Desarrollo Web en Entorno Cliente.

Unidad 3.

JavaScript
Expresiones Regulares.

Expresiones Regulares

- Nos permiten ver si una cadena de texto cumple un determinado formato, por ejemplo, si todos son dígitos, si tiene formato de fecha y hora, etc.
- Nos permiten extraer partes de esa cadena de texto que cumplan ese formato, por ejemplo, si dentro de un texto más o menos largo hay una fecha, podemos extraerla fácilmente.

Caracteres especiales de las "Expresiones Regulares".

Carácter	Texto buscado
^	Principio de entrada o línea.
$	Fin de entrada o línea.
*	El carácter anterior 0 o más veces.
+	El carácter anterior 1 o más veces.
?	El carácter anterior una vez como máximo (es decir, indica que el carácter anterior es opcional).
.	Cualquier carácter individual, salvo el de salto de línea.
x|y	x o y.
{n}	Exactamente n apariciones del carácter anterior.
{n,m}	Como mínimo n y como máximo m apariciones del carácter anterior.
[abc]	Cualquiera de los caracteres entre corchetes. Especifique un rango de caracteres con un guión (por ejemplo, [a-f] es equivalente a [abcdef]).
[^abc]	Cualquier carácter que no esté entre corchetes. Especifique un rango de caracteres con un guión (por ejemplo, [^a-f] es equivalente a [^abcdef]).
\b	Límite de palabra (como un espacio o un retorno de carro).
\B	Cualquiera que no sea un límite de palabra.
\d	Cualquier carácter de dígito. Equivalente a [0-9].
\D	Cualquier carácter que no sea de dígito. Equivalente a [^0-9].
\f	Salto de página.
\n	Salto de línea.
\r	Retorno de carro.
\s	Cualquier carácter individual de espacio en blanco (espacios, tabulaciones, saltos de página o saltos de línea).
\S	Cualquier carácter individual que no sea un espacio en blanco.
\t	Tabulación.
\w	Cualquier carácter alfanumérico, incluido el de subrayado. Equivalente a [A-Za-z0-9_].
\W	Cualquier carácter que no sea alfanumérico. Equivalente a [^A-Za-z0-9_].

La tabla siguiente contiene algunos de los "Patrones" más utilizados a la hora de validar formularios.

Cualquier letra en minuscula	[a-z]	
Entero	^(?:\+|-)?\d+$	
Correo electrónico	/[\w-\.]{3,}@([\w-]{2,}\.)*([\w-]{2,}\.)[\w-]{2,4}/	
URL	^(ht|f)tp(s?)\:\/\/[0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*(:(0-9)*)*(\/?)([a-zA-Z0-9\-\.\?\,\'\/\\\+&%\$#_]*)?$	
Contraseña segura	(?!^[0-9]*$)(?!^[a-zA-Z]*$)^([a-zA-Z0-9]{8,10})$ (Entre 8 y 10 caracteres, por lo menos un digito y un alfanumérico, y no puede contener caracteres especiales).	
Fecha	^\d{1,2}\/\d{1,2}\/\d{2,4}$	(Por ejemplo 01/01/2007)
Hora	^(0[1-9]|1\d|2[0-3]):([0-5]\d):([0-5]\d)$	(Por ejemplo 10:45:23)
Número tarjeta de crédito	^((67\d{2})|(4\d{3})|(5[1-5]\d{2})|(6011))(-?\s?\d{4}){3}|(3[4,7])\d{2}-?\s?\d{6}-?\s?\d{5}$	
Número teléfono	^[0-9]{2,3}-? ?[0-9]{6,7}$	

Código postal `^([1-9]{2}|[0-9][1-9]|[1-9][0-9])[0-9]{3}$`

Certificado Identificación Fiscal `^(X(-|\.)?0?\d{7}(-|\.)?[A-Z]|[A-Z](-|\.)?\d{7}(-|\.)? [0-9A-Z]|\d{8}(-|\.)?[A-Z])$`

Una expresión regular es un patrón que se emplea para compararlo con un grupo de caracteres.

Una expresión regular es un patrón que puede estar formado por un conjunto de caracteres (letras, números o signos) y por un conjunto de metacaracteres que representan otros caracteres o que indican la forma de combinar los caracteres.

Caracteres → Letras, números, signos de puntuación (**, .**), símbolos (**@,%**)

Metacaracteres → No se representan a ellos mismos, sino que son interpretados de una manera especial y son:

\ . * ? + {} [] - () | ^ $

\ Carácter de Escape general, se emplea para escapar cualquiera de los metacaracteres, incluido el mismo \.

. Representa cualquier carácter (excepto salto de línea)

Metacaracteres Cuantificadores:

***** El patrón que lo precede se repite 0 ó más veces.

? El patrón se repite 0 ó 1 vez (lo que le precede es opcional)

+ El patrón se repite 1 ó más veces.

{x,y} El patrón se repite un mínimo de x veces y un máximo de y veces. (lo que le precede)

[ ] Indica un conjunto de caracteres.

- Define un rango de caracteres en un conjunto.

( ) Define una agrupación, un subpatrón que puede ser posteriormente referenciado.

| Indica una expresión alternativa. (Elegir entre dos expresiones)

^ Indica que el patrón que lo acompaña está al principio de la cadena.

[^] Niega el conjunto de caracteres. Si aparece al principio de un conjunto de caracteres indica que no esté en ese conjunto de caracteres.

$ Indica que el patrón está al final de una cadena.

Patrón	Significado
.	cualquier carácter (excepto \n y \r)
^c	empezar por el carácter c
c$	terminar por el carácter c
c+	1 o más caracteres c
c*	0 o más caracteres c
c?	0 o 1 caracteres c
\n	nueva línea
\t	tabulador
\	escape, para escribir delante de caracteres especiales: ^ . [ ] % ( ) | * ? { } \
(cd)	caracteres c y d agrupados
c|d	carácter c o d
c{n}	n veces el carácter c
c{n,}	n o más caracteres c
c{n,m}	desde n hasta m caracteres c

Patrón	Significado
[a-z]	cualquier letra minúscula
[A-Z]	cualquier letra mayúscula
[0-9]	cualquier dígito
[cde]	cualquiera de los caracteres c, d o e
[c-f]	cualquier letra entre c y f (es decir, c, d, e o f)
[^c]	que no esté el carácter c
[^c-f]	que no esté cualquier letra en c y f

Caracteres Genéricos:

Son secuencias de escape que definen conjuntos de caracteres, cada uno de estos caracteres genéricos divide el conjunto de caracteres en dos grupos, un grupo de caracteres que sí que cumplen el patrón y otro grupo de caracteres que no cumplen el patrón.

Patrón	Significado
\w	cualquier carácter "palabra": letra o dígito o subrayado (pero no vocales acentuadas, ñ, ç, etc.)
\W	lo contrario de \w
\d	cualquier dígito
\D	lo contrario de \d
\s	cualquier carácter espacio en blanco (tabulador, espacio en blanco, salto de línea)
\S	lo contrario de \s
\b	busca un emparejamiento a partir de un límite de palabra
\B	busca un emparejamiento cuando **no** es un límite de palabra

Ejemplos:

[abc] → Coincide con la cadena si en esta hay cualquiera de estos tres caracteres. "a", "b", "c"

[a-c] → Coincide si existe una letra en el rango ("a","b" ó "c")

a|b|c → Coincide con la cadena si en ésta hay cualquiera de estos tres caracteres: "a", "b", "c"

[^abc] → Coincide con la cadena si en ésta No hay ninguno de estos tres caracteres. "a","b" ó "c"

c[ao]sa → Coincide con "casa" y con "cosa"

c[^a]sa → Coincide con "cosa", pero no con "casa"

Ejemplo.

Escribe una expresión regular que se corresponda a todos los siguientes formatos:

555-555-5555
(555)-555-5555

555.555.5555
(555).555.5555

555/555/5555
(555)/555/5555

555 555 5555
(555) 555 5555

Solución:

```
\(?\d{3}\)?([-\/\.\s])\d{3}\1\d{4}
```

```
(\(\d{3}\)|\d{3})([-\/\.\s])\d{3}\2\d{4}
```

Una expresión regular se emplea para:

- Buscar en un texto las partes que cumplen un determinado patrón.
- Validar que los datos cumplen un determinado patrón, por ejemplo en un formulario.

Las expresiones regulares se pueden emplear en:

- Comandos de sistemas operativos, como sed y grep en Linux.
- Editores de texto como emacs.
- Lenguajes de programación, de forma nativa como JavaScript, PHP, awk y Perl, o a través de librerías como Java ó .NET.
- En la definición de tipos de datos, como en XML Schema.

Básicamente existen tres estilos de expresiones regulares:

Perl y **PCRE** que son casi iguales, y **POSIX** que varía.

Cuando vayas a usar expresiones regulares en un lenguaje de programación debes consultar la documentación para ver que estilo admite.

Por ejemplo, si consultamos la documentación de **JavaScript** (Standard ECMA-262) encontramos que en el punto 15.10 nos dice que la sintaxis de sus expresiones regulares está **basada en Perl**, es más fácil entender la documentación de Mozilla que también usa Perl, para entender las expresiones regulares.

PHP tiene dos estilos: Por un lado los patrones PCRE compatibles con Perl y las expresiones regulares compatibles con Posix

Ejemplos:

[0-9]{3}-[0-9]{3}-[0-9]{4} → Teléfonos (Secuencia de números del [0 al 9] y se tiene que repetir {3} veces(dígitos)).

352-754-7423

[0-9]{5}(-[0-9]{4})? → Códigos Postales en EEUU la segunda parte es opcional por eso lleva ?

85312-6589

[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4} → Detecta correos electrónicos pero no es el estándar

Tenemos varios sitios webs para probar las expresiones regulares:

- regexpal.com
- www.regextester.com

El Objeto RegExp

Es un objeto intrínseco (Built-in) (Objeto nativo de JavaScript, que están creados en el lenguaje y disponibles sin que el programador tenga que declararlo), otros objetos intrínsecos de JavaScript son: Array, Date, Math, Number, String, Boolean, Global.

El Objeto ExpReg me permite usar expresiones regulares en JavaScript.

En JavaScript se pueden crear expresiones regulares de dos formas:

var txt=/patrón/atributos; → Expresión Regular "**Literal**", "**Estática**" ó "**Constante**" por que posteriormente no se puede modificar durante la ejecución. La ventaja es un mayor rendimiento.

Var txt=new RegExp(patrón, atributos); → Creamos la expresión regular como un objeto a partir de RegExp, se utiliza cuando no se conoce la expresión regular por que proviene de otra fuente (por ejemplo la entrada del usuario) ó por que puede cambiar durante la ejecución.

En ambos casos los atributos (también llamados modificadores ó flags) no son obligatorios.
Recomendación, visitar http://www.w3schools.com/js/ y ver la referencia a RegExp, donde podemos ver un resumen de la sintaxis de las expresiones regulares en JavaScript, donde encontramos la tabla de metacaracteres y cuantificadores, podemos destacar:

Propiedades del Objeto RegExp

- **global:** El modificador "**g**" indica que la búsqueda es global y no se detiene en la primera ocurrencia que encuentra.
- **ignoreCase:** El modificador "**i**" indica que no es sensible a mayúsculas y minúsculas, se ignoran mayúsculas y minúsculas.
- **multiline:** El modificador "**m**" indica que el texto es multilínea y las comparaciones se hacen línea a línea. Indica que si el texto tiene varias líneas, varios saltos de línea y se indican las anclas de inicio y final de cadena en la expresión regular, no se tienen que aplicar desde el inicio hasta el final de la cadena sino desde el inicio hasta el final de cada línea

Métodos del objeto RegExp

- **compile()**: Cambia la expresión regular del objeto.
- **exec()**: Busca la expresión regular, devuelve el valor encontrado y recuerda la posición.
- **test()**: Busca la expresión regular, devuelve true ó false.

El Objeto String

El objeto String de JavaScript también tiene métodos que admiten expresiones regulares como parámetros.
En concreto los métodos:

- **search():** Busca en una cadena la expresión regular, devuelve la posición.
- **match():** Busca en una cadena la expresión regular, devuelve un array con los valores emparejados.
- **replace():** Sustituye unos caracteres por otros caracteres.
- **split():** Divide una cadena en una array de cadenas.

Ejemplo 1.

Validar una matrícula de coche con el formato de la Unión Europea: 4 dígitos, un espacio en blanco y 3 letras.
Podemos usar una expresión regular a partir de un objeto ó a partir de un literal.

```
function validaMatricula() {
  var mat = document.getElementById("matricula").value;

  var ex1 = new RegExp("^[0-9]{4} [A-Z]{3}$");
  var ex2 = /^[0-9]{4} [A-Z]{3}$/;

  if(ex1.test(mat))
    alert("Ok");
  else
    alert("Error");

  if(ex2.test(mat))
    alert("Ok");
  else
    alert("Error");
}
```

Ejemplo 2.

Validar una fecha, pero solo el formato, es decir, no valida que la fecha real introducida sea válida, osea que 31 de febrero es válido. dd/mm/aaaa.

```
function validaFecha() {
var fec = document.getElementById("fecha").value;

  var ex1 = new RegExp("^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]
   |1[012])\/[0-9]{1,2}$");
  var ex2 = /^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[012])\/[0-9]
   {1,2}$/;

  if(ex1.test(fec)) alert("Ok");
  else alert("Error");

  if(ex2.test(fec)) alert("Ok");
  else alert("Error");
 }
```

Ejemplo 3.

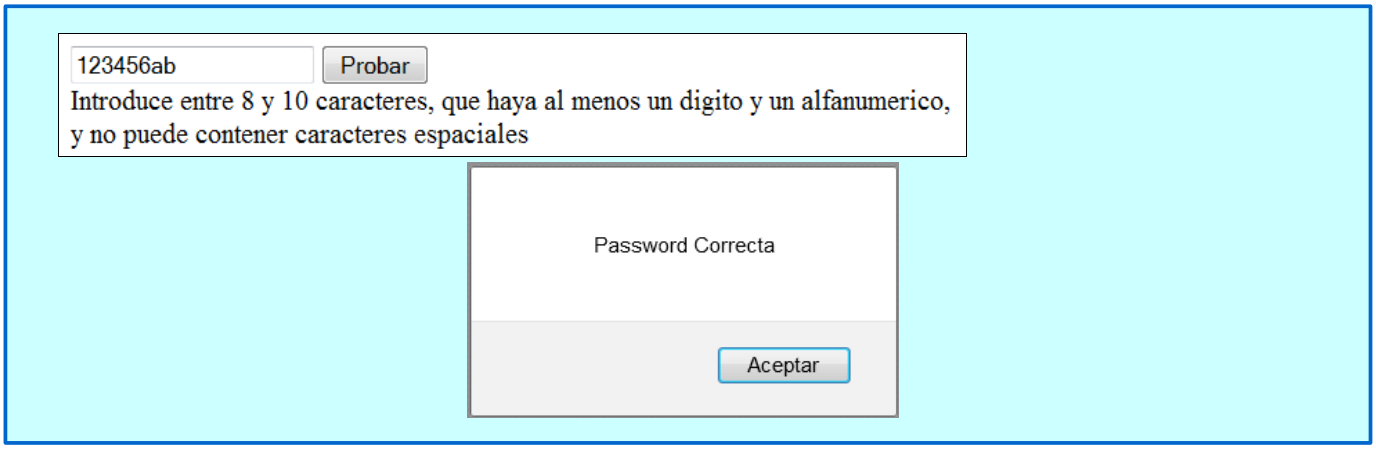
```
var exp = new RegExp(" (el|la|los|las) ", "gim");
var txt = "En un lugar de la Mancha, de cuyo nombre no quiero
   acordarme, no ha mucho tiempo que vivía un hidalgo de los de
   lanza en astillero, adarga antigua, rocín flaco y galgo
   corredor. Una olla de algo más vaca que carnero, salpicón las
   más noches, duelos y quebrantos los sábados, lentejas los
   viernes, algún palomino de añadidura los domingos, consumían
   las tres partes de su hacienda.";

var res = txt.match(exp);

document.write(txt + "<br />");

for(i = 0; i < res.length; i++) {
  document.write(i + ": " + res[i] + "<br />");
}
```

¿Cual es el resultado que genera el siguiente código?**Un array con los artículos definidos del texto.**

En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lentejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda.
0: la
1: los
2: las
3: los
4: los
5: los
6: las

Ejercicio 1. Resuelto

Realizar el siguiente formulario, para introducir una contraseña de forma que nos devuelva si la contraseña introducida es correcta ó no.



Solución:

```
<html >
<head>
   <title> Expresiones Regulares 1. ej1E.html</title>
   <script type="text/javascript">

   function validar(campo) {
       var Patron = /(?!^[0-9]*$)(?!^[a-zA-Z]*$)^([a-zA-Z0-9]{8,10})$/;

       if ((campo.value.match(Patron)) && (campo.value!='')) {
           alert('Password Correcta');
       } else {
           alert('Password Incorrecta.');
          campo.focus();
       }
   }

   </script>
</head>

<body>
  <form action="#" method="post">
      <p><input type="text" name="date" onblur="validar(this);">
         <input name="button" type="button" value="Probar">
      <br>
         Introduce entre 8 y 10 caracteres, que haya al menos un digito y un alfanumerico, <br>
         y no  puede contener caracteres espaciales
  </form>
</body>
</html>
```

Las expresiones regulares nos permiten ver si una cadena de texto cumple un determinado formato, por ejemplo, si todo son dígitos, si tiene formato de fecha y hora, etc.

También nos permiten extraer partes de esa cadena de texto que cumplan ese formato, por ejemplo, si dentro de un texto más o menos largo hay una fecha, podemos extraerla fácilmente.

Expresiones Simples

Una expresión regular se construye con una cadena de texto que representa el formato que debe cumplir el texto. En JavaScript se puede hacer de dos formas, bien instanciando una clase RegExp pasando como parámetro la cadena de formato, bien poniendo directamente la cadena de formato, en vez de entre comillas, entre /

Son equivalentes

```
var reg = new RegExp("aa")
var reg = /aa/
```

Ver si un texto tiene una determinada secuencia de letras o números fija. La expresión simplemente es esa cadena y podemos ver si un texto la tiene usando el método **match()** de ese texto

```
var reg = /javascript/;
"hola javascript".match(reg);
// devuelve un array de 1 elemento ["javascript"], indicando que sí existe esa cadena dentro del texto

 "adios tu".match(reg);
// devuelve null, indicando que no existe javascript dentro de "adios tu".
```

No es necesario definir la expresión regular antes, podemos hacerlo así

```
"hola javascript".match(/javascript/);
// Devuelve ["javascript"]
```

Y para verificar si existe o no la cadena, podemos poner directamente un if

```
if ("hola javascript".match(/javascript/) {
        // Pasará por aquí, porque un array con un elemento se evalua como true en el if
 }

if ("adios tu".match(/javascript/) {
        // No pasa por aquí, null se evalúa como false en el if.
 }
```

Caracteres no alfabéticos ni numéricos

Algunos de los caracteres no numéricos ni alfabéticos tienen un significado especial (lo vemos más adelante), como por ejemplo [ ] { } ( ) * . ^ $ etc. No podemos ponerlos tal cual si forman parte de nuestro formato, debemos "escaparlos" poniendo \ delante

```
"esto es un *".match(/\*/);
// Devuelve ["*"] indicando que existe un asterisco.
```

Conjunto opcional de caracteres

A veces nos da igual que una letra, por ejemplo, sea mayúscula o minúscula, o queremos que sea una vocal, o un dígito. Cuando queremos que una de las letras de nuestro texto pueda ser una cualquiera de un conjunto de letras determinado, las ponemos entre [] en nuestra expresión. Por ejemplo, si nos vale "Javascript" y "javascript", podemos poner la expresión como /[Jj]avascript/ para indicar que nos vale J mayúscula o j minúscula

```
"javascript con minuscula".match(/[Jj]avascript/);
// Sí encuentra la cadena

"Javascript con minuscula".match(/[Jj]avascript/);
// También la encuentra.
```

Si los caracteres válidos son varios y van ordenados según el juego de caracteres, podemos poner el primero y el último separados por un -.
Por ejemplo, [a-z] vale para cualquier letra minúscula, [0-9] para cualquier dígito y [a-zA-Z] para cualquier letra mayúscula o minúscula

```
"aa2bb".match(/[0-9]/);  // Encuentra el 2, devolviendo ["2"].
```

Podemos hacer lo contrario, es decir, que la letra no esté en ese conjunto de caracteres. Se hace poniendo el juego de caracteres que no queremos entre [^ y].
Por ejemplo, para no dígitos pondríamos [^0-9]

```
"22 33".match(/[^0-9]/); // Encuentra el espacio en blanco, devolviendo [" "]
```

Conjuntos habituales

Hay varios conjuntos que se usan con frecuencia, como el de letras [a-zA-Z], el de dígitos [0-9] o el de espacios en blanco (espacio, tabulador, etc). Para estos conjuntos la expresión regular define formas abreviadas, como:

\w para letras, equivalente a [a-zA-Z]
\W para no letras, equivalente a [^a-zA-Z]
\d para dígitos, equivalente a [0-9]
\D para no dígitos, equivalente a [^0-9]
\s para espacios en blanco (espacios, tabuladores, etc).
\S para no espacios en blanco.
ver Metacharacters

Por ejemplo:

```
"aa2bb".match(/\d/); // Encuentra el 2, devolviendo ["2"]
```

Repetición de caracteres

Podemos querer buscar por ejemplo un conjunto de tres digitos, podemos hacerlo repitiendo tres veces el \d

```
"aa123bb".match(/\d\d\d/);   // Encuentra el 123, devolviendo ["123"]
```

pero esta forma es un poco engorrosa si hay muchos caracteres y es poco versátil.

Las expresiones regulares nos permiten poner entre { } un rango de veces que debe repetirse. por ejemplo

/\d{3}/ Busca 3 dígitos en la cadena

/\d{1,5}/ Busca entre 1 y 5 dígitos en la cadena.

/\d{2,}/ Busca 2 dígitos o más en la cadena.

Ejemplos:

```
"1234".match(/\d{2}/);
["12"]

"1234".match(/\d{1,3}/);
["123"]

"1234".match(/\d{3,10}/)
["1234"]
```

También suele haber rangos habituales como 0 o más veces, 1 o más veces, 0 ó 1 vez. Estos rangos habituales tienen caracteres especiales que nos permiten ponerlos de forma más simple.

* equivale a 0 o más veces {0,}

+ equivale a 1 o más veces {1,}

? equivale a 0 ó 1 vez {0,1}

Ejemplos:

```
"a2a".match(/a\d+a/);   // Encuentra a2a
"a234a".match(/a\d+a/);  // Encuentra a234a
```

Cosas como * o + encuentran el máximo posible de caracteres. Por ejemplo, si nuestro patrón es /a+/ y nuestra cadena es "aaaaa", el resultado será toda la cadena

```
"aaaaa".match(/a+/);  // Devuelve ["aaaaa"]
```

Para hacer que se encuentre lo menos posible, se pone un **?** detrás.

Así por ejemplo, si nuestra expresión regular es **/a+?/** y nuestra cadena es **"aaaaa"**, sólo se encontrará una "a"

```
"aaaaa".match(/a+?/);  // Devuelve ["aaaaa"]
```

→ El comportamiento inicial se conoce como "**greedy**" o codicioso, en el que el patrón intenta coger la mayor parte de la cadena de texto posible.

→ El segundo comportamiento se conoce como **"nongreedy"** o no codicioso, en el que el patrón coge lo menos posible de la cadena.

Extraer partes de la cadena.

A veces nos interesa no sólo saber si una cadena cumple un determinado patrón, sino extraer determinadas partes de él.

Por ejemplo, si una fecha está en el formato "27/11/2012" puede interesarnos extraer los números. Una expresión regular que vale para esta cadena puede ser:

```
/\d{1,2}\/\d{1,2}\/\d{4}/
```

suponiendo que el día y el mes puedan tener una cifra y que el año sea obligatoriamente de 4 cifras. En este caso

```
"27/11/2012".match(/\d{1,2}\/\d{1,2}\/\d{4}/);
```

nos devuelve un array con un único elemento que es la cadena "27/11/2012".
Para extraer los trozos, únicamente debemos poner entre paréntesis en la expresión regular aquellas partes que nos interesan. Es decir, **/(\d{1,2})\/(\d{1,2})\/(\d{4})/**

Si ahora ejecutamos el método match() con la misma cadena anterior, obtendremos un array de 4 cadenas. La primera es la cadena completa que cumple la expresión regular. Los otros tres elementos son lo que cumple cada uno de los paréntesis

```
"27/11/2012".match(/(\d{1,2})\/(\d{1,2})\/(\d{4})/);  // Devuelve el array
["27/11/2012", "27", "11", "2012"]
```

Los paréntesis también nos sirven para agrupar un trozo y poner detrás uno de los símbolos de cantidades. Por ejemplo: **"xyxyxyxy".match(/(xy)+/);** // Se cumple, hay xy una o más veces.

Usar lo encontrado en la expresión:

Las partes de la cadena que cumplen la parte de expresión regular entre paréntesis, se pueden reutilizar en la misma expresión regular.
Estas partes encontradas se van almacenando en \1, \2, \3... y podemos usarlas.
Esta posibilidad es realmente interesante si queremos por ejemplo, verificar que una cadena de texto va cerrada entre comillas del mismo tipo, es decir, queremos buscar algo como 'esto' o "esto", pero no nos vale 'esto".

La expresión regular para buscar una cadena entre este tipo de comillas puede ser /(['"]).*\1/ es decir, buscamos una ' o una " con ['"].

Hacemos que lo que se encuentre se guarde metiéndolo entre paréntesis (['"]) y a partir de ahí nos vale cualquier conjunto de caracteres terminados en \1, que es lo que habíamos encontrado al principio.

```
"'hola tu' tururú".match(/(['"]).*\1/);  // Devuelve ["'hola tu'", "'"]
"\"hola tu' tururú".match(/(['"]).*\1/); // Devuelve null, la cadena comienza con " y termina en '
```

Ignorar lo encontrado

A veces nos interesa encontrar una secuencia que se repita varias veces seguidas y la forma de hacerlo es con los paréntesis, por ejemplo, si ponemos /(pa){2}/ estamos buscando "papa".
Para evitar que esos paréntesis guarden lo encontrado en \1, podemos poner **?:**, tal que así /(?:pa){2}/, de esta forma encontraremos "papa", pero se nos devolverá el trozo "pa" encontrado ni lo tendremos disponible en \1. Compara las dos siguientes:

```
"papa".match(/(pa){2}/);    // Devuelve ["papa", "pa"]
"papa".match(/(?:pa){2}/);  // Devuelve ["papa"]
```

Posición de la expresión

A veces nos interesa que la cadena busque en determinadas posiciones. Las expresiones regulares nos ofrecen algunos caracteres especiales para esto.

^ indica el principo de cadena, por ejemplo, /^hola/ vale si la cadena "hola" está al principio

```
"hola tu".match(/^hola/);          // Devuelve ["hola"]
"pues hola tu".match(/^hola/);     // Devuelve null
```

**$** es el final de la cadena, por ejemplo /tu$/ vale si la cadena termina en "tu"

```
"hola tu".match(/tu$/);            // Devuelve ["tu"]
"hola tu tururú".match(/tu$/);     // Devuelve null
```

\b indica una frontera de palabra, es decir, entre un caracter "letra" y cualquier otra cosa como espacios, fin o principio de linea, etc. De esta forma, por ejemplo, `/\bjava\b/` buscará la palabra java, pero ignorará javascript

```
"java es güay".match(/\bjava\b/);          // Devuelve ["java"]
"javascript es güay".match(/\bjava\b/);    // Devuelve null
```

\B es lo contrario de \b, así por ejemplo, `/\bjava\B/` buscará una palabra que empiece por "java", pero no sea sólo java sino que tenga algo más

```
"java es güay".match(/\bjava\B/);          // Devuelve null
"javascript es güay".match(/\bjava\B/);    // Devuelve ["java"]
```

(?=expresion) sirve para posicionar el resto de la expresión regular y buscar antes o depués. Por ejemplo si queremos buscar un número que vaya delante de km, podemos hacer esto `/\d+(?= km)/` , es decir, uno o más dígitos seguidos de un espacio y las letras km. La diferencia con esta expresión (`/\d+ km/`) es que en el primer caso sólo casan con la expresión los números, mientras que en el segundo caso se incluye también el " km"

```
"11 millas 10 km".match(/\d+(?= km)/);    // Devuelve ["10"]
"11 millas 10 km".match(/\d+ km/);        // Devuelve ["10 km"]
```

Hay que tener cuidado si buscamos detrás, porque como el trozo (?=expresion) no se tiene en cuenta, sigue contando para el resto de la expresión. Por ejemplo, si queremos extraer la parte decimal de "11.22" podríamos pensar en hacer esto `/(?=\.)\d+/` , pero no funciona porque el . decimal no se "consume" con (?=\.), así que debemos tenerlo en cuenta y ponerlo detrás, así `/(?=\.)\.\d+/`

```
"11.22".match(/(?=\.)\d+/);          // Devuelve null
"11.22".match(/(?=\.)\.\d+/);        // Devuelve [".22"]
```

(?!expresion) hace lo contrario que **(?=expresion)**, es decir, busca una posición donde no se cumpla expresión. Por ejemplo, para sacar lo que no sean km de "11 km, 12 km, 14 m" podemos poner `/\d{2}(?! km)/`

```
"11 km 12 km 14 m".match(/\d{2}(?! km)/);    // Devuelve ["14"]
```

Flags de opciones

Hemos visto que una expresión regular es **/expresion/**. Podemos poner algunos flags detrás, básicamente unas letras que cambian algo el comportamiento

i es para ignorar mayúsculas y minúsculas

```
"hola".match(/HOLA/);    // Devuelve null
"hola".match(/HOLA/i);   // Devuelve ["hola"]
```

g es para buscar todas las veces posibles la expresión, no sólo una vez

```
"11 223 44 66 77".match(/\d+/);    // Devuelve ["11"]
"11 223 44 66 77".match(/\d+/g);   // Devuelve ["11", "223", "44", "66", "77"]
```

m busca en cadenas con retornos de carro \n considerando estos como inicios de linea ^ o fin $

```
"hola\ntu".match(/^tu/);     // Devuelve null
"hola\ntu".match(/^tu/m);    // Devuelve ["tu"]
"hola\ntu".match(/hola$/);   // Devuelve null
"hola\ntu".match(/hola$/m);  // Devuelve ["hola"]
```

Otros métodos de cadena y de expresión regular.

Para todos estos ejemplos hemos usado el método match() de la clase String, ya que nos devuelve un array con las cosas que se encuentran y viendo los resultados es la forma más clara de ver cómo funcionan las distintas partes de la expresión regular. Sin embargo, tanto String como RegExp tienen otros métodos útiles

String.search(/expresion/)

Devuelve la posición donde se encuentra esa expresión dentro de la cadena, o -1 si no se encuentra.

String.replace(/expresion/,cadena)

Busca el trozo de cadena que casa con la expresión y la reemplaza con lo que le pasemos en el parámetro cadena. Este método tiene además un detalle intresante. Cuando en la expresión regular tenemos paréntesis para extraer algunas partes de la cadena, la misma expresión regular recuerda qué ha encontrado. En el método replace, si en la cadena de segundo parámetro aparecen cosas como $1, $2, utilizará lo encontrado.

```
"ho3la".replace(/\d/,"X");          // Devuelve "hoXla"
"ho3la".replace(/(\d)/,"-$1-");     // Devuelve "ho-3-la"
```

String.match(/expresion/)

Ya lo hemos visto.

String.split(/expresion/)

Usa lo que sea que case con la expresión como separador y devuelve un array con la cadena partida en trozos por ese separador

```
"hola, dos tres; cuatro".split(/\W+/); // Devuelve ["hola", "dos", "tres", "cuatro"]
```

RegExp constructor

Además de crear las expresiones regulares con /expresion/flags, podemos hacerlo con un new de la clase RegExp, por ejemplo new RegExp("expresion","flags").

```
var reg = new RegExp("\\d+","g");
    "11 22 33".match(reg);  // Devuelve ["11","22","33"]
```

Hay que fijarse en este caso que las \ debemos escaparlas, con \\

RegExp.exec()

Es similar a match() de String, pero sólo devuelve un resultado y hace que RegExp guarde la posición en la que lo ha encontrado. Sucesivas llamadas a exec(), nos iran devolviendo los siguientes resultados

```
var reg = new RegExp("\\d+");
    reg.exec("11 22 33");    // Devuelve ["11"]
    reg.exec("11 22 33");    // Vuelve a devolver ["11"], puesto que no hay flag g
var reg = new RegExp("\\d+","g");
    reg.exec("11 22 33");    // Devuelve ["11"]
    reg.exec("11 22 33");    // Vuelve a devolver ["22"], puesto que si hay flag g
    reg.exec("11 22 33");    // Vuelve a devolver ["33"], puesto que si hay flag g
    reg.exec("11 22 33");    // Devuelve null, ya no hay más resutlados.
    reg.exec("11 22 33");    // Vuelve a devolver ["11"], despues de devolver null la RegExp se "reinicializa"
```

RegExp.test()

Similar a exec(), pero en vez de devolver lo encontrado, devuelve true si ha encontrado algo o false si no. Como la expresión regular recuerda las búsquedas anteriores, sucesivas llamadas a test() pueden devolver resultados distintos

```
var reg = new RegExp("\\d+","g");
    reg.test("11 22 33");    // Devuelve true, porque encuentra el ["11"]
    reg.test("11 22 33");    // Vuelve a devolver true, porque encuentra el ["22"], puesto que si hay flag g
    reg.test("11 22 33");    // Vuelve a devolver true, porque encuentra el ["33"], puesto que si hay flag g
    reg.test("11 22 33");    // Devuelve false, ya no hay más resutlados.
    reg.test("11 22 33");    // Vuelve a devolver true, porque vuelve a encontrar el ["11"], despues de devolver
                             null la RegExp se "reinicializa"
```

Ejercicio 2

Realizar el siguiente formulario, para introducir una contraseña de forma que nos devuelva si la contraseña introducida es correcta ó no.