5. Expresiones regulares y objetos RegExp

```
var patron=/Aloe\s+Vera/;
```

Una expresión regular es un patrón que lo pueden cumplir muchas cadenas. En este ejemplo en las cadenas “Aloe Vera”, “Aloe Vera”, “Aloe Vera”, etc. se encuentra este patrón.

En JavaScript las expresiones regulares **se gestionan a través del objeto `RegExp`**.

Hay dos formas de definir un expresión regular:

- Mediante una **expresión literal**:

```
var  expresion = /patron/modificadores;
//No llevan dobles comillas
```

- Mediante el **objeto RegExp**:

```
var expresion= new RegExp(“/patron/”, modificadores)
```

5.1. Modificadores

Las expresiones regulares se componen de dos partes:

- **Patrón de la expresión**
- **Modificadores.**

Los modificadores son opcionales y sirven para definir el comportamiento de la expresión regular. Son los siguientes (y no están activados por defecto):

- **`g`(global)**: El patrón se aplicará a toda la cadena en lugar de detenerse al encontrar la primera correspondencia correcta. Es decir, se devolverán todas las ocurrencias encontradas. Tiene sentido con métodos `match()` y `replace()` de `String`, no con métodos `exec()` y `test()` de `RegExp`, ni con el método `search()` de `String`.

Ejemplo: https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_regexp_g_string (probar con g y sin g)

- **`i`(insensible)** a mayúsculas y minúsculas.
- **`m`(multilinea)**: varía el comportamiento de los caracteres especiales `^` y `$` de las expresiones regulares. Cuando está activado el modificador `m` si la cadena en la que se busca un patrón tiene varias líneas (por ejemplo en el caso de `textarea`), la cadena es tratada como varias líneas, con tantos comienzos y finales de línea como líneas haya. Por lo que si `m` está activado, al utilizar el carácter `^cad` se va a buscar `cad` en cada comienzo de línea, y lo mismo con el carácter `$`. Por defecto, aunque `cad` tuviera varias líneas sería tratado como una sola línea, con un solo comienzo de línea y un solo fin de línea.

Ejemplo: crear un textarea y buscar el patrón `^hola`

5.2. Caracteres especiales en el patrón de las expresiones regulares. Clases de caracteres

.	Cualquier carácter excepto nueva línea	/a.e/	Que aparezca cualquier carácter, excepto nueva línea entre la a y la e: "ape" y "axe".
\w	Coincide con caracteres del tipo (letras, dígitos, subrayados)	/\w/	Que aparezca un carácter (letra, dígito o subrayado): "J" en "JavaScript".
\W	Coincide con caracteres que no sean (letras, dígitos, subrayados)	/\W/	Que aparezca un carácter (que no sea letra, dígito o subrayado): "%" en "100%".
\n	Coincide con una nueva línea		
\s	Coincide con un espacio en blanco		
\S	Coincide con un carácter que no es un espacio en blanco		
\t	Un tabulador		
\r	Un retorno de carro		
\d	Dígitos del 0 al 9		
\D	Cualquier carácter que no sea un dígito		

\w: cualquier carácter alfanumérico del alfabeto latino básico (excepto ñ) mayúsculas o minúsculas, incluido

\s: espacio en blanco, tabulador, retorno de línea, avance de página, , etc

**** Para escapar un carácter especial y, por tanto, no sea interpretado, y se trata como un literal. Por ejemplo: /\./ : se busca el carácter **.** y no cualquier carácter

5.2. Caracteres especiales en el patrón de las expresiones regulares
Cuantificadores

Carácter	Coincidencias	Patrón	Ejemplo de cadena
*	Coincide 0 o más veces	/se*/	Que la "e" aparezca 0 o más veces: "seeee" y también "se".
?	Coincide 0 o 1 vez	/ap?	Que la p aparezca 0 o 1 vez: "apple" y "and".
+	Coincide 1 o más veces	/ap+/	Que la "p" aparezca 1 o más veces: "apple" pero no "and".
{n}	Coincide exactamente n veces	/ap{2}/	Que la "p" aparezca exactamente 2 veces: "apple" pero no "apabullante".
{n,}	Coincide n o más veces	/ap{2,}/	Que la "p" aparezca 2 o más veces: "apple" y "appple" pero no en "apabullante".
{n,m}	Coincide al menos n, y máximo m veces	/ap{2,4}/	Que la "p" aparezca al menos 2 veces y como máximo 4 veces: "apppppple" (encontrará 4 "p").

Ejemplo ? : /ap?/ → la coincidencia del patrón en **`apple`** es **`ap`**, y en **`and`** es **`a`**.

Ejemplo ? : /ap?l/ → **`apple`** no tiene ninguna coincidencia con el patrón. **`aple`** sí, y **`ale`** sí.

5.2. Caracteres especiales en el patrón de las expresiones regulares. Aserciones

Carácter	Coincidencias	Patrón	Ejemplo de cadena
^	Al inicio de una cadena	/^Esto/	Coincidencia en "Esto es...".
$	Al final de la cadena	/final$/	Coincidencia en "Esto es el final".

\b: es un límite de palabra (comienzo o final). Por ejemplo:

- `/na\b/` buscará la cadena seguida de un límite de palabra (`,` `:` `;` retorno de línea tabulador `?` ... O lo que es lo mismo al final de una palabra
- `/\bna/` buscará la cadena precedida de un límite de palabra (`,` `:` `;` retorno de línea tabulador `?` ... O lo que es lo mismo al comienzo de una palabra

\B: al contario. Por ejemplo:

- `/na\B/` buscará la cadena pero no seguida de un límite de palabra. Siendo límite de palabra (`,` `:` `;` retorno de línea tabulador `?` …. O lo que es lo mismo, no buscará la cadena al final de una palabra, la cadena podrá estar al comienzo o en medio de una palabra pero no al final.
- `/\Bna/` buscará la cadena pero no precedida de un límite de palabra (`,` `:` `;` retorno de línea tabulador `?` ... O lo que es lo mismo, no buscará la cadena al comienzo de una palabra, la cadena podrá estar al final o en medio pero no al comienzo de una palabra

5.2. Caracteres especiales en el patrón de las expresiones regulares. Aserciones

cad1(?=cad2) → que aparezca cad1 seguido de cad2. La ocurrencia sería cad1. Ejemplo: /e(?=\scla)/ (e seguido de uno o más espacios en blanco y después cla en la cadena "Hoy es el primer dia d**e** clase" sí hay coincidencia.

```
var patron=/e(?=\sclase)/;

var str="Final de clase";

demo.innerHTML+=patron.exec(str); /devuelve e


var patron=/e(?=\sclase)/;

var str="Final de hora";

demo.innerHTML+=patron.exec(str); /no devuelve nada. Hay e
pero no seguido de " clase"


var patron=/e\sclase/;

var str="Final de clase";

demo.innerHTML+=patron.exec(str); /devuelve e clase
```

5.2. Caracteres especiales en el patrón de las expresiones regulares. Aserciones

`cad1(?!cad2)` → que aparezca cad1 si no está seguido de cad2. La ocurrencia sería cad1.

`(?<=cad2)cad1` → que aparezca cad1 si está precedido de cad2. La ocurrencia sería cad1. No soportada en todos los navegadores.

`(?<!cad2)cad1` → que aparezca cad1 si no está precedido de cad2. La ocurrencia sería cad1. No soportada en todos los navegadores.

5.2. Caracteres especiales en el patrón de las expresiones regulares. Grupos y rangos

`[...]`	Cualquier carácter entre corchetes	`/a[px]e/`	Que aparezca alguno de los caracteres "p" o "x" entre la a y la e: "ape", "axe", pero no "ale".
`[^...]`	Cualquier carácter excepto los que están entre corchetes	`/a[^px]/`	Que aparezca cualquier carácter excepto la "p" o la "x" después de la letra a: "ale", pero no "axe" o "ape".

`[a-c]` Cualquier carácter del rango, a,b,c

`[A-D]` Cualquier carácter del rango: A,B,C,D

`[4-7]` Cualquier carácter del rango: 4,5,6,7

cad1|cad2→ que aparezca o cad1 o cad2. Ejemplo: /rojo|verde/ en la cadena "Cuadro rojo" si encontrará una coincidencia. En la cadena "Cuadro verde" también. En la cadena "Cuadro amarillo" no.

```
var patron=/rojo|verde|azul/;
var str="Pajaro rojo";  //Cumple con el patrón.
var patron=/(rojo|verde|azul)3/;
var str="Pajaro rojo3";  //Cumple con el patrón.
```

`()`: para aplicar funciones a un grupo de caracteres. Por ejemplo:

`/(le)+/g`: buscará `le` 1 o más veces. En la cadena `lelere` encontrará `lele`

Métodos del objeto RegExp

`exec()`	Busca la coincidencia en una cadena. Devolverá la primera coincidencia.
`test()`	Busca la coincidencia en una cadena. Devolverá true o false.

El **método `exec()`** se ejecutará así:

```
var str = "Is this it?";

//expresion→ que comience por is ignorando mayúsculas y minúsculas.
var expresion = /^is/i;
```

Devolverá en un array la 1ª ocurrencia de `expresion` encontrada en `str` aunque haya más de 1. **NO devolverá más ocurrencias aún utilizando modificador g. Si no encuentra ninguna ocurrencia devuelve null.**

```
var result = expresion.exec(str);;  // result será Is
```

El **método `test()`** se ejecutará así:

```
var str = "Is this it?";

//expresion→ que comience por is ignorando mayúsculas y minúsculas.
var expresion = /^is/i;
```

Devuelve `true` si encuentra alguna ocurrencia de `expresion` en `str`, o `false` si no hay ninguna.

```
var result = expresion.test(str);; //devuelve true
```

Métodos de String que usan expresiones regulares

`cadena.`**`match`**`(expresionRegular)`**`:`** devuelve un `array` con todas las ocurrencias del patrón encontradas en la cadena, si la expresión regular tiene el modificador `g` activado. Si no tiene activado el modificador `g` devuelve un `array` con solo la primera ocurrencia. Si no encuentra ninguna ocurrencia del patrón en la cadena devuelve `null`.

`cadena.`**`replace`**`(expresionRegular, nuevacadena)`**`:`** reemplaza todas las ocurrencias del patrón encontradas en la cadena por `nuevacadena`, si la expresión regular tiene el modificador `g` activado. Si no tiene activado el modificador `g` reemplaza solo la primera ocurrencia. Si no encuentra ninguna ocurrencia del patrón en la cadena devuelve la cadena sin modificar.

`cadena.`**`search`**`(expresionRegular)`**`:`** devuelve la posición en la que se encuentra la primera ocurrencia del patrón encontrada en la cadena. Si no encuentra ninguna ocurrencia del patrón en la cadena devuelve `-1`. No devuelve más posiciones aunque la expresión regular tenga el modificador `g` activado.

constructor RegExp()

Hasta ahora hemos creado las expresiones regulares literales, pero se puede crear una expresión regular con el constructor `RegExp()`. La expresión regular se compondrá llamando al constructor y pasándole 2 argumentos de tipo `String` (que pueden ser variables):

- Primer argumento: un string que guardará el patrón (sin las `/`)
- Segundo argumento: un string que guardará los modificadores. Este argumento es opcional.

```
var patron="[0-3]";
var modificadores="g";
var expr=new RegExp(patron,modif)    //1° argumento el patron y 2° modificadores
var texto="hola3"
expr.test(texto) // devuelve true porque encuentra un 3 en el texto
```

Cuando creamos una expresión regular con `RegExp` el navegador ha de crear a partir de ella una expresión regular literal, y esto le lleva más tiempo.

Sin embargo utilizar RegExp es necesario cuando la expresión regular no es conocida de antemano, por ejemplo si la facilita el usuario a través de un input, o va variando a lo largo del programa.

Expresiones regulares literales. Problemas metacaracteres

Utilizar metacaracteres de expresiones regulares como caracteres regulares con expr.literal

Si una expresión literal contiene metacaracteres que no se desea que sean utilizados con su significado especial hay que escaparlos (como hemos visto anteriormente):

```
var expr=/hola\.$/i; // la cadena hola. (con el punto literal) al final de línea

var expr=new RegExp(patron,modif)    //1° argumento el patron y 2° modificadores
var texto="Hola Juan: por favor cuando me veas dime hola."

text.match(expr) // devuelve "hola."
```

```
>> var expr=/hola\.$/i;
<- undefined
>> expr
<- ▶ /hola\.$/i
>> var texto="Hola Juan: por favor cuando me veas dime hola."
<- undefined
>> texto.match(expr)
<- ▶ Array [ "hola." ]
```

constructor RegExp() – Problemas metacaracteres

Utilizar metacaracteres de expresiones regulares como caracteres regulares con RegExp()-I

¿Qué ocurre si en el string de patron se utilizan metacaracteres de expresiones regulares pero queremos tratarlos en la expresión regular como un carácter regular?

- `. * + ? ^ $ { } ( ) | [ ] < ! = \ -` : si queremos que estos metacaracteres de expresiones regulares formen parte del patrón pero como carácter regular tendríamos que escaparlos, para que no ocurra lo siguiente.

```
>> var patron="3$"
<- undefined
>> var e=new RegExp(patron)
<- undefined
>> e
<- ▶ /3$/
>> "3$ es lo que cuesta".match(e)
<- null
>> "El numero 3".match(e)
<- ▶ Array [ "3" ]
```

¡¡¡ Observa cómo queda la expresión regular en la consola tras procesarla !!!

constructor RegExp() – Problemas metacaracteres

Utilizar metacaracteres de expresiones regulares como caracteres regulares con RegExp()-II

- Si quiero buscar `3$` (siendo $ un carácter regular, es decir significa una moneda), tendría que escapar el carácter $, y para que el escape sea interpretado en expresiones regulares como un escape tenemos que escapar también el carácter de escape.

```
>> var patron="3\$"
<- undefined
>> var e=new RegExp(patron)
<- undefined
>> e
<- ▶ /3$/
>> "El numero 3".match(e)
<- ▶ Array [ "3" ]
>> "3$ es lo que cuesta".match(e)
<- null
```

```
>> var patron="3\\$"
<- undefined
>> var e=new RegExp(patron)
<- undefined
>> e
<- ▶ /3\$/
>> "El numero 3".match(e)
<- null
>> "3$ es lo que cuesta".match(e)
<- ▶ Array [ "3$" ]
```

- En definitiva, si nuestra expresión regular en literal sería: `/3\$/` Al crear la expresión regular con el constructor `RegExp()` y pasar el patrón como string, el `\$`, va a ser interpretado por el constructor como `$`, que significa fin de línea en expresiones regulares, y no la moneda dólar. Por lo que tenemos que escapar el escape así: `\\$`, para que este string sea interpretado por el constructor `RegExp` como `\$`

Constructor RegExp() – Problemas metacaracteres

Utilizar metacaracteres de expr. regulares como caracteres regulares con RegExp()-III

- En el ejemplo anterior el programa Javascript sabía que en el patron existía un `$` (como moneda dólar), y por ello en la definición del string patrón se ponía `\\$`, pero puede ocurrir que el programa Javascript no sepa de antemano si el patron va a contener alguno de estos caracteres (`. * + ? ^ $ { } ( ) | [ ] \ - < ! =`) por ejemplo si el contenido del patron va a ser tomado de forma externa, como a través de un `textarea`. Por lo que conviene transformar el patrón desconocido a un patrón con estos posibles caracteres especiales escapados (2 veces) de la siguiente forma:

```
function escapeRegExp(string) {
  return string.replace(/[.*+?^${}()|[\]\\-<!=]/g, '\\$&')
}
```

```
function escapeRegExp(string) {
    let cambiar = /[.*+?^${}()|[\]\\]/g;
    return string.replace(cambiar, '\\$&');
}
```

Esta función lo que hace es sustituir cualquier de estos caracteres especiales que hay incluidos entre los corchetes que están en negrita por `\\` seguido del carácter en cuestión (`$&`)

```
var patron=escapeRegExp(textarea.value);

var expr=new RegExp(patron);
```

constructor RegExp(). Problemas con metacaracteres

Utilizar metacaracteres de expresiones regulares como metacaracteres con RegExp()-I

Cuando construimos una expresión regular con el constructor `RegExp()`, le pasamos como primer argumento el patrón de la expresión regular a través de un string.

En este punto pueden entrar en conflicto los metacaracteres del objeto string y los metacaracteres del patrón de los objetos `RegExp`

Los metacaracteres son caracteres que tienen un significado especial, a diferencia de los caracteres regulares que no tienen ningún significado especial, sino únicamente su valor literal.

En string tenemos los siguientes metacaracteres:

- ' : comilla sencilla -> delimitador del string
- " : comilla doble -> delimitador del string
- \ : carácter de escape -> para eliminar el significado especial que pueda tener el siguiente carácter al escape.
- `\n`: retorno de línea
- `\r`: retorno de carro
- `\v`: tabulación vertical.
- `\t`: tabulación
- `\b`: retroceso
- `\f`: avance de página

En los patrones de expresiones regulares tenemos los siguientes metacaracteres:

```
. * + ? ^ $ { } ( ) | [ ] \ < ! = - \b \B \s \S \w \W \d \D \t \r \n \v \f
```

constructor RegExp() – Problemas metacaracteres

Utilizar metacaracteres de expresiones regulares como metacaracteres con RegExp()-II

¿Qué ocurre si en el string de patron se utilizan los metacaracteres siguientes de expresiones regulares?

- `. * + ? ^ $ { } ( ) | [ ] < ! = -` :no hay ningún problema si el string utiliza estos metacaracteres de expresiones regulares, porque no existen en el objeto string.
- `\t \r \n \v \f` : no supone un hay problema porque tienen significados iguales para el objeto string que para el objeto expresión regular.

```
>> var patron='3$';
<- undefined
>> var e=new RegExp(patron)
<- undefined
>> e
<- ▸ /3$/
>> "Es el numero 3".match(e)
<- ▸ Array [ "3" ]
>> "3$ es lo que cuesta".match(e)
<- null
```

```
>> var patron="\thola"
<- undefined
>> patron
<-  "      hola"
>> var e=new RegExp(patron)
<- undefined
>> e
<- ▸ /     hola/
>> "   hola".match(e)
<- ▸ Array [ "\thola" ]
```

constructor RegExp() – Problemas metacaracteres

Utilizar metacaracteres de expresiones regulares como metacaracteres con RegExp()-III

¿Qué ocurre si en el string de patron se utilizan los metacaracteres siguientes de expresiones regulares?

- **`\B \s \S \w \W \d \D`**: si el string `patron` utiliza estos metacaracteres de expresiones regulares de esta forma habría un problema, ya que el objeto string ve a estos bloques como que se está escapando la `B, s, S, w, W, d` y `D` respectivamente, por lo que cuando el constructor `RegExp()` reciba estos bloques procesará realmente `B, s, S, w, W, d` y `D` respectivamente, y por tanto, no serán los metacaracteres de expresiones regulares `\B \s \S \w \W \d \D`

```
>> var patron="\d"
<- undefined
>> var e=new RegExp(patron)
<- undefined
>> e
<- ▸ /d/
>> "ho3a".match(e)
<- null
```

Como solución tendríamos que escapar la `\` de esta forma `\\B \\s \\S \\w \\W \\d \\D` :

```
>> var patron="\\d"
<- undefined
>> var e=new RegExp(patron)
<- undefined
>> e
<- ▸ /\d/
>> "ho3a".match(e)
<- ▸ Array [ "3" ]
```

constructor RegExp() – Problemas metacaracteres

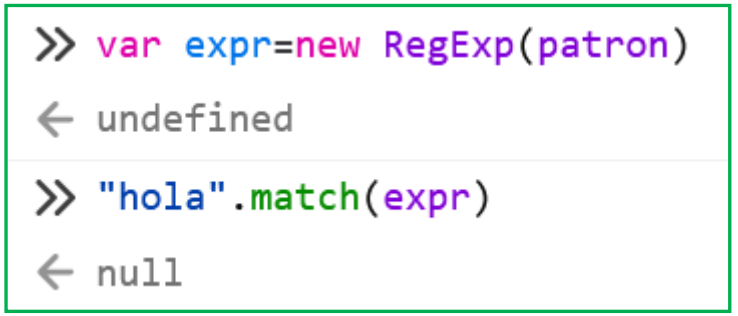
Utilizar metacaracteres de expresiones regulares como metacaracteres con RegExp()-IV

¿Qué ocurre si en el string de patron se utilizan los metacaracteres siguientes de expr regulares?

- **\b:** aquí también habría problema (aunque diferente a los anteriores). Ocurre que `\b` es un metacaracter también en string (no solo en expresiones regulares) pero tiene un significado diferente como metacaracter de string que como metacaracter de expresión regular. Por tanto si ponemos en un string de un patrón esto:

```
var patron="\bhola";
// cuando el string patron sea utilizado la \b será sustituida por tecla de retroceso.

var expr=new RegExp(patron);
// cuando el constructor RegExp procese el string patrón sustituirá \b por la tecla de retroceso. Y no funcionará porque se buscará la cadena retroceso seguida de hola.
```



constructor RegExp() – Problemas metacaracteres

Utilizar metacaracteres de expresiones regulares como metacaracteres con RegExp()-V

Por lo que tenemos que poner en el string: `\\b` en definitiva tenemos que escapar la `\` para que el string del patrón le llegue a la expresión regular como `\\b` que interpretará como `\b`

```
var patron="\\bhola";
// cuando el objeto string es utilizado se sustituirá \\b por \b


var expr=new RegExp(patron);
// cuando el constructor RegExp procese el string interpretará la \b cuyo
significado límite de palabra
```