UD05 - Objetos nativos y formularios

1. Objetos Nativos

1.1 Introducción

En este tema vamos a ver las funciones globales de Javascript (muchas de las cuales ya hemos visto como *Number()* o *String()*) y los objetos nativos que incorpora Javascript y que nos facilitarán el trabajo proporcionándonos métodos y propiedades útiles para no tener que "reinventar la rueda" en nuestras aplicaciones. Dentro de ellos está el objeto **RegExpr** que nos permite trabajar con **expresiones regulares** (son iguales que en otros lenguajes) que nos serán de gran ayuda, sobre todo a la hora de validar formularios y que por eso veremos en la siguiente sección.

1.2 Funciones globales

- `parseInt(valor)`: devuelve el valor pasado como parámetro convertido a entero o *NaN* si no es posible la conversión. Este método es mucho más permisivo que *Number* y convierte cualquier cosa que comience por un número (si encuentra un carácter no numérico detiene la conversión y devuelve lo convertido hasta el momento). Ejemplos:

```
console.log( parseInt(3.84) )            // imprime 3 (ignora los decimales)
console.log( parseInt('3.84') )          // imprime 3
console.log( parseInt('28manzanas') )    // imprime 28
console.log( parseInt('manzanas28') )    // imprime NaN
```

- `parseFloat(valor)`: igual pero devuelve un número decimal. Ejemplos:

```
console.log( parseFloat(3.84) )            // imprime 3.84
console.log( parseFloat('3.84') )          // imprime 3.84
console.log( parseFloat('3,84') )          // imprime 3 (la coma no es un carácter numérico)
console.log( parseFloat('28manzanas') )    // imprime 28
console.log( parseFloat('manzanas28') )    // imprime NaN
```

- `Number(valor)`: convierte el valor a un número. Es como *parseFloat* pero más estricto y si no puede convertir todo el valor devuelve *NaN*. Ejemplos:

```
console.log( Number(3.84) )            // imprime 3.84
console.log( Number('3.84') )          // imprime 3.84
console.log( Number('3,84') )          // imprime NaN (la coma no es un carácter numérico)
console.log( Number('28manzanas') )    // imprime NaN
console.log( Number('manzanas28') )    // imprime NaN
```

- `String(valor)`: convierte el valor pasado a una cadena de texto. Si le pasamos un objeto llama al método *.toString()* del objeto. Ejemplos:

```
console.log( String(3.84) )                // imprime '3.84'
console.log( String([24, 3. 12]) )         // imprime '24,3,12'
console.log( {nombre: 'Marta', edad: 26} )  // imprime "[object Object]"
```

- `isNaN(valor)`: devuelve *true* si lo pasado NO es un número o no puede convertirse en un número. Ejemplos:

```
console.log( isNaN(3.84) )              // imprime false
console.log( isNaN('3.84') )            // imprime false
console.log( isNaN('3,84') )            // imprime true (la coma no es un
carácter numérico)
console.log( isNaN('28manzanas') )      // imprime true
console.log( isNaN('manzanas28') )      // imprime true
```

- `isFinite(valor)`: devuelve *false* si es número pasado es infinito (o demasiado grande)

```
console.log( isFinite(3.84) )              // imprime true
console.log( isFinite(3.84 / 0) )              // imprime false
```

- `encodeURI(string)` / `decodeURI(string)`: transforma la cadena pasada a una URL codificada válida transformando los caracteres especiales que contenga, excepto , / ? : @ & = + $ #. Debemos usarla siempre que vayamos a pasar una URL. Ejemplo:
  - 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)`: transforma también los caracteres que no transforma la anterior. Debemos usarla para codificar parámetros pero no una URL entera. Ejemplo:
  - 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”

1.3 Objetos nativos del lenguaje

En Javascript casi todo son objetos. Ya hemos visto diferentes objetos:

- window
- screen
- navigator
- location
- history
- document

Los 5 primeros se corresponden al modelo de objetos del navegador y *document* se corresponde al modelo de objetos del documento. Todos nos permiten interactuar con el navegador para realizar distintas acciones.

Pero además tenemos los tipos de objetos nativos, que no dependen del navegador. Son:

- Number
- String
- Boolean

- Array
- Function
- Object
- Math
- Date
- RegExp

Además de los tipos primitivos de número, cadena, booleano, undefined y null, Javascript tiene todos los objetos indicados. Como vimos se puede crear un número usando su tipo primitivo (`let num = 5`) o su objeto (`let num = new Number(5)`) pero es mucho más eficiente usar los tipos primitivos. Pero aunque lo creemos usando el tipo de dato primitivo se considera un objeto y tenemos acceso a todas sus propiedades y métodos (`num.toFixed(2)`).

Ya hemos visto las principales propiedades y métodos de *Number*, *String*, Boolean y *Array* y aquí vamos a ver las del resto.

1.4 Objeto Math

Proporciona constantes y métodos para trabajar con valores numéricos:

- constantes: `.PI` (número pi), `.E` (número de Euler), `.LN2` (algoritmo natural en base 2), `.LN10` (logaritmo natural en base 10), `.LOG2E` (logaritmo de E en base 2), `.LOG10E` (logaritmo de E en base 10), `.SQRT2` (raíz cuadrada de 2), `.SQRT1_2` (raíz cuadrada de 1⁄2). Ejemplos:

```
console.log( Math.PI )            // imprime 3.141592653589793
console.log( Math.SQRT2 )         // imprime 1.4142135623730951
```

- `Math.round(x)`: redondea x al entero más cercano
- `Math.floor(x)`: redondea x hacia abajo (5.99 → 5. Quita la parte decimal)
- `Math.ceil(x)`: redondea x hacia arriba (5.01 → 6)
- `Math.min(x1,x2,...)`: devuelve el número más bajo de los argumentos que se le pasan.
- `Math.max(x1,x2,...)`: devuelve el número más alto de los argumentos que se le pasan.
- `Math.pow(x, y)`: devuelve x y (x elevado a y).
- `Math.abs(x)`: devuelve el valor absoluto de x.
- `Math.random()`: devuelve un número decimal aleatorio entre 0 (incluido) y 1 (no incluido). Si queremos un número entre otros rangos haremos `Math.random() * (max - min) + min` o si lo queremos sin decimales `Math.round(Math.random() * (max - min) + min)`
- `Math.cos(x)`: devuelve el coseno de x (en radianes).
- `Math.sin(x)`: devuelve el seno de x.
- `Math.tan(x)`: devuelve la tangente de x.
- `Math.sqrt(x)`: devuelve la raíz cuadrada de x

Ejemplos:

```
console.log( Math.round(3.14) )      // imprime 3
console.log( Math.round(3.84) )      // imprime 4
console.log( Math.floor(3.84) )      // imprime 3
console.log( Math.ceil(3.14) )       // imprime 4
console.log( Math.sqrt(2) )          // imprime 1.4142135623730951
```

1.5 Objeto Date

Es la clase que usaremos siempre que vayamos a trabajar con fechas. Al crear una instancia de la clase le pasamos la fecha que queremos crear o lo dejamos en blanco para que nos cree la fecha actual. Si le pasamos la fecha podemos pasarle:

- milisegundos, desde la fecha EPOCH
- cadena de fecha
- valor para año, mes (entre 0 y 11), día, hora, minutos, segundos, milisegundos

Ejemplos:

```
let date1=new Date()    // Mon Jul 30 2018 12:44:07 GMT+0200 (CEST) (es cuando he ejecutado la instrucción)
let date7=new Date(1532908800000)    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (miliseg. desde 1/1/1970)
let date2=new Date('2018-07-30')    // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST) (la fecha pasada a las 0h. GMT)
let date3=new Date('2018-07-30 05:30')  // Mon Jul 30 2018 05:30:00 GMT+0200 (CEST) (la fecha pasada a las 05:300h. local)
let date6=new Date('07-30-2018')    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (OJO: formato MM-DD-AAAA)
let date7=new Date('30-Jul-2018')    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (tb. podemos poner 'Julio')
let date4=new Date(2018,7,30)    // Thu Ago 30 2018 00:00:00 GMT+0200 (CEST) (OJO: 0->Ene,1->Feb... y a las 0h. local)
let date5=new Date(2018,7,30,5,30)    // Thu Ago 30 2018 05:30:00 GMT+0200 (CEST) (OJO: 0->Ene,1->Feb,...)
```

> **EJERCICIO 1_UD5**: Crea en la consola dos variables fecNac1 y fecNac2 que contengan tu fecha de nacimiento. La primera la creas pasandole una cadena y la segunda pasándole año, mes y día

Cuando ponemos la fecha como texto, como separador de las fechas podemos usar `-`, `/` o `espacio`. Como separador de las horas debemos usar `:`. Cuando ponemos la fecha cono parámetros numéricos (separados por `,`) podemos poner valores fuera de rango que se sumarán al valor anterior. Por ejemplo:

```
let date=new Date(2018,7,41)    // Mon Sep 10 2018 00:00:00 GMT+0200 (CEST) -> 41=31Ago+10
let date=new Date(2018,7,0)     // Tue Jul 31 2018 00:00:00 GMT+0200 (CEST) -> 0=0Ago=31Jul (el anterior)
let date=new Date(2018,7,-1)    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) -> -1=0Ago-1=31Jul-1=30Jul
```

OJO con el rango de los meses que empieza en 0->Ene, 1->Feb,...,11->Dic

Tenemos métodos ***getter*** y ***setter*** para obtener o cambiar los valores de una fecha:

- **fullYear**: permite ver (*get*) y cambiar (*set*) el año de la fecha:

```
let fecha=new Date('2018-07-30')    // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getFullYear() )  // imprime 2018
fecha.setFullYear(2019)             // Tue Jul 30 2019 02:00:00 GMT+0200 (CEST)
```

- **month**: devuelve/cambia el número de mes, pero recuerda que 0->Ene,...,11->Dic

```
let fecha=new Date('2018-07-30')    // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getMonth() )     // imprime 6
fecha.setMonth(8)                   // Mon Sep 30 2019 02:00:00 GMT+0200 (CEST)
```

- **date**: devuelve/cambia el número de día:

```
let fecha=new Date('2018-07-30')    // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )      // imprime 30
fecha.setDate(-2)                   // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
```

- **day**: devuelve el número de día de la semana (0->Dom, 1->Lun, ..., 6->Sáb). Este método NO tiene *setter*:

```
let fecha=new Date('2018-07-30')    // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDay() )       // imprime 1
```

- **hours**, **minutes**, **seconds**, **milliseconds**, : devuelve/cambia el número de la hora, minuto, segundo o milisegundo, respectivamente.
- **time**: devuelve/cambia el número de milisegundos desde Epoch (1/1/1970 00:00:00 GMT):

```
let fecha=new Date('2018-07-30')     // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getTime() )       // imprime 1532908800000
fecha.setTime(1000*60*60*24*25)      // Fri Jan 02 1970 01:00:00 GMT+0100 (CET) (le hemos añadido 25 días a Epoch)
```

EJERCICIO 2_UD5: Realiza en la consola los siguientes ejercicios (usa las variables que creaste antes)

- muestra el día de la semana en que naciste
- modifica fecNac1 para que contenga la fecha de tu cumpleaños de este año (cambia sólo el año)
- muestra el día de la semana de tu cumpleaños de este año
- calcula el nº de días que han pasado desde que naciste hasta hoy

Para mostrar la fecha tenemos varios métodos diferentes:

- **.toString()**: "Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)"
- **.toUTCString()**: "Mon, 30 Jul 2018 00:00:00 GMT"
- **.toDateString()**: "Mon, 30 Jul 2018"
- **.toTimeString()**: "02:00:00 GMT+0200 (hora de verano de Europa central)"
- **.toISOString()**: "2018-07-30T00:00:00.000Z"
- **.toLocaleString()**: "30/7/2018 2:00:00"
- **.toLocaleDateString()**: "30/7/2018"
- **.toLocaleTimeString()**: "2:00:00"

EJERCICIO 3_UD5: muestra en distintos formatos la fecha y la hora de hoy

NOTA: recuerda que las fechas son objetos y que se copian y se pasan como parámetro por referencia:

```
let fecha=new Date('2018-07-30')     // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
let otraFecha=fecha
otraFecha.setDate(28)                // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )       // imprime 28 porque fecha y otraFecha son el mismo objeto
```

Una forma sencilla de copiar una fecha es crear una nueva pasándole la que queremos copiar:

```
let fecha=new Date('2018-07-30')     // Mon Jul 30 2018 02:00:00 GMT+0200
(CEST)
let otraFecha=new Date(fecha)
otraFecha.setDate(28)                // Thu Jun 28 2018 02:00:00 GMT+0200
(CEST)
console.log( fecha.getDate() )       // imprime 30
```

En realidad lo que le estamos pasando es el tiempo Epoch de la fecha (es como hacer `otraFecha=new Date(fecha.getTime())`)

NOTA: la comparación entre fechas funciona correctamente con los operadores `>`, `>=`, `<` y `<=` pero NO con `==`, `===`, `!=` y `!==` ya que compara los objetos y ve que son objetos diferentes. Si queremos saber si 2 fechas son iguales (siendo diferentes objetos) el código que pondremos NO es `fecha1 === fecha2` sino `fecha1.getTime() === fecha2.getTime()`.

> **EJERCICIO 4_UD5**: Comprueba si es mayor tu fecha de nacimiento o el 1 de enero de este año

Podemos probar los distintos métodos de las fechas en la página de w3schools.

2. Formularios

2.1 Introducción

En este apartado vamos a ver cómo realizar una de las acciones principales de Javascript que es la validación de formularios en el lado cliente.

Se trata de una verificación útil porque evita enviar datos al servidor que sabemos que no son válidos pero **NUNCA** puede sustituir a la validación en el lado servidor ya que en el lado cliente se puede manipular el código desde la consola para que se salte las validaciones que le pongamos.

Podéis encontrar una guía muy completa de validación de formularios en el lado cliente el la página de MDN web docs que ha servido como base para estos apuntes.

Básicamente tenemos 2 maneras de validar un formulario en el lado cliente:

- usar la validación incorporada en HTML5 y dejar que sea el navegador quien se encargue de todo
- realizar nosotros la validación mediante Javascript

La ventaja de la primera opción es que no tenemos que escribir código sino simplemente poner unos atributos a los INPUT que indiquen qué se ha de validar. La principal desventaja es que no tenemos ningún control sobre el proceso, lo que provocará cosas como:

- el navegador valida campo a campo: cuando encuentra un error en un campo lo muestra y hasta que no se soluciona no valida el siguiente lo que hace que el proceso sea molesto apra el usuario que no ve todo lo que hay mal de una vez
- los mensajes son los predeterminados del navegador y en ocasiones pueden no ser muy claros para el usuario
- los mensajes se muestran en el idioma en que está configurado el navegador, no en el de nuestra página

Además, al final de este tema, veremos una pequeña introducción a las expresiones regulares en Javascript.

2.2 Validación incorporada en HTML5

Funciona añadiendo atributos a los campos del formulario que queremos validar. Los más usados son:

- **required**: indica que el campo es obligatorio. La valdación fallará si no hay nada escrito en el input. En el caso de un grupo de *radiobuttons* se pone sobre cualquiera de ellos (o sobre todos) y obliga a que haya seleccionada una opción cualquiera del grupo
- **pattern**: obliga a que el contenido del campo cumpla la expresión regular indicada. Por ejemplo para un código postal sería `pattern="^[0-9]{5}$"`
- **minlength** / **maxlength**: indica la longitud mínima/máxima del contenido del campo
- **min** / **max**: indica el valor mínimo/máximo del contenido de un campo numérico

También producen errores de validación si el contenido de un campo no se adapta al *type* indicado (email, number, ...) o si el valor de un campo numérico no cumple con el *step* indicado.

Cuando el contenido de un campo es valido dicho campo obtiene automáticamente la pseudoclase **:valid** y si no lo es tendrá la pseudoclase **:invalid** lo que nos permite poner reglas en nuestro CSS para destacar dichos campos, por ejemplo:

```
input:invalid {
  border: 2px dashed red;
}
```

La validación se realiza al enviar el formulario y al encontrar un error se muestra, se detiene la validación del resto de campos y no se envía el formulario.

2.3 Validación mediante API de validación de formularios

Mediante Javscript tenemos acceso a todos los campos del formulario por lo que podemos hacer la validación como queramos, pero es una tarea pesada, repetitiva y que provoca código spaguetti difícil de leer y mantener más adelante.

Para hacerla más simple podemos usar la API de validación de formularios de HTML5 que nos da la ventaja de:

- los requisitos de validación de cada campo están como atributos HTML de dicho campo por lo que son fáciles de ver
- no tenemos que comprobar nosotros si el contenido del campo cumple o no esos requisitos sino que lo comprueba el navegador y nosotros mediante la API sólo preguntamos si se cumplen o no
- automáticamente pone a los campos las clases *:valid* o *:invalid* por lo que no tenemos que añadirles clases para desacarlos

Esta API nos proporciona estas propiedades y métodos:

- **checkValidity()**: método que nos dice si el campo al que se aplica es o no válido. También se puede aplicar al formulario para saber si es válido o no
- **validationMessage**: en caso de que un campo no sea válido esta propiedad contiene el texto del error de validación proporcionado por el navegador
- **validity**: es un objeto que tiene propiedades booleanas para saber qué requisito del campo es el que falla:
  - **valueMissing**: indica si no se cumple la restricción *required* (es decir, valdrá *true* si el campo tiene el atributo *required* pero no se ha introducido nada en él)
  - **typeMismatch**: indica si el contenido del campo no cumple con su atributo *type* (ej. type="email")
  - **patternMismatch**: indica si no se cumple con el *pattern* indicado
  - **tooShort** / **tooLong**: indica si no se cumple el atributo *minlength*/*maxlength*
  - **rangeUnderflow** / **rangeOverflow**: indica si no se cumple el atributo *min* / *max*
  - **stepMismatch**: indica si no se cumple el atributo *step* del campo
  - **customError**: indica al campo se le ha puesto un error personalizado con *setCustomValidity*
  - **valid**: indica si es campo es válido
  - ...
- **setCustomValidity(mensaje)**: añade un error personalizado al campo (que ahora ya NO será válido) con el mensaje pasado como parámetro. Nos permite personalizar el mensaje del navegador. Para quitar este error se hace `setCustomValidity('')`

En la página de W3Schools podéis ver algún ejemplo básico de esto.

Para validar un formulario nosotros pero usando esta API debemos añadir al FORM el atributo **novalidate** que hace que no se encargue el navegador de mostrar los mensajes de error ni de decidir si se envía o no el formulario (aunque sí valida los campos) sino que lo haremos nosotros.

2.3.1 Ejemplo

Un ejemplo sencillo de validación de un formulario podría ser:

- index.html

```
<form novalidate>
  <p>
    <label for="nombre">
      <span>Por favor, introduzca su nombre (entre 5 y 50 caracteres):
</span>
      <input type="text" id="nombre" name="nombre" required minlength="5"
maxlength="50">
      <span class="error"></span>
    </label>
    <label for="mail">
      <span>Por favor, introduzca una dirección de correo electrónico:
</span>
      <input type="email" id="mail" name="mail" required minlength="8">
      <span class="error"></span>
    </label>
  </p>
  <button type="submit">Enviar</button>
</form>
```

- main.js

```
const form  = document.getElementsByTagName('form')[0];

const nombre = document.getElementById('nombre');
const nombreError = document.querySelector('#nombre + span.error');
const email = document.getElementById('mail');
const emailError = document.querySelector('#mail + span.error');

form.addEventListener('submit', (event) => {
  if(!form.checkValidity()) {
    event.preventDefault();
  }
  nombreError.textContent = nombre.validationMessage;
  emailError.textContent = email.validationMessage;
});
```

- style.css

```css
.error {
  color: red;
}

input:invalid {
  border: 2px dashed red;
}
```

Existen múltiples librerías que facilitan enormenmente el tedioso trabajo de validar un formulario. Un ejemplo es yup.

2.4 Expresiones regulares

Las expresiones regulares permiten buscar un patrón dado en una cadena de texto. Se usan mucho a la hora de validar formularios o para buscar y reemplazar texto. En Javascript se crean poniéndolas entre caracteres `/` (o instanciándolas de la clase *RegExp*, aunque es mejor de la otra forma):

```javascript
let cadena='Hola mundo';
let expr=/mundo/;
expr.test(cadena);      // devuelve true porque en la cadena se encuentra la expresión 'mundo'
```

2.4.1 Patrones

La potencia de las expresiones regulares es que podemos usar patrones para construir la expresión. Los más comunes son:

- **[..]** (corchetes): dentro se ponen varios caracteres o un rango y permiten comprobar si el carácter de esa posición de la cadena coincide con alguno de ellos. Ejemplos:
  - `[abc]`: cualquier carácter de los indicados ('a' o 'b' o 'c')
  - `[^abc]`: cualquiera excepto los indicados
  - `[a-z]`: cualquier minúscula (el carácter '-' indica el rango entre 'a' y 'z', incluidas)
  - `[a-zA-Z]`: cualquier letra
- **(|)** (*pipe*): debe coincidir con una de las opciones indicadas:
  - `(x|y)`: la letra x o la y (sería equivalente a `[xy]`
  - `(http|https)`: cualquiera de las 2 palabras
- **Metacaracteres**:
  - `.` (punto): un único carácter, sea el que sea
  - `\d`: un dígito (`\D`: cualquier cosa menos dígito)

    - `\s`: espacio en blanco (`\S`: lo opuesto)
    - `\w`: una palabra o carácter alfanumérico (`\W` lo contrario)
    - `\b`: delimitador de palabra (espacio, ppio, fin)
    - `\n`: nueva línea
- **Cuantificadores**:
    - `+`: al menos 1 vez (ej. `[0-9]+` al menos un dígito)
    - `*`: 0 o más veces
    - `?`: 0 o 1 vez
    - `{n}`: n caracteres (ej. `[0-9]{5}` = 5 dígitos)
    - `{n,}`: n o más caracteres
    - `{n,m}`: entre n y m caracteres
    - `^`: al ppio de la cadena (ej.: `^[a-zA-Z]` = empieza por letra)
    - `$`: al final de la cadena (ej.: `[0-9]$` = que acabe en dígito)
- **Modificadores**:
    - `/i`: que no distinga entre Maysc y minsc (Ej. `/html/i` = buscará html, Html, HTML, ...)
    - `/g`: búsqueda global, busca todas las coincidencias y no sólo la primera
    - `/m`: busca en más de 1 línea (para cadenas con saltos de línea)

> **EJERCICIO 5_UD5**: contruye una expresión regular para lo que se pide a continuación y pruébala con distintas cadenas:
>
> - un código postal
> - un NIF formado por 8 números, un guión y una letra mayúscula o minúscula
> - un número de teléfono y aceptamos 2 formatos: XXX XX XX XX o XXX XXX XXX. EL primer número debe ser un 6, un 7, un 8 o un 9

2.4.2 Métodos

Los usaremos para saber si la cadena coincide con determinada expresión o para buscar y reemplazar texto:

- `expr.test(cadena)`: devuelve **true** si la cadena coincide con la expresión. Con el modificador */g* hará que cada vez que se llama busque desde la posición de la última coincidencia. Ejemplo:

```
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;          // ahora no distinguirá mayúsculas y minúsculas
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
```

- `expr.exec(cadena)`: igual pero en vez de *true* o *false* devuelve un objeto con la coincidencia encontrada, su posición y la cadena completa:

```
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
```

- `cadena.match(expr)`: igual que *exec* pero se aplica a la cadena y se le pasa la expresión. Si ésta tiene el modificador */g* devolverá un array con todas las coincidencis:

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

- `cadena.search(expr)`: devuelve la posición donde se encuentra la coincidencia buscada o -1 si no aparece
- `cadena.replace(expr, cadena2)`: devuelve una nueva cadena con las coincidecias de la cadena reemplazadas por la cadena pasada como 2º parámetro:

```
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"
```

No vamos a profundizar más sobre las expresiones regulares. Es muy fácil encontrar por internet la que necesitemos (para validar un e-mail, un NIF, un CP, ...). Podemos aprender más en:

- w3schools
- regular-expressions.info
- html5pattern
- y muchas otras páginas

También, hay páginas que nos permiten probar expresiones regulares con cualquier texto, como regexr.

Bibliografía

- Curso 'Programación con JavaScript'. CEFIRE Xest. Arturo Bernal Mayordomo
- Curso de JavaScript y TypeScript de Arturo Bernal en Youtube
- MDN Web Docs. Moz://a. https://developer.mozilla.org/es/docs/Web/JavaScript
- Introducción a JavaScript. Librosweb. http://librosweb.es/libro/javascript/
- Curso de Javascript (Desarrollo web en entorno cliente). Ada Lovecode - Didacticode (90 vídeos)
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). García Barea, Sergi
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). Segura Vasco, Juan. CIPFP Batoi.