DESARROLLO EN ENTORNO CLIENTE/SERVIDOR

UD5: ACCESO A Y OBJETOS DEL NAVEGADOR

JavaScript

UD5 - Acceso a Objetos del Navegador

1. EVALUACIÓN

El presente documento, junto con sus actividades, cubren los siguientes criterios de evaluación:

RESULTADOS DE APRENDIZAJE	CRITERIOS DE EVALUACIÓN
RA3. Escribe código, identificando y aplicando las funcionalidades aportadas por los objetos predefinidos del lenguaje.	a) Se han identificado los objetos predefinidos del lenguaje. b) Se han analizado los objetos referentes a las ventanas del navegador y los documentos web que contienen. f) Se han utilizado las características propias del lenguaje en documentos compuestos por varias ventanas y marcos. g) Se han utilizado mecanismos del navegador web para almacenar información y recuperar su contenido. h) Se ha depurado y documentado el código.

2. Objetos de más alto nivel en JavaScript.

Una página web, es un documento HTML que será interpretado por los navegadores de forma gráfica, pero que también va a permitir el acceso al código fuente de la misma.

El Modelo de Objetos del Documento (DOM), permite ver el mismo documento de otra manera, describiendo el contenido del documento como un conjunto de objetos, sobre los que un programa de Javascript puede interactuar.

Según el W3C, el Modelo de Objetos del Documento es una interfaz de programación de aplicaciones (API), para documentos válidos HTML y bien construidos XML. Define la estructura lógica de los documentos, y el modo en el que se acceden y se manipulan.

Ahora que ya has visto en la unidad anterior, los fundamentos de la programación, vamos a profundizar un poco más en lo que se refiere a los objetos, que podremos colocar en la mayoría de nuestros documentos.

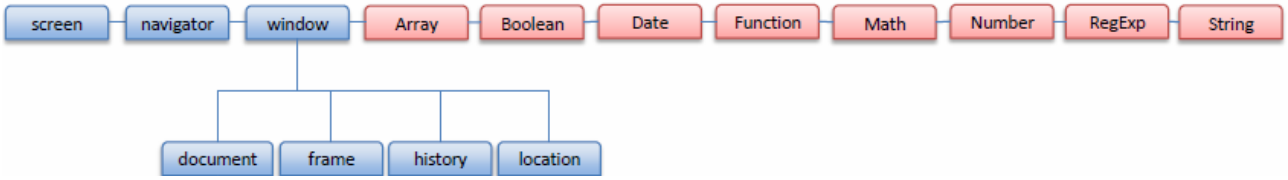
Definimos como objeto, una entidad con una serie de propiedades que definen su estado, y unos métodos (funciones), que actúan sobre esas propiedades. La forma de acceder a una propiedad de un objeto es la siguiente:

```
nombreobjeto.propiedad
```

La forma de acceder a un método de un objeto es la siguiente:

```
nombreobjeto.metodo( [parámetros opcionales] )
```

También podemos referenciar a una propiedad de un objeto, por su índice en la creación.



3. Objeto window

En la jerarquía de objetos, tenemos en la parte superior el objeto **window**. Este objeto está situado

justamente ahí, porque es el contenedor principal de todo el contenido que se visualiza en el navegador.

Tan pronto como se abre una ventana (window) en el navegador, incluso aunque no se cargue ningún documento en ella, este objeto window ya estará definido en memoria.

Además de la sección de contenido del objeto window, que es justamente dónde se cargarán los documentos, el campo de influencia de este objeto, abarca también las dimensiones de la ventana, así como todo lo que rodea al área de contenido: las barras de desplazamiento, barra de herramientas, barra de estado, etc.

El objeto **document** será el que contendrá toda la jerarquía de objetos, que tengamos dentro de nuestra página HTML.

Atención: en los navegadores más modernos, los usuarios tienen la posibilidad de abrir las páginas tanto en nuevas pestañas dentro de un navegador, como en nuevas ventanas de navegador. Para JavaScript tanto las ventanas de navegador, como las pestañas, son ambos objetos window.

Acceso a propiedades y métodos.

Para acceder a las propiedades y métodos del objeto window, lo podremos hacer de diferentes formas, dependiendo más de nuestro estilo, que de requerimientos sintácticos. Así, la forma más lógica y común de realizar esa referencia, incluiría el objeto window tal y como se muestra en este

ejemplo:

```
window.nombrePropiedad
window.nombreMétodo( [parámetros] )
```

Como puedes ver, los parámetros van entre corchetes, indicando que son opcionales y que dependerán del método al que estemos llamando.

Un objeto window también se podrá referenciar mediante la palabra self, cuando estamos haciendo la referencia desde el propio documento contenido en esa ventana:

```
self.nombrePropiedad
self.nombreMétodo( [parámetros] )
```

Podremos usar cualquiera de las dos referencias anteriores, pero intentaremos dejar la palabra reservada self, para scripts más complejos en los que tengamos múltiples marcos y ventanas.

Debido a que el objeto window siempre estará presente cuando ejecutemos nuestros scripts, podremos omitirlo, en referencias a los objetos dentro de esa ventana. Así que, si escribimos:

```
nombrePropiedad
nombreMétodo( [parámetros] )
```

También funcionaría sin ningún problema, porque se asume que esas propiedades o métodos, son del objeto de mayor jerarquía (el objeto window) en el cual nos encontramos.

3.1. Gestión de ventanas.

Un script no creará nunca la ventana principal de un navegador. Es el usuario, quien realiza esa tarea abriendo una URL en el navegador o un archivo desde el menú de abrir. Pero sin embargo, un script que esté ejecutándose en una de las ventanas principales del navegador, si que podrá crear o abrir nuevas subventanas.

El método que genera una nueva ventana es window.open(). Este método contiene hasta tres parámetros, que definen las características de la nueva ventana: la URL del documento a abrir, el nombre de esa ventana y su apariencia física (tamaño, color,etc.).

Por ejemplo, si consideramos la siguiente instrucción que abre una nueva ventana de un tamaño

determinado y con el contenido de un documento HTML:

```
let  subVentana=window.open("https://www.google.es","nueva","height=800,width=600");
```

Lo importante de esa instrucción, es la asignación que hemos hecho en la variable subVentana, de esta forma podremos a lo largo de nuestro código, referenciar a la nueva ventana desde el script original de la ventana principal. Por ejemplo, si quisiéramos cerrar la nueva ventana desde nuestro script, simplemente

tendríamos que hacer:

```
subVentana.close();
```

Aquí sí que es necesario especificar subVentana, ya que si escribiéramos window.close(), self.close() o close() estaríamos intentando cerrar nuestra propia ventana (previa confirmación), pero no la subVentana que creamos en los pasos anteriores. Véase el siguiente ejemplo que permite abrir y cerrar una sub-ventana:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>Apertura y Cierre de Ventanas</title>
<script type="text/javascript">
function inicializar(){
document.getElementById("crear-ventana").onclick=crearNueva;
document.getElementById("cerrar-ventana").onclick=cerrarNueva;
}
let nuevaVentana;
function crearNueva(){
nuevaVentana = window.open ("http://www.google.es","," height=400,width=800");
}
function cerrarNueva(){
if (nuevaVentana){
nuevaVentana.close();
nuevaVentana = null;
```

```
}
}
</script>
</head>
<body onLoad="inicializar()">
<h1>Abrimos y cerramos ventanas</h1>
<form>
<p>
<input type="button" id="crear-ventana" value="Crear Nueva Ventana">
<input type="button" id="cerrar-ventana" value="Cerrar Nueva Ventana">
</p>
</form>
</body>
</html>
```

3.1.1. Propiedades y métodos.

El objeto window representa una ventana abierta en un navegador. Si un documento contiene marcos (<frame> o <iframe>), el navegador crea un objeto window para el documento HTML, y un objeto window adicional para cada marco.

Propiedades del objeto Window	
Propiedad	**Descripción**
`closed`	Devuelve un valor `Boolean` indicando cuando una ventana ha sido cerrada o no.
`defaultStatus`	Ajusta o devuelve el valor por defecto de la barra de estado de una ventana.
`document`	Devuelve el objeto `document` para la ventana.
`frames`	Devuelve un array de todos los marcos (incluidos `iframes`) de la ventana actual.
`history`	Devuelve el objeto `history` de la ventana.
`length`	Devuelve el número de `frames` (incluyendo `iframes`) que hay en dentro de una ventana.
`location`	Devuelve la Localización del objeto ventana (URL del fichero).
`name`	Ajusta o devuelve el nombre de una ventana.
`navigator`	Devuelve el objeto `navigator` de una ventana.
`opener`	Devuelve la referencia a la ventana que abrió la ventana actual.
`parent`	Devuelve la ventana padre de la ventana actual.
`self`	Devuelve la ventana actual.
`status`	Ajusta el texto de la barra de estado de una ventana.

Métodos del objeto Window	
Método	**Descripción**
`alert()`	Muestra una ventana emergente de alerta y un botón de aceptar.
`blur()`	Elimina el foco de la ventana actual.
`clearInterval()`	Resetea el cronómetro ajustado con `setInterval()`.
`setInterval()`	Llama a una función o evalúa una expresión en un intervalo especificado (en milisegundos).
`close()`	Cierra la ventana actual.
`confirm()`	Muestra una ventana emergente con un mensaje, un botón de aceptar y un botón de cancelar.
`focus()`	Coloca el foco en la ventana actual.
`open()`	Abre una nueva ventana de navegación.
`prompt()`	Muestra una ventana de diálogo para introducir datos.
`parent`	Devuelve la ventana padre de la ventana actual.
`self`	Devuelve la ventana actual.
`status`	Ajusta el texto de la barra de estado de una ventana.

4. Objeto location.

El objeto location contiene información referente a la URL actual. Este objeto, es parte del objeto window y accedemos a él a través de la propiedad window.location. El objeto location contiene información referente a la URL actual.

Propiedades del objeto Location	
Propiedad	**Descripción**
`hash`	Cadena que contiene el nombre del enlace, dentro de la URL.
`host`	Cadena que contiene el nombre del servidor y el número del puerto, dentro de la URL.
`hostname`	Cadena que contiene el nombre de dominio del servidor (o la dirección IP), dentro de la URL.
`href`	Cadena que contiene la URL completa.
`pathname`	Cadena que contiene el camino al recurso, dentro de la URL.
`port`	Cadena que contiene el número de puerto del servidor, dentro de la URL.
`protocol`	Cadena que contiene el protocolo utilizado (incluyendo los dos puntos), dentro de la URL.
`search`	Cadena que contiene la información pasada en una llamada a un script, dentro de la URL.

Métodos del objeto Location	
Método	**Descripción**
`assign()`	Carga un nuevo documento.
`reload()`	Vuelve a cargar la URL especificada en la propiedad `href` del objeto `location`.
`replace()`	Reemplaza el historial actual mientras carga la URL especificada en cadenaURL.
`href`	Cadena que contiene la URL completa.
`pathname`	Cadena que contiene el camino al recurso, dentro de la URL.
`port`	Cadena que contiene el número de puerto del servidor, dentro de la URL.
`protocol`	Cadena que contiene el protocolo utilizado (incluyendo los dos puntos), dentro de la URL.
`search`	Cadena que contiene la información pasada en una llamada a un script, dentro de la URL.

5. Objeto navigator.

Este objeto navigator, contiene información sobre el navegador que estamos utilizando cuando abrimos una URL o un documento local.

Propiedades del objeto Navigator	
Propiedad	**Descripción**
`appCodeName`	Cadena que contiene el nombre en código del navegador.
`appName`	Cadena que contiene el nombre del cliente.
`appVersion`	Cadena que contiene información sobre la versión del cliente.
`cookieEnabled`	Determina si las cookies están o no habilitadas en el navegador.
`platform`	Cadena con la plataforma sobre la que se está ejecutando el programa cliente.
`userAgent`	Cadena que contiene la cabecera completa del agente enviada en una petición HTTP. Contiene la información de las propiedades `appCodeName` y `appVersion`.

Métodos del objeto Navigator	
Método	**Descripción**
javaEnabled()	Devuelve true si el cliente permite la utilización de Java, en caso contrario, devuelve false.

6. Objeto document.

Cada documento cargado en una ventana del navegador, será un objeto de tipo document. El objeto document proporciona a los scripts, el acceso a todos los elementos HTML dentro de una página.

Este objeto forma parte además del objeto window, y puede ser accedido a través de la propiedad window.document o directamente document (ya que podemos omitir la referencia a la window actual).

Colecciones del objeto Document	
Colección	**Descripción**
`anchors[]`	Es un array que contiene todos los hiperenlaces del documento.
`applets[]`	Es un array que contiene todos los applets del documento
`forms[]`	Es un array que contiene todos los formularios del documento.
`images[]`	Es un array que contiene todas las imágenes del documento.
`links[]`	Es un array que contiene todos los enlaces del documento.

Propiedades del objeto Document	
Propiedad	**Descripción**
`cookie`	Devuelve todos los nombres/valores de las cookies en el documento.
`domain`	Cadena que contiene el nombre de dominio del servidor que cargó el documento.
`lastModified`	Devuelve la fecha y hora de la última modificación del documento
`readyState`	Devuelve el estado de carga del documento actual
`referrer`	Cadena que contiene la URL del documento desde el cuál llegamos al documento actual.
`title`	Devuelve o ajusta el título del documento.
`URL`	Devuelve la URL completa del documento.

Propiedades del objeto Document	
Método	**Descripción**
`close()`	Cierra el flujo abierto previamente con document.open().
`getElementById()`	Para acceder a un elemento identificado por el id escrito entre paréntesis.
`getElementsByName()`	Para acceder a los elementos identificados por el atributo name escrito entre paréntesis.
`getElementsByTagName()`	Para acceder a los elementos identificados por el tag o la etiqueta escrita entre paréntesis.
`open()`	Abre el flujo de escritura para poder utilizar document.write() o document.writeln en el documento.
`write()`	Para poder escribir expresiones HTML o código de JavaScript dentro de un documento.
`writeln()`	Lo mismo que write() pero añade un salto de línea al final de cada instrucción.

7. Objetos y funciones globales

JavaScript tiene algunas funciones y objetos globales, las cuales pueden ser accedidas desde cualquier sitio. Estas funciones y objetos son bastante útiles para trabajar con números, cadenas, etc.

7.1. Funciones globales

- **parseInt(value)** → Transforma cualquier valor en un entero. Devuelve el valor entero, o NaN si no puede ser convertido.
- **parseFloat(value)** → Igual que parseInt, pero devuelve un decimal.
- **IsNaN(value)** → Devuelve true si el valor es NaN.
- **isFinite(value)** → Devuelve true si el valor es un número fnito o false si es infnito.
- **Number(value)** → Transforma un valor en un número (o NaN).
- **String(value)** → Convierte un valor en un string (en objetos llama a toString()).
- **encodeURI(string), decodeURI(string)** → Transforma una cadena en una URL codificada, codificando caracteres especiales a excepción de: , / ? : @ & = + $ #. Usa decodeURI para transformarlo a string otra vez.
  - Decoded: “http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: “http://domain.com?val=1%202%203&val2=r+y%256”
- **encodeURIComponent(string), decodeURIComponent(string)** → Estas funciones también codifcan y decodifcan los caracteres especiales que encodeURI no hace. Se deben usar para codifcar elementos de una url como valores de parámetros (no la url entera).
  - Decoded: “http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: “http%3A%2F%2Fdomain.com%3Fval%3D1%202%203%26val2%3Dr%2By%256”

7.2. El objeto Math

El objeto Math nos proporciona algunas constantes o métodos bastante útiles.

- Constants → E (Número de Euler), PI, LN2 (algoritmo natural en base 2), LN10, LOG2E (base-2 logaritmo de E), LOG10E, SQRT1_2 (raíz cuadrada de ½), SQRT2.
- round(x) → Redondea x al entero más cercano.
- foor(x) → Redondea x hacia abajo (5.99 → 5. Quita la parte decimal)
- ceil(x) → Redondea x hacia arriba (5.01 → 6)
- min(x1,x2,...) → Devuelve el número más bajo de los argumentos que se le pasan.
- max(x1,x2,...) → Devuelve el número más alto de los argumentos que se le pasan.
- pow(x, y) → Devuelve xy (x elevado a y).
- abs(x) → Devuelve el valor absoluto de x.
- random() → Devuelve un número decimal aleatorio entre 0 y 1 (no incluidos).
- cos(x) → Devuelve el coseno de x (en radianes).
- sin(x) → Devuelve el seno de x.
- tan(x) → Devuelve la tangente de x.
- sqrt(x) → Devuelve la raíz cuadrada de x

```
console.log("Raíz cuadrada de 9: " + Math.sqrt(9));
console.log("El valor de PI es: " + Math.PI);
console.log(Math.round(4.546342));
// Número aleatorio entre 1 y 10
console.log(Math.foor(Math.random() * 10) + 1);
```

7.3. Fechas

En Javascript tenemos la clase Date, que encapsula información sobre fechas y métodos para operar, permitiéndonos almacenar la fecha y hora local (timezone).

```
let date = new Date(); // Crea objeto Date almacena la fecha actual
console.log(typeof date); // Imprime object
console.log(date instanceof Date); // Imprime true
console.log(date); // Imprime fecha actual
```

Podemos enviarle al constructor el número de milisegundos desde el 1/171970 a las 00:00:00 GMT (Llamado Epoch o UNIX time). Si pasamos más de un número, (sólo el primero y el segundo son obligatorios),el orden debería ser: 1to → año, 2o → mes (0..11), 3o → día, 4o → hora, 5o → minuto, 6o → segundo. Otra opción es pasar un string que contenga la fecha en un formato válido. let date = new Date(1363754739620); // Nueva fecha 20/03/2013 05:45:39 (milisegundos desde Epoch)

```
let date2 = new Date(2015, 5, 17, 12, 30, 50); // 17/06/2015 12:30:50 (Mes empieza en 0 -> Ene, ...
11 -> Dic)
let date3 = new Date("2015-03-25"); // Formato de fecha largo sin la hora YYYY-MM-DD (00:00:00)
let date4 = new Date("2015-03-25T12:00:00"); // Formato fecha largo con la fecha
let date5 = new Date("03/25/2015"); // Formato corto MM/DD/YYYY
let date6 = new Date("25 Mar 2015"); // Formato corto con el mes en texto (March también válido)
let date7 = new Date("Wed Mar 25 2015 09:56:24 GMT+0100 (CET)"); // Formato completo con el timezone
```

Si, en lugar de un objeto Date, queremos directamente obtener los milisegundos que han pasado desde el 1/1/1970 (Epoch), lo que tenemos que hacer es usar los métodos Date.parse(string) y Date.UTC(año, mes, día, hora, minuto, segundos). También podemos usar Date.now(), para la fecha y hora actual en milisegundos:

```
let nowMs = Date.now(); // Momento actual en ms
let dateMs = Date.parse("25 Mar 2015"); // 25 Marzo 2015 en ms
let dateMs2 = Date.UTC(2015, 2, 25); // 25 Marzo 2015 en ms
```

La clase Date tiene setters y getters para las propiedades: fullYear, month (0-11), date (día), hours, minutes, seconds, y milliseconds. Si pasamos un valor negativo, por ejemplo, mes = -1, se establece el último mes (dic.) del año anterior.

```
// Crea un objeto fecha de hace 2 horas
let twoHoursAgo = new Date(Date.now() - (1000*60*60*2)); // (Ahora - 2 horas) en ms
// Ahora hacemos lo mismo,pero usando el método setHours
let now = new Date();
now.setHours(now.getHours() - 2);
```

Cuando queremos imprimir la fecha, tenemos métodos que nos la devuelven en diferentes formatos:

```
let now = new Date();
```

```
console.log(now.toString());
console.log(now.toISOString()); // Imprime 2016-06-26T18:00:31.246Z
console.log(now.toUTCString()); // Imprime Sun, 26 Jun 2016 18:02:48 GMT
console.log(now.toDateString()); // Imprime Sun Jun 26 2016
console.log(now.toLocaleDateString()); // Imprime 26/6/2016
console.log(now.toTimeString()); // Imprime 20:00:31 GMT+0200 (CEST)
console.log(now.toLocaleTimeString()); // Imprime 20:00:31
```

8. Expresiones regulares

Las expresiones regulares nos permiten buscar patrones en un string, por tanto buscar el trozo que coincida o cumpla dicha expresión. En JavaScript, podemos crear una expresión regular instanciando un objeto de la clase RegExp o escribiendo directamente la expresión entre dos barras '/'.

En este apartado veremos conceptos básicos sobre las expresiones regulares, podemos aprender más sobre ellas en el siguiente enlace. También, hay páginas web como esta que nos permiten probar expresiones regulares con cualquier texto.

Una expresión también puede tener uno o más modificadores como 'g' → Búsqueda global (Busca todas las coincidencias y no sólo la primera), ' i' → Caseinsensitive (No distingue entre mayúsculas y minúsculas), o 'm' → Búsqueda en más de una línea (Sólo tiene sentido en cadenas con saltos de línea). En Javascript puedes crear un objeto de expresión regular estos modificadores de dos formas:

```
let reg = new RegExp("^[0-9]{2}","gi");
let reg2 = /^[0-9]{2}/gi;
console.log(reg2 instanceof RegExp); // Imprime true
```

Estas dos formas son equivalentes, por tanto podéis elegir cual usar.

8.1. Expresiones regulares básicas

La forma más básica de una expresión regular es incluir sólo caracteres alfanuméricos (o cualquier otro que no sea un carácter especial). Esta expresión será buscada en el string, y devolverá true si encuentra ese substring.

```
let str = "Hello, I'm using regular expressions";

let reg = /reg/;

console.log(reg.test(str)); // Imprime true
```

En este caso, “reg” se encuentra en “Hello, I'm using reg ular expressions”, por lo tanto, el método test devuelve true.

8.2. Corchetes (opción entre caracteres)

Entre corchetes podemos incluir varios caracteres (o un rango), y se comprobará si el carácter en esa posición de la cadena coincide con alguno de esos.

[abc] → Algún carácter que sea ‘a’, ‘b’, ó ‘c’

[a-z] → Algún carácter entre ‘a’ y ‘z’ (en minúsculas)

[0-9] → Algún carácter entre 0 y 9 (carácter numérico)

[^ab0-9] → Algún caracter excepto (^) ‘a’, ‘b’ ó número.

Pipe → | (opción entre subexpresiones)

(exp1|exp2) → La coincidencia en la cadena será con la primera expresión o con la segunda. Podemos añadir tantos pipes como queramos.

8.3. Meta-caracteres

. (punto) → Un único carácter (cualquier carácter excepto salto de línea)

\w → Una palabra o carácter alfanumérico. \W → Lo opuesto a \w

\d → un número o dígito. \D → Lo opuesto a \d

\s → Carácter de espacio. \S → Lo opuesto a \s

\b → Delimitador de palabra. Coincide el principio o fnal de palabra (no un carácter). Por ejemplo, /\bcase\b/, coincide con “case” pero no “uppercase” o “casein”.

\n → Nueva línea

\t → Carácter de tabulación

8.4. Cuantificadores

+ → El carácter que le precede (o la expresión dentro de un paréntesis) se repite 1 o más veces.

* → Cero o más ocurrencias. Lo contrario a +, no tiene porqué aparecer

? → Cero o una ocurrencia. Se podría interpretar como que lo anterior es opcional.

{N} → Debe aparecer N veces seguidas.

{N,M} → De N a M veces seguidas.

{N,} → Al menos N veces seguidas.

^ → Principio de cadena.

$ → Final de la cadena.

Podemos encontrar más sobre las expresiones regulares de JavaScript en: https://www.w3schools.com/jsref/jsref_obj_regexp.asp

8.5. Ejemplos

/^[0-9]{8}[a-z]$/i → Comprueba si un string tiene un DNI. Esta expresión coincidirá con una cadena que comience (^) con 8 números, seguidos con una letra. No puede haber nada después ($ → fnal de cadena). El modificador 'i' hace que no se distinga entre mayúsculas y minúsculas.

/^\d{2}\/\d{2}\/(\d{2}|\d{4})$/ → Comprueba si una cadena contiene una fecha en el formato DD/MM/YY ó DD/MM/YYYY. El metacaracter \d es equivalente a usar [0-9], y el año no puede tener tres números, o son 2 o 4.

/\b[aeiou]\w*\b/i → Comprueba si hay alguna palabra en el string que comienza

por una vocal seguida de cero o más caracteres alfanuméricos (números, letras o '_').

8.6. Métodos para las expresiones regulares en JavaScript

A partir de un objeto RegExp, podemos usar dos métodos para comprobar si una cadena cumple una expresión regular. Los dos métodos son test y exec.

El método test() recibe una cadena e intenta de encontrar una la coincidencia con la expresión regular. Si el modificador global 'g' se especifica, cada vez que se llame al método se ejecutará desde la posición en la que se encontró la última coincidencia, por tanto podemos saber cuántas veces se encuentran coincidencias con esa expresión. Debemos recordar que si usamos el modificador 'i' no diferencia entre mayúsculas y minúsculas.

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime false, hay solo dos coincidencias
let reg2 = /am/gi; // "Am" será lo que busque ahora
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true. Ahora tenemos 3 coincidencias con este nuevo patrón
```

Si queremos más detalles sobre las coincidencias, podríamos usar el método exec(). Este método devuelve un objeto con los detalles cuando encuentra alguna coincidencia. Estos detalles incluyen el índice en el que empieza la coincidencia y también el string entero.

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Imprime ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime null. No hay más coincidencias
```

Una mejor forma de usarlo sería:

```
let str = "I am amazed in America";
let reg = /am/gi;
let match;
while(match = reg.exec(str)) {
console.log("Patrón encontrado!: " + match[0] + ", en la posición: " + match.index);
}
/* Esto imprimirá:
* Patrón encontrado!: am, en la posición: 2
* Patrón encontrado!: am, en la posición: 5
* Patrón encontrado!: Am, en la posición: 15 */
```

De forma similar, hay métodos de la clase String (sobre la cadena esta vez) que podemos usar, estos admiten expresiones regulares como parámetros (vamos, a la inversa). Estos métodos son match (Funciona de forma similar a exec) y replace.

El método match devuelve un array con todas las coincidencias encontradas en la cadena si ponemos el modifcador global → g, si no ponemos el modifcador global, se comporta igual que exec().

```
let str = "I am amazed in America";
console.log(str.match(/am/gi)); // Imprime ["am", "am", "Am"]
```

El método replace devuelve una nueva cadena con las coincidencias de la expresión regular reemplazadas por la cadena que le pasamos como segundo parámetro (si el modificador global no se especifica, sólo se modifica la primera coincidencia). Podemos enviar una función anónima que procese cada coincidencia encontrada y nos devuelva la cadena con los reemplazos correspondientes.

```
let str = "I am amazed in America";
console.log(str.replace(/am/gi, "xx")); // Imprime "I xx xxazed in xxerica"
console.log(str.replace(/am/gi, function(match) {
return "-" + match.toUpperCase() + "-";
})); // Imprime "I -AM- -AM-azed in -AM-erica
```

9. “Timers” (avisadores)

Hay dos tipos de “timers” que podemos crear en JavaScript para ejecutar algún trozo de código en el futuro (especifcado en milisegundos), timeouts e intervals. El primero se ejecuta sólo una vez (debemos volver a crearlo manualmente si queremos que se repita algo en el tiempo), y el segundo se repite cada X milisegundos sin parar (o hasta que sea cancelado).

- **timeout(función, milisegundos)** → Ejecuta una función pasados un número de milisegundos.

```
console.log(new Date().toString()); // Imprime inmediatamente la fecha actual
setTimeout(() => console.log(new Date().toString()), 5000); // Se ejecutará en 5 segundos
(5000 ms)
```

- **cleartimeout(timeoutId)** → Cancela un timeout (antes de ser llamado)

```
// setTimeout devuelve un número con el id, y a partir de ahí, podremos cancelarlo
let idTime = setTimeout(() => console.log(new Date().toString()), 5000);
clearTimeout(idTime); // Cancela el timeout (antes de que se ejecute)
```

- **setInterval(funcion, milisegundo)** → La diferencia con timeout es que cuando el tiempo acaba y se ejecuta la función, se resetea y se repite cada X milisegundos automáticamente hasta que nosotros lo cancelemos.

```
let num = 1;
setInterval(() => console.log(num++), 1000); // Imprime un número y lo incrementa cada
segundo
```

- **clearInterval(idIntervalo)** → Cancela un intervalo (no se repetirá más).

```
let num = 1;
let idInterval = setInterval(() => {
console.log(num++);
if(num > 10) { // Cuando imprimimos 10, paramos el timer para que no se repita más
clearInterval(idInterval);
}
}, 1000);
```

- **setInterval/setTimeOut(nombreFuncion, milisegundos, argumentos…)** → Podemos pasarle un nombre función existente. Si se requieren parámetros podemos establecerlos tras los milisegundos.

```
function multiply(num1, num2) {
console.log(num1 * num2);
}
setTimeout(multiply, 3000, 5, 7); // Después de 3 segundos imprimirá 35 (5*7)
```