# Resumen

**Expresiones regulares**

**Definición**

Las expresiones regulares permiten comprobar si una cadena de texto se ajusta a un determinado tipo de estructura o patrón. Utilizan un lenguaje de signos propio, y permiten también buscar o remplazar texto, no sólo un texto concreto sino un tipo de texto que se ajuste a un determinado patrón.

Las expresiones regulares no son exclusivas de javascript, las usan también otros lenguajes de programación, de hecho javascript las ha importado de ellos.

La clase RegExp posee varios métodos que veremos más adelante, los cuales permiten comprobar, buscar o remplazar los elementos del texto.

**Crear una expresión regular**

Las expresiones regulares van siempre escritas entre los caracteres: / ... / por lo que tenemos dos maneras de crearlas:

Mediante el método general de creación de objetos:

expresion = new RegExp( ... código\_expresion ... )

Asignando a una variable su valor:

expresion = / ...b código\_expresion... /

**Comprobar textos**

El caso más simple es comprobar si una cadena está contenida en otra:

expresion = /martes/

el método **test()** comprueba si la cadena está incluida en otra:

resultado = expresion.test("hoy es martes");

Se comprueba si la expresión regular está contenida en la cadena pasada como parámetro. si es así el resultado será el valor booleano true, y si no devolverá false.

Las expresiones regulares pueden comprobar también si la cadena de texto se ajusta a un determinado patrón o reglas o si posee un determinado grupo de caracteres.

expresion = /[aeiou]/;

Si se busca que la cadena contenga cualquiera de los caracteres de un grupo determinado, éstos se encerrarán entre corchetes. En el ejemplo se buscará en el texto cualquiera de las vocales y en ese caso se ajusta a patrón.

Los elementos dentro del corchete pueden formar grupos homogéneos, tales como cifras consecutivas o letras consecutivas del alfabeto, en ese caso ponemos la primera y la última y un guión en medio:

expresion = /[1-5]/; //equivale a /[12345]/

Si queremos incluir el guíón en la expresión anterior lo pondremos al final:

expresion = /[1-5-]/; //equivale a /[12345-]/

**Caracteres especiales**

Podemos incluir también una serie de caracteres especiales que indican si la cadena tiene un determinado tipo de caracteres.

* **\**t — Representa un tabulador.
* \r — Representa el "retorno de carro" o "regreso al inicio" o sea el lugar en que la línea vuelve a iniciar.
* \n — Representa la "nueva línea" el carácter por medio del cual una línea da inicio. Es necesario recordar que en [Windows](https://es.wikipedia.org/wiki/Windows) es necesaria una combinación de \r\n para comenzar una nueva línea, mientras que en [Unix](https://es.wikipedia.org/wiki/Unix) solamente se usa \n y en [Mac\_OS](https://es.wikipedia.org/wiki/Mac_OS) clásico se usa solamente \r.
* \a — Representa una "campana" o "beep" que se produce al imprimir este carácter.
* \e — Representa la tecla "Esc" o "Escape"
* \f — Representa un salto de página
* \v — Representa un tabulador vertical
* \x — Se utiliza para representar caracteres [ASCII](https://es.wikipedia.org/wiki/ASCII) o ANSI si conoce su código. De esta forma, si se busca el símbolo de derechos de autor y la fuente en la que se busca utiliza el conjunto de caracteres [latín-1](https://es.wikipedia.org/wiki/ISO/IEC_8859-1) es posible encontrarlo utilizando \xA9".
* \u — Se utiliza para representar caracteres [Unicode](https://es.wikipedia.org/wiki/Unicode) si se conoce su código. "\u00A2" representa el símbolo de centavos. No todos los motores de Expresiones Regulares soportan Unicode. El .Net Framework lo hace, pero el EditPad Pro no, por ejemplo.
* \d — Representa un dígito del 0 al 9.
* \w — Representa cualquier carácter [alfanumérico](https://es.wikipedia.org/wiki/Alfanum%C3%A9rico).
* \s — Representa un espacio en blanco.
* \D — Representa cualquier carácter que no sea un dígito del 0 al 9.
* \W — Representa cualquier carácter no alfanumérico.
* \S — Representa cualquier carácter que no sea un espacio en blanco.
* \A — Representa el inicio de la cadena. No un carácter sino una posición.
* \Z — Representa el final de la cadena. No un carácter sino una posición.
* \b — Marca la posición de una palabra limitada por espacios en blanco, puntuación o el inicio/final de una cadena.
* \B — Marca la posición entre dos caracteres alfanuméricos o dos no-alfanumérico

**El acento circunflejo "^"**

ste carácter tiene una doble funcionalidad, que difiere cuando se utiliza individualmente y cuando se utiliza en conjunto con otros caracteres especiales. En primer lugar su funcionalidad como carácter individual: el carácter "^" representa el inicio de la cadena (de la misma forma que el signo de dólar "$" representa el final de la cadena). Por tanto, si se utiliza la expresión regular "^[a-z]" el motor encontrará todos los párrafos que den inicio con una letra minúscula. Cuando se utiliza en conjunto con los corchetes de la siguiente forma "[^\w ]" permite encontrar cualquier carácter que NO se encuentre dentro del grupo indicado. La expresión indicada permite encontrar, por ejemplo, cualquier carácter que no sea alfanumérico o un espacio, es decir, busca todos los símbolos de puntuación y demás caracteres especiales.

La utilización en conjunto de los caracteres especiales **"^"** y **"$"** permite realizar validaciones en forma sencilla. Por ejemplo **"^\d$"** permite asegurar que la cadena a verificar representa un único dígito **"^\d\d/\d\d/\d\d\d\d$"** permite validar una fecha en formato corto, aunque no permite verificar si es una fecha válida, ya que 99/99/9999 también sería válido en este formato; la validación completa de una fecha también es posible mediante expresiones regulares, como se ejemplifica más adelante.

El asterisco "\*"

El asterisco sirve para encontrar algo que se encuentra repetido 0 o más veces. Por ejemplo, utilizando la expresión "[a-zA-Z]\d\*" será posible encontrar tanto "H" como "H1", "H01", "H100" y "H1000", es decir, una letra seguida de un número indefinido de dígitos. Es necesario tener cuidado con el comportamiento del asterisco, ya que este, por defecto, trata de encontrar la mayor cantidad posible de caracteres que correspondan con el patrón que se busca. De esta forma si se utiliza "\(.\*\)" para encontrar cualquier cadena que se encuentre entre paréntesis y se lo aplica sobre el texto "Ver (Fig. 1) y (Fig. 2)" se esperaría que el motor de búsqueda encuentre los textos "(Fig. 1)" y "(Fig. 2)", sin embargo, debido a esta característica, en su lugar encontrará el texto "(Fig. 1) y (Fig. 2)". Esto sucede porque el asterisco le dice al motor de búsqueda que llene todos los espacios posibles entre los dos paréntesis. Para obtener el resultado deseado se debe utilizar el asterisco en conjunto con el signo de interrogación de la siguiente forma: "\(.\*?\)" Esto es equivalente a decirle al motor de búsqueda que "Encuentre un paréntesis de apertura y luego encuentre cualquier secuencia de caracteres hasta que encuentre un paréntesis de cierre".

Las llaves "{}"

Comúnmente las llaves son caracteres literales cuando se utilizan por separado en una expresión regular. Para que adquieran su función de metacaracteres es necesario que encierren uno o varios números separados por coma y que estén colocados a la derecha de otra expresión regular de la siguiente forma: "\d{2}" Esta expresión le dice al motor de búsqueda que encuentre dos dígitos contiguos. Utilizando esta fórmula podríamos convertir el ejemplo "^\d\d/\d\d/\d\d\d\d$" que servía para validar un formato de fecha en "^\d{2}/\d{2}/\d{4}$" para una mayor claridad en la lectura de la expresión.

"\d{2,4}" Esta forma añade un segundo número separado por una coma, el cual indica al motor de búsqueda que como máximo debe aparecer 4 veces la expresión regular \d. Los posibles valores son:

"^\d\d$" (mínimo 2 repeticiones)

"^\d\d\d$"(tiene 3 repeticiones, por lo tanto entra en el rango 2-4)

"^\d\d\d\d$" (máximo 4 repeticiones)

Nota: aunque esta forma de encontrar elementos repetidos es muy útil, algunas veces no se conoce con claridad cuantas veces se repite lo que se busca o su grado de repetición es variable. En estos casos los siguientes metacaracteres son útiles.

el signo **/./** indica cualquier caracter, con lo que se suele usar para comprobar que la cadena no está vacía.

Para comprobar si la cadena tiene un punto utilizaremos el caracter de escape: **/\./**

los signos ^ al principio de la expresión regular y $ al final indican que la cadena de texto debe coincidir totalmente con la expresión regular y no sólo contenerla.

expresion = /^\d$/; //comprueba si una cadena es un número.

**Cuantificadores**

Los cuantificadores son signos que se ponen detrás de un determinado patrón de búsqueda, y que indican las veces que debe aparecer en el texto para ajustarse al patrón.

* /a+/ : la cadena debe contener el patron (letra "a") una o más veces.
* /a\*/ : Cero o más veces.
* /a?/ : Cero o una vez.
* /a{3}/ : Exactamente tres veces (en lugar del 3 puede ser cualquier otro número).
* /a{3,8}/ : Entre 3 y 5 veces (ambos inclusive).

**Modificadores**

Los modificadores o "flags" son unas letras con un significado especial que se ponen detrás de la expresión regular, y matizan la forma de buscar. Estos son los siguientes:

* **g** : /a/g ; Explora la cadena hasta el final.
* **i** : /a/i ; No distingue entre mayúsculas y minúsculas.
* **m** : /a/m ; Permite usar varios ^ y $ en la cadena.
* **s** : /a/s ; Incluye el salto de línea en el comodín punto .
* **x** : /a/x ; ignora los espacios en el patrón .

Si escribimos más de un modificador en una expresión regular, debemos ponerlos en el mismo orden que aparecen arriba, es decir, en orden alfabético.

**Métodos de Reg Exp**

Para comprobar, buacar o remplazar con expresiones regulares podemos usar métodos del objeto RegExp o métodos del objeto String (cadenas).

En los métodos de RegExp el método se aplica a la expresion regular, pasando la cadena como parámetro, mientras que en los métoos de String es al contrario.

Los métodos de RegExp son los siguientes:

**test()**

comprobar = expresion.test(cadena)

Ya visto anteriormente. Busca el patrón en el texto y devuelve el valor booleano true si el texto se ajusta o false si no se ajusta.

**compile()**

compilado = expresion.compile(expresion);

Convierte el patrón en un formato interno para que la ejecución sea más rápida. Por ejemplo, esto permite un uso más eficiente de expresiones regulares en bucles.

**exec()**

buscar = expresion.exec(cadena);

Busca la expresión en el texto, y devuelve el primer texto que concuerda con la expresión buscada. Si no encuentra ninguna coincidencia, entonces devuleve null.

Este método tiene las propiedades **index**, la cual nos indica la posición en la que se encuentra la cadena buscada, y la propiedad **input**, la cual devuelve la cadena completa en la que estamos realizando la búsqueda.

posicion = expresion.exec(cadena).index;  
textoCompleto = expresion.exec(cadena).input

La posicion en la propiedad index empieza a contarse desde el número 0. Si no se encuentra la cadena estas propiedades no devuelven nada.

**Métodos de String**

Los siguientes métodos se aplican a las cadenas de texto pero sirven también para trabajar con expresiones regulares, las cuales las pasamos como parámetros.

**search()**

buscar = cadena.search(expresion);

Devuelve el número del caracter donde encuentra la primera coincidencia (empieza a contar desde el 0). Si no hay coincidencia devuelve -1.

**split()**

miarray = cadena.split(expresion);

Transforma la cadena en un array, la expresión regular indica cual es el delimitador que separa los elementos del array.

**replace()**

nuevoTexto = cadena.replace(expresion,remplazo);

Devuelve la cadena original, en la que se ha remplazado las coincidencias encontradas con la

**Match(String)**

Busca en la cadena de entrada especificada la primera aparición de la expresión regular especificada.

**Propiedades de RegExp**

Estas son propiedades del propio objeto RegExp y no de los objetos concretos de la clase por lo que se escriben en la forma RegExp.propiedad. Sin embargo se actualizan automáticamente cada vez que usamos un método con una expresión regular, ya sea método de RegExp o método de String, correspondiendo el resultado al último método empleado.

**RegExp.$1** hasta **RegExp.$9**

índices que contienen las partes agrupadas con paréntesis en el patrón de búsqueda.

**RegExp.input;**

Cadena que se ha explorado

**cadena RegExp.lastmatch;**

Última coincidencia encontrada.

**RegExp.multiline;**

Variable booleana que indica si la cadena explorada incluye saltos de línea.

**RegExp.lastParent;**

Última coincidencia encontrada con un patrón entre paréntesis.

**RegExp.leftContext;**

Desde el principio de la cadena hasta la coincidencia hallada.

**RegExp.rightContext;**

Desde la coincidencia hasta el final de la cadena.

**Expresiones regulares habituales**

Se dan aquí algunos ejemplos de las expresiones regulares empleadas para las búsquedas más habituales.

**Comprobar un e-mail**

/^[\w]+@{1}[\w]+\.[a-z]{2,3}$/

**Comprobar una página web**

/^http[s]?:\/\/[\w]+([\.]+[\w]+)+$/

**Comprobar una nombre propio**

/^([A-ZÁÉÍÓÚ]{1}[a-zñáéíóú]+[\s]\*)+$/

**Comprobar un número de teléfono de España**

/^[9|6]{1}([\d]{2}[-]\*){3}[\d]{2}$/

Admite los formatos "000000000", ó 000-00-00-00, también el 000-000000, pero no el 000-000-000.

### 

### Resumen de las construcciones de expresiones regulares

|  |  |  |
| --- | --- | --- |
| **Patrón** | **Igualación a** | **Descripción** |
| x | el caracter x | Una letra cualquiera se iguala a la misma. |
| [abc] | letra a ó b ó c | cual quier palabra que contenga alguna de las letras dentro del corchete. |
| [^abc] | cualquier caracter excepto la 'a' 'b' 'c' | el operador ^ es de negación, aunque la palabra contenga la 'a', matcheara con las otras letras. |
| [a-zA-Z] | cualquier letra desde la 'a' hasta la 'z' | rango de valores desde la primera letra a la segunda en minúsculas y mayúsculas. |
| [a-d[m-p]] | cualquier letra desde la 'a' hasta la 'd' y también desde la 'm' a la 'p' | unión de rangos de valores distintos. |
| [a-z&&[def]] | cualquier letra desde la 'a' hasta la 'z' que sean iguales a 'd','e' y 'f' | intersección de rangos de letras. |
| [a-z&&[^bc]] | cualquier letra desde la 'a' hasta la 'z' excepto las letras 'b' y 'c' | Substracción de rango de letras. |
| [a-z&&[^m-p]] | cualquier letra de la 'a' a la 'z' excepto el rango de de 'm' a 'p' | idem anterior pero con un rango de letras. |
| . | Cualquier caracter (no solo letras) | \_Comodin para todos los caracteres. |
| \d | un digito de 0 a 9 | rango numerico decimal. |
| \D | un no digito de 0 a 9 (letras y caracteres) | negación de un rango númerico de 0 a 9. |
| \s | un espacio en blanco | similar a los tipos: [ \t\n\x0B\f\r] |
| \S | un NO espacio en blanco | la mayúscula es negación. |
| \w | similar a [a-zA-Z\_0-9] | un caracter alfanúmerico |
| \W | un NO caracter alfanúmerico | negacion de [^/w] |
| ^ | El principio de una linea | operador que indica agrega información al patrón |
| $ | final de la línea | operador que indica agrega información al patrón |
| \b | limite de una palabra | operador que indica agrega información al patrón |
| \B | un NO limite de una palabra | operador que indica agrega información al patrón |
| \A | el principio de un input | operador que indica agrega información al patrón |
| \G | el final del macheo anterior | operador que indica agrega información al patrón |
| \Z | el final de una linea (\n \r\n \r \u0085 \u2028 \u2029) | operador que indica agrega información al patrón |
| \z | el final de un input | operador que indica agrega información al patrón |
| X? | X, al menos una vez o ninguna vez | ? cuantificar una o ninguna |
| X\* | x cero veces o más | \* cuantificador cero o más |
| X+ | X al menos una vez o más veces | + cuantificador una o más |
| X{n} | X 'n' veces | exactamente el número de n veces |
| X{n,} | X al menos el número de n veces | cuantificador n veces o más |
| X{n,m} | X n veces y no mas de m veces | cuantificador de rango entre n y m veces (n<m) |

/Oh{3,6}\sno/