- online tienen valor *booleano* que indica si tiene conexión o no.
- deviceMemory indica la memoria del dispositivo
- *connection* tipo de conexión (Wifi, 5g, etc.)
- *cookieEnabled*, valor *booleano* que nos indica si están habilitadas las *cookies*
- *mimeTypes* guarda información sobre los tipos MIME que soporta el navegador
- *online* valor booleano que indica si el navegador está conectado a internet
- *geolocation* almacena la información de geolocalización, lo que nos permitirá mediante la API de geolocalización acceder a estos datos.
- *mediaDevices* almacena información sobre los dispositivos, lo que permite a través de la API correspondiente acceder a ellos
- serviceWorker permite el acceso a la API para el trabajo fuera de línea y notificaciones push

Algunas de estas propiedades tienen asociado un evento que nos permite detectar si ha habido alguna modificación, por ejemplo, **online** y **offline**:

```
window.addEventListener('online', funcionQueSea);
```

FRAME es un array que almacena varios objetos Window cuando en el navegador hay cargados varios marcos. A través de este objeto podemos acceder a los diferentes marcos por ejemplo frames[0] corresponde con el objeto Window del primer marco. Si existen diferentes marcos (lo cual podemos comprobar por la extensión del array **frames.length**) podemos acceder a los demás por el índice del array o por sus identificador, por ejemplo **frames.miMarco** y a partir de ahí a las propiedades y métodos de ese **Window** concreto.

Como no todos los navegadores pueden acceder mediante la colección frames, es más seguro hacerlo por identificador.

Podemos acceder al objeto window del iframe a través de la propiedad **contentWindow**:

```
let ventanaIframe = document.getElementById("miFrame").contentWindow;
```

De esta forma, *ventanaIframe* equivale al objeto window del iframe, por lo que podremos acceder a sus métodos, como alert() o sus propiedades como innerHeight o location:

```
ventanaIframe.location="https://www.cdmfp.es"
```

Igualmente podemos ejecutar funciones javascript que estén definidas en el iframe:

```
ventanaIframe.miFuncion()
```

O acceder a sus eventos:

```
ventanaIframe.onclick=function(){

}
```

Igualmente podemos acceder al document del iframe a través de la propiedad **contentDocument**.

En HTML el frame se crea así: <iframe id="miFrame" src="casilla.html"></iframe>

LOCATION almacena la información sobre la URL que se ejecuta en el navegador, así **window.location** es la URL actual. Al ser una propiedad tanto de lectura como escritura, podemos modificarla, por lo que podemos cargar una URL distinta, por ejemplo **window.location="https://www.cdmfp.es"**. En ambos casos es equivalente a usar la propiedad **href** del objeto **location**: **window.location.href="https://www.cdmfp.es"**

Las URL se componen de varias partes. La primera parte es el protocolo de acceso, así una URL típica a una página web utiliza el protocolo HTTP o HTTPS (servidor seguro). La segunda parte de la URL indica el dominio en el que se encuentra el recurso. Tras el dominio, podemos además tener una tercera parte de la URL, ya que después podemos encontrar el nombre del recurso o los nombres de directorios dentro del dominio. Cada vez que tras el dominio encontramos un nombre separado por la barra estamos accediendo a un directorio nuevo dentro del dominio. Y después podemos encontrar el recurso en sí, identificado por un nombre y una extensión de fichero. Finalmente pueden encontrarse marcadores o anclas, es decir enlaces a identificadores dentro del documento identificados con # y también parámetros enviados a través de la URL a partir de ?

El objeto **location** almacena como hemos visto la URL completa, pero también podemos acceder a las diferentes partes de la misma, así **location.host** o **location.hostname** nos devuelve la parte principal de la url, es decir el dominio. Por su parte, **location.pathname** nos devuelve el resto de la URL que no es host. E igualmente podemos acceder al puerto (**location.port**) y al protocolo (**location.protocol**). La propiedad **location.hash** nos devolvería el ancla o marcador del enlace. Y por último, **location.search** nos permite recuperar la parte de la URL que incluye parámetros, es decir, valores que se envían al servidor para que los procese.

Pero además de estas propiedades, **location**, cuenta con algunos métodos, por ejemplo **location.assign()** nos permite asignar una nueva URL, lo que equivale a cambiar la dirección del navegador de la misma forma que si usáramos directamente **location.href**; y el método **replace()** que carga igualmente una nueva página sustituyendo a la que existía, pero de forma que la primera desaparece del historial y no es posible volver a ella usando los métodos de volver del navegador.

Otro método de **location** es **reload()** que podemos usar recargar de nuevo la página que ya está cargada.

A su vez, desde Javascript podemos escuchar los eventos de cambio de valores de **location**, con eventos del objeto Window que ya hemos visto, así, el evento **popstate** se desencadena cuando se usan los botones de atrás o adelante del navegador, mientras que **hashchange** se desencadena cuando hay cambio de hash en la URL, por ejemplo, para el trabajo con aplicaciones de una sola página (SPA).

HISTORY almacena el historial de navegación, lo que nos permite acceder a los documentos previos y posteriores (si los hubiera), así podemos consultar el valor de la propiedad ***length*** para saber cuántos documentos almacena ***history.length*** y el método ***history.back()*** nos permitirá retroceder al documento anterior, mientras **history.forward()** nos permitirá avanzar. El método **go()** nos permite movernos hacia adelante (con valores positivos) o hacia atrás (con valores negativos) en el historial de navegación del navegador.

El método ***history.pushState(objeto_estado, titulo, recurso)*** permite añadir elementos al historial de navegación pero sin recargar la página lo que es utilizado en aplicaciones de una sola página (SPA) en las que el nuevo contenido se carga en la misma página asíncronamente, simulando así la carga de diferentes páginas, y permitiendo a su vez acceder a ellas mediante el historial de navegación. Por su parte, ***history.replaceState()*** reemplaza un elemento del historial, por ejemplo esto es algo que hacen algunas páginas para evitar que al volver atrás regreses a la página de referencia y en su lugar te llevan a su página principal, por ejemplo.

SCREEN almacena información sobre la pantalla en la que se visualiza, así podemos saber sus dimensiones, resolución etc., accediendo a estas propiedades directamente:

- **width**: ancho de la pantalla en píxeles.
- **height**: alto de la pantalla en píxeles.
- **availWidth**: ancho disponible de la pantalla en píxeles (es decir, espacio para el contenido, sin barras).
- **availHeight**: altura disponible de la pantalla en píxeles (sin incluir la barra de tareas).
- **colorDepth**: profundidad de color en bits de la pantalla, es decir cuántos bits puede representar en cada píxel. Por ejemplo, con 24 bits se pueden representar más de 16 millones de colores
- **pixelDepth**: También devuelve la profundidad de color en bits, por lo que suele coincidir con el valor de **colorDepth** aunque puede ser mayor en algunos tipos de pantallas

Screen tiene también algunos métodos para trabajar con pantallas de dispositivos móviles, por ejemplo, los siguientes:

screen.orientation.lock(): se utiliza para bloquear la orientación de las pantallas, forzando que el contenido se vea en una determinada orientación, horizontal o vertical.

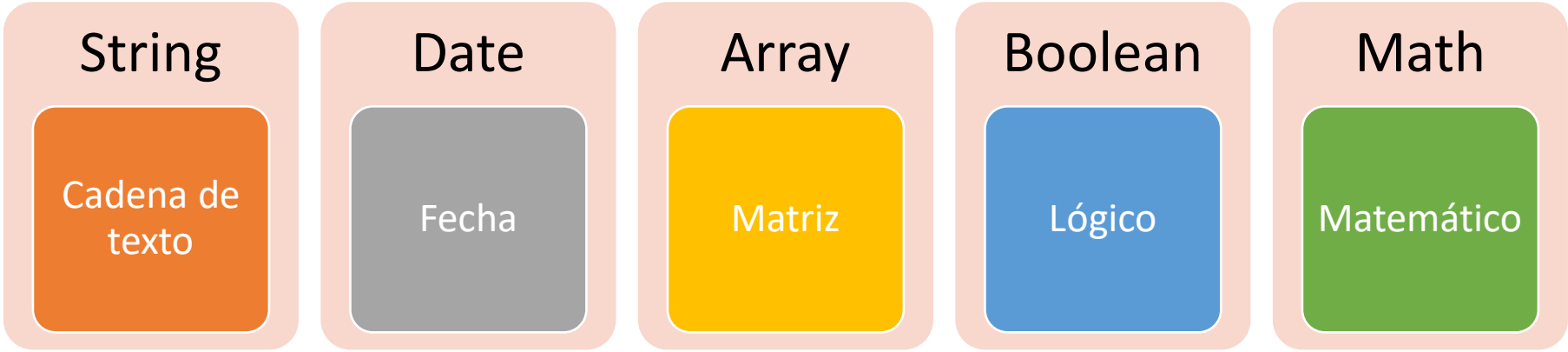
screen.orientation.unlock(): Desbloquea la orientación de la pantalla

Este método y el siguiente devolverán error en dispositivos de escritorio, y en dispositivos móviles solo funcionará a pantalla completa.

screen.orientation: es un objeto que contiene la información sobre la orientación de la pantalla, con propiedades como **angle y type** y el evento **change**:

```
screen.orientation.addEventListener('change', function() {

console.log('La orientación ha cambiado a:', screen.orientation.type); });
```

Pero además del objeto **Window**, disponemos de otros muchos objetos nativos o predefinidos disponibles en el entorno de ejecución, y que sirven como prototipos para crear nuevos objetos. Destacamos los siguientes:



Además de estos objetos básicos y del objeto principal, Window, existen otros objetos nativos como RegExp, Object, Function, Error, JSON, Map, Promise, Set o Symbol, que iremos viendo más adelante.

STRING. Cuando creamos una variable de tipo **string**, es decir, una cadena de texto, podemos hacerlo directamente asignando un valor entre comillas, por ejemplo:

```
texto="Hola, mundo";
```

Cualquier variable de tipo *string* hereda los métodos y propiedades del objeto ***String*** y por tanto podemos ejecutar esas acciones sobre cualquier cadena de texto.

La forma de ejecutar un método o acceder a una propiedad de un objeto es usar el nombre del objeto seguido de un punto y el nombre del método o propiedad, por ejemplo:

```
texto.length;
```

Esta propiedad devuelve por tanto un valor del objeto, la extensión en caracteres del **string**, en el caso de nuestro ejemplo: 11

Los métodos son como las funciones en el sentido de que al invocarlos también podemos pasar argumentos a ese método, por tanto, el uso de los métodos de un objeto se caracterizará porque incluye paréntesis en los que podemos o no añadir argumentos, por ejemplo:

texto.indexOf("Hola",0);

En este caso, el método **indexOf** recibe como parámetro una cadena de texto y un índice (opcional) y devuelve la posición en la que la cadena de texto pasada como parámetro se da por primera vez en el objeto a partir de la posición indicada por el índice, así **texto.indexOf("o")** devolverá 1, porque la primera ocurrencia del carácter "o" en la cadena "Hola, mundo" es la segunda (que como las cadenas empiezan en 0 es la posición 1), ya que no le hemos pasado el argumento índice y por tanto empieza a buscar desde el principio, sin embargo, **texto.indexOf("o",3)** devolverá 10, porque la primera "o" que hay en el texto "Hola, mundo" buscando a partir de la posición 3, es precisamente la "o" de "mundo" que ocupa la posición 10.

Si no se encuentra la subcadena que buscamos, el método **indexOf** devuelve -1 .

En las cadenas de texto la posición inicial es 0

Las propiedades y métodos del objeto **String** devuelven valores, pero no realizan ningún cambio sobre la cadena original.

Podemos crear cadenas a partir de otros objetos con **toString()**: let a=12; a.toString() devolverá “12”.

Algunas propiedades y métodos del objeto String

length

- Devuelve la extensión de la cadena

indexOf(subcadena, indice_opcional) / lastIndexOf (subcadena)

- Devuelve la posición en la que se encuentra la subcadena en la cadena a partir de la posición pasada en el índice, si no se pasa índice se considera el valor por defecto 0. Si no encuentra resultado devuelve -1.

lastIndexOf

- Igual que indexOf pero empezando a buscar por el final de la cadena por lo que devolverá la primera ocurrencia desde el final (pero, cuidado, la posición se empieza a contar desde el principio igualmente).

CharAt(indice)

- Devuelve el caracter que se encuentra en la posición indicada por el índice.

CharCodeAt(indice)

- Devuelve el código UNICODE del carácter que se encuentra en la posición indicada.

fromCharCode(código)

- Devuelve el carácter correspondiente al código. Como lo que se parametriza es el código UNICODE, se utiliza el objeto String, no una cadena concreta: String.fromCharCode(109) devolverá el carácter "m".

padStar / padEnd(longitud, subcadena)

- Devuelve una nueva cadena completada al principio o al fin con la repetición de la subcadena indicada hasta completar la longitud deseada.

trim()

- Elimina espacios vacios al principio y final de la cadena.

startsWith(subcadena) / endsWidth(subcadena) / includes(subcadena)

- Devuelven true si la cadena tiene al principio, al final o incluye respectivamente la subcadena indicada.

toLowerCase()

- Devuelve la cadena en minúsculas.

toUpperCase()

- Devuelve la cadena en mayúsculas.

concat(subcadena1, subcadena2)

- Devuelve una cadena con la cadena original a la que se concatenan la subcadena o subcadenas indicadas.

substr(inicio,longitud_opcional)

- Devuelve una subcadena a partir del índice indicado como inicio y con la longitud indicada o hasta el final si no se indica longitud. Si usamos un índice negativo, empieza a contar desde el final de la cadena, por ejemplo substr(-2) cogería los dos últimos caracteres de la cadena. Similar a usar slice().

substring(inicio,fin)

- Devuelve una subcadena a partir de la posición indicada en inicio hasta la posición indicada en fin, o hasta el final de la cadena si no se indica fin.

search(subcadena)

- Devuelve la primera posición en la que se encuentra la subcadena o -1 si no se encuentra. Permite expresiones regulares. No confundir con match(), que también permite expresiones regulares, pero lo que devuelve es un array formado por las coincidencias.

replace(subcadenaorignal,subcadenanueva)

- Reemplaza en la cadena el texto de la subcadenaoriginal por el de la subcadenanueva.

split(separador,tamaño_opcional)

- Crea un array a partir de la cadena usando como elemento separador el indicado como parámetro. Si se indica tamaño, se obtendrá un array de este tamaño (número de elementos totales) , no añadiendo más elementos aunque los hubiera en la cadena original.

DATE. El objeto date sirve para almacenar información de fechas. Al contrario que en el caso anterior en el que cualquier variable de tipo cadena hereda propiedades y métodos del objeto String, en el caso de las fechas es necesario crear un nuevo objeto a partir del objeto Date, para lo cual se utiliza el método **new()**, así:

miFecha=new Date()

De esta forma miFecha se convierte en un nuevo objeto **Date** con la fecha indicada como parámetro o con la fecha actual, si no se indican parámetros, y a partir de ese momento podemos utilizar los métodos del objeto Date que el nuevo objeto ha heredado.

Es importante tener en cuenta que el formato de fecha que utiliza Javascript por defecto es el anglosajón, por tanto, el formato fecha sería "aaaa-mm-dd" y también que el primer día de la semana es el domingo. Igualmente, en Javascript se toma como referencia la fecha del 1 de enero de 1970, por lo que todas las fechas son en realidad los milisegundos transcurridos desde ese momento. El método **now()** nos devolverá precisamente el número de milisegundos desde entonces a este momento **Date.now()**

Podemos usar el método **toLocaleDateString()** para convertir una fecha al formato español (o de otro país), pasando como parámetro el código de país (en el caso de España “ES-es”) y como segundo parámetro un objeto que indique el formato que queremos recibir, por ejemplo **{weekday: 'long'}** para los días de la semana.

Por ejemplo:

```
let hoy=new Date():

console.log(hoy.toLocaleDateString("ES-es",{weekday:'long'}))
```

Principales métodos del objeto Date:

getFullYear()

- Devuelve el año.

getMonth()

- Devuelve el número correspondiente al mes. pero empezando a contar desde 0 (que sería el mes enero).

getDate()

- Devuelve el día del mes.

getDay()

- Devuelve el día de la semana en número, empezando desde 0 (Domingo)

getHours()

- Devuelve la hora empezando por 0

getMinutes()

- Devuelve los minutos

getSeconds()

- Devuelve los segundos

getTime()

- Devuelve la fecha en milisengundos (desde las 0 horas del 1 de enero de 1970)

Todos estos métodos (salvo **getDay()** pues el día de la semana no puede cambiarse) tienen el equivalente con **set** en vez de **get** en el nombre del método, por ejemplo **setFullYear()** pasando como parámetro un nuevo valor que sustituye al de la fecha, por ejemplo **miFecha.setFullYear(2050)**;

ARRAY. Los arrays o **matrices** (también llamados ***arreglos***) son objetos que permiten almacenar un conjunto de valores en una única variable, por tanto, en un array no encontramos un único valor, sino varios valores a los que podemos acceder a través del índice del array (teniendo en cuenta que siempre se empieza en 0) expresado entre corchetes, por ejemplo:

miArray[0]

La forma de crear un Array es asignando a una variable varios valores separados por comas entre corchetes, por ejemplo:

Let miArray=[1,2,3,4,5,6,7,8,9,0]

También podemos crear un nuevo array creando un nuevo objeto Array usando el constructor (de forma similar a como creamos un nuevo objeto Date), sin pasarle parámetros, en cuyo caso estará vacío, pasándole como parámetro su extensión, o pasándole como parámetro los elementos que lo forman:

Let miArray= new Array(1,2,3,4,5,6,7,8,9,0)

Hay otras formas de crear arrays, por ejemplo a partir de una cadena de texto con el método **split()**

Una vez creado un array con cualquiera de estos métodos, el nuevo array hereda los métodos y propiedades del objeto Array:

length

- Devuelve la extensión del array, es decir, el número de elementos que lo componen.

concat(Array1,Array2)

- Concatena los arrays que se pasan como parámetro.

pop()

- Elimina y devuelve el último elemento del array.

shift()

- Elimina y devuelve el primer elemento del array.

push(elementos)

- Añade al final del array los elementos que se pasan como parámetro.

unshift(elementos)

- Añade al principio del array los elementos que se pasan como parámetro.

reverse()

- Invierte el orden de los elementos del array.

sort()

- Ordena alfabéticamente los elementos de array.

slice(inicio,fin)

- Devuelve los elementos del array desde la posición expresada como inicio (recuerda que empieza en 0) hasta el elemento anterior al pasado como parámetro fin o hasta el final, si no se pasa este parámetro.

join(separador_opcional)

- Convierte el array en una cadena de texto en el que los elementos aparecen separados por el carácter pasado como parámetro (si no se pasa separador, se usa el valor por defecto, que es la coma).

splice(inicio,cantidad,elemento_opcional)

- Elimina del array a partir del valor inicio -si se omite es cero-, la cantidad de elementos indicados. Si se añade como tercer parámetro un elemento, este sustituye al eliminado

indexOf (elemento, índice)

- De la misma forma que usamos este método con string, podemos usar los métodos indexOf y lastIndexOf para localizar un elemento en el array desde el principio o a partir de un índice: miArray(elemento, indice) y devolverá el índice si lo encuentra y -1 si no lo encuentra.

includes(elemento)

- Devolverá true si el array contiene el elemento y false si no lo contiene.

Para recorrer los arrays contamos con varios métodos, (además de usar un sencillo bucle **for**)

El método **forEach** permite recorrer todos sus elementos recuperando tanto el índice como el valor guardado en cada índice. Para ello indicamos como argumentos las acciones a realizar mediante una función que utiliza a su vez dos parámetros donde se almacenaran el valor y el índice (en ese orden, ya que si solo se incluye un parámetro es el del valor) de cada elemento:

```
miArray.forEach(function(valor,indice){
console.log(indice+": "+valor)
});
```

Igualmente podemos usarlo con una función flecha:

```
miArray.forEach((valor,indice)=>console.log(indice,valor))
```

for in. Es un método que permite recorre un array u otro objeto para obtener el índice de cada elemento almacenándolo en una variable. Aunque solo obtiene el índice se puede después acceder a los valores usando el índice:

```
for (let indice in miArray){ console.log(miArray[indice])
```

for of. Es un método similar al anterior, pero en este caso la variable almacena no el índice sino el valor:

```
for (let valor of miArray){ console.log(valor])
```

Por su parte el método map() de array permite también iterar el array. El método recibe como parámetro el nombre de una función que recibe a su vez como parámetro cada uno de los elementos del array, así, por ejemplo, se puede usar para crear un nuevo array modificando los valores del array inicial:

```
miArray=[1,2,3,4,5]
miNuevoArray=miArray.map(miFuncion)
function miFuncion(elemento){
return elemento*2
}
```

O directamente con una función anónima:

```
miArray=[1,2,3,4,5]
miNuevoArray=miArray.map(function(elemento){return elemento*2})
```

BOOLEANO. Nos permite crear directamente objetos booleanos, para ello usamos le constructor **new Boolean()** y en los paréntesis incluimos la expresión a evaluar como booleana, por ejemplo:

miVariable=new Boolean(1>2)

Como parámetro también podemos incluir directamente los valores *true/false* en cuyo caso el objeto Booleano tomará ese valor.

Cualquier otro valor que incluyamos como parámetro en el constructor que no sea una expresión evaluable devolverá *true*, salvo si es 0 o una cadena vacía, que devolverá *false*.

El objeto booleano no tiene métodos propios.

MATH. El objeto **Math** consta de una serie de propiedades que son constantes matemáticas como el número **E** o el número **PI**. Así **Math.PI** devolverá 3.141592653589793 y **Math.E** devolverá 2.718281828459045.

Recuerda que las constantes se escriben en mayúsculas.

Otras constantes del objeto Math son **LN2** (el resultado del logaritmo neperiano de 2), **LN10** (resultado del logaritmo neperiano de 10), **SQRT2** (resultado de la raíz cuadrada de 2), etc.

Además de las propiedades, el objeto **Math** dispone de métodos que permiten realizar operaciones matemáticas más complejas que los que podemos realizar con los operadores aritméticos.

Entre estos métodos los más prácticos son generar un número aleatorio entre 0 y 1 con **math.random(),** o extraer el número mayor o menor de varios números con **max(num1,num2,...)** y **min(num1,num2,...)**, así como los métodos para redondear números eliminado decimales: **floor()** o devolviendo el número entero menor: **round()** o superior: **ceil()**.

Podemos así combinar varios de estos métodos para obtener un número aleatorio entre 0 y 100:

Math.ceil(Math.random()*100)

Para comprobar valores numéricos pueden usarse también las siguientes funciones globales que ya conocemos: **isNaN(numero)**: comprueba si el parámetro no es un número (devuelve **false** si es un número); **parseInt(cadena)**: convierte la cadena en un número entero; **parseFloat(cadena)**: convierte la cadena en un número de coma flotante. El objeto **Number** dispone también de esos mismos métodos, así como **isInteger**, **isFinity**, así como constantes como **MAX_VALUE** o **MIN_VALUE** (número mayor y menor posible en Javascript).

Algunos métodos del objeto Math:

abs()

- Devuelve el valor absoluto de un número.

acos() / asin()

- Devuelve el arco coseno y el arco seno de un número respectivamente.

sqrt()

- Devuelve la raíz cuadrada de un número.

tan() / atan()

- Devuelve la tangente o el arco tangente de un número respectivamente.

sin() / cos()

- Devuelve el seno y el coseno de un número respectivamente.

ceil()

- Devuelve el número entero superior más próximo a un número.

round()

- Devuelve el número entero más próximo a un número. Así 1.4 devolverá 1, pero 1.6 devolverá 2.

floor()

- Devuelve la parte entera de un número, pero redondeando hacia abajo, por lo que en número negativos devolvería el entero más bajo, por ejemplo Math.floor(-4.2) no devuelve -4, sino -5.

trunc()

- Elimina la parte decimal del número.

log()

- Devueve el logaritmo neperiano de un número.

max(numero1,numero2) / min (numero1,numero2)

- Devuelve el número mayor o el número menor de los números pasados como parámetros respectivamente.

pow(base, exponente)

- Devuelve el resultado de elevar la base al exponente.

random()

- Devuelve un número aleatorio entre 0 y 1.