



# Expresiones regulares (RegExp)

Lenguaje JS CheatSheets YouTube Twitter

← Eventos en Javascript JSON →  
*Capítulo anterior* *Capítulo siguiente*

En muchas ocasiones nos encontraremos en la situación de querer realizar una o varias acciones si se cumple un determinado caso. Pero muchas veces, esos casos no son tan simples, sino que existe un número muy alto de situaciones diferentes que no podemos cubrir de formas tradicionales. En esas situaciones es donde las **expresiones regulares** quizás nos puedan ser de ayuda.

## ¿Qué es una RegExp?

Las **expresiones regulares** (*a menudo llamadas RegExp o RegEx*) son un sistema para buscar, capturar o reemplazar texto utilizando **patrones**. Estos patrones permiten realizar una búsqueda de texto de una forma relativamente sencilla y abstracta, de forma que abarca una gran cantidad de posibilidades que de otra forma sería imposible o muy costosa.

Constructor	Descripción
-------------	-------------



Constructor	Descripción
<small>REGEXP</small> <code>new RegExp(r, flags)</code>	Crea una nueva expresión regular a partir de <b>r</b> con los <b>flags</b> indicados.
<small>REGEXP</small> <code>/r/flags</code>	Simplemente, la expresión regular <b>r</b> entre barras <b>/</b> . <b>Notación preferida.</b>

Así pues, podríamos crear expresiones regulares de estas dos formas, siempre teniendo como notación preferida la primera:

```
// Notación literal (preferida)
const r = /.a.o/i;

// Notación de objeto
const r = new RegExp(".a.o", "i");
const r = new RegExp(/.a.o/, "i");
```

JS

En ambos ejemplos, estamos estableciendo la expresión regular **.a.o**, donde el **punto** (*como veremos más adelante*) es un comodín que simboliza cualquier carácter, y la **i** es un **flag** que establece que no diferencia mayúsculas de minúsculas.

En Javascript, se prefiere utilizar las barras **/** para delimitar una expresión regular en una variable. Se trata de una forma más cómoda y compacta que evita tener que hacer un **new** del objeto

REGEXP .

## Propiedades de una RegExp

Cada **expresión regular** creada, tiene unas propiedades definidas, donde podemos consultar ciertas características de la expresión

regular en cuestión. Además, también tiene unas propiedades de comprobación para saber si un flag determinado está activo o no:

Propiedades	Descripción
STRING <b>.source</b>	Devuelve un string con la expresión regular original al crear el objeto ( <i>sin flags</i> ).
STRING <b>.flags</b>	Devuelve un string con los flags activados en la expresión regular.
NUMBER <b>.lastIndex</b>	Devuelve la posición donde se encontró una ocurrencia en la última búsqueda.
BOOLEAN <b>.global</b>	Comprueba si el flag <b>g</b> está activo en la expresión regular.
BOOLEAN <b>.ignoreCase</b>	Comprueba si el flag <b>i</b> está activo en la expresión regular.
BOOLEAN <b>.multiline</b>	Comprueba si el flag <b>m</b> está activo en la expresión regular.
BOOLEAN <b>.unicode</b>	Comprueba si el flag <b>u</b> está activo en la expresión regular.
BOOLEAN <b>.sticky</b>	Comprueba si el flag <b>y</b> está activo en la expresión regular.

El funcionamiento de los **flags** los veremos en el apartado siguiente. No obstante, con las propiedades **.source** y **.flags** se puede obtener casi toda la información que se puede hacer con dichos flags.

```
const r = /reg/gi;  
  
r.source; // 'reg'  
r.flags; // 'ig'  
  
r.flags.includes("g"); // true (equivalente a r.global)  
r.flags.includes("u"); // false (equivalente a r.unicode)
```

JS



También podemos comprobar si un flag está activo con la propiedad `.flags` combinada con `includes()`, como se puede ver en el ejemplo anterior. Por otro lado, `.source` nos devuelve un `STRING` con la expresión regular definida (*y sin flags*). La utilidad de `lastIndex()` la veremos más adelante.

## Flags de una RegExp

El segundo parámetro del `new RegExp()` o el que se escribe después de la segunda barra `/` delimitadora del literal de las expresiones regulares, son una serie de caracteres que indican los **flags** activos en la expresión regular en cuestión:

```
const r1 = /reg/;  
const r2 = /reg/i;  
const r3 = /reg/gi;
```

JS

La expresión regular `r1` no tiene ningún **flag** activado, mientras que `r2` tiene el flag `i` activado y `r3` tiene el flag `i` y el flag `g` activado. Veamos para que sirve cada flag:

Flag	Booleano	Descripción
<code>i</code>	<code>.ignoreCase</code>	Ignora mayúsculas y minúsculas. Se suele denominar <b>insensible a mayús/minús</b> .
<code>g</code>	<code>.global</code>	Búsqueda global. Sigue buscando coincidencias en lugar de pararse al encontrar una.
<code>m</code>	<code>.multiline</code>	Multilínea. Permite a <code>^</code> y <code>\$</code> tratar los finales de línea <code>\r</code> o <code>\n</code> .



Flag	Booleano	Descripción
<b>u</b>	<b>.unicode</b>	Unicode. Interpreta el patrón como un código de una secuencia Unicode.
<b>y</b>	<b>.sticky</b>	Sticky. Busca sólo desde la posición indicada por <b>lastIndex</b> .

Cada una de estas flags se pueden comprobar si están activas desde Javascript con su booleano asociado, que es una propiedad de la expresión regular:

```
const r = /reg/gi;

r.global; // true
r.ignoreCase; // true
r.multiline; // false
r.sticky; // false
r.unicode; // false
```

JS

## Métodos de RegExp

Los objetos **RegExp** tienen varios métodos para utilizar expresiones regulares contra textos y saber si «casan» o no, es decir, si el patrón de la expresión regular encaja con el texto propuesto.

Método	Descripción
BOOLEAN <b>test(str)</b>	Comprueba si la expresión regular «casa» con el texto <b>str</b> pasado por parámetro.
ARRAY <b>exec(str)</b>	Ejecuta una búsqueda de patrón en el texto <b>str</b> . Devuelve un array con las capturas.

Por ejemplo, veamos como utilizar la expresión regular del ejemplo anterior con el método `test()` para comprobar si encaja con un texto determinado:

```
const r = /.a.o/i;  
  
r.test("gato"); // true  
r.test("pato"); // true  
r.test("perro"); // false  
r.test("DATO"); // true (el flag i permite mayús/minús)
```

El método `exec()` lo veremos un poco más adelante en el apartado de **captura de patrones**, ya que es algo más complejo. Primero debemos aprender que caracteres especiales existen en las expresiones regulares para dominarlas.

## Caracteres especiales

Antes de comenzar a utilizar **expresiones regulares** hay que aprender la parte más compleja de ellas: los caracteres especiales. Dentro de las expresiones regulares, existen ciertos caracteres que tienen un significado especial, y también, muchos de ellos dependen de donde se encuentren para tener ese significado especial, por lo que hay que aprender bien como funcionan.

### » Clases básicas

Empecemos con algunos de los más sencillos:

Caracter especial	Descripción
-------------------	-------------



Caracter especial	Descripción
.	Comodín, cualquier caracter.
\	Invierte el significado de un carácter. Si es especial, lo escapa. Si no, lo vuelve especial.
\t	Caracter especial. Tabulador.
\r	Caracter especial. Retorno de carro. A menudo denominado <b>CR</b> .
\n	Caracter especial. Nueva línea. A menudo denominado «line feed» o <b>LF</b> .

En esta pequeña tabla vemos algunos caracteres especiales que podemos usar en expresiones regulares. Observa que al igual que con otros tipos de datos, podemos utilizar el método **test()** sobre el literal de la expresión regular, sin necesidad de guardarla en una variable previamente:

```
// Buscamos RegExp que encaje con "Manz"
/M.nz/.test("Manz"); // true
/M.nz/.test("manz"); // false (La «M» debe ser mayúscula)
/M.nz/i.test("manz"); // true (Ignoramos mayús/minús con el f

// Buscamos RegExp que encaje con "A."
/A./.test("A."); // true (Ojo, nos da true, pero el punto es
/A./.test("Ab"); // true (Nos da true con cualquier cosa)
/A\./.test("A."); // true (Solución correcta)
/A\./.test("Ab"); // false (Ahora no deja pasar algo que no se
```

## » Conjunto de caracteres o alternativas

Dentro de las expresiones regulares los corchetes **[]** tienen un significado especial. Se trata de un mecanismo para englobar un



**conjunto de caracteres personalizado.** Por otro lado, si incluimos un circunflejo `^` antes de los caracteres del corchete, invertimos el significado, pasando a ser que **no exista** el conjunto de caracteres personalizado:

Caracter especial	Descripción
<code>[]</code>	Rango de caracteres. Cualquiera de los caracteres del interior de los corchetes.
<code>[^]</code>	No exista cualquiera de los caracteres del interior de los corchetes.
<code> </code>	Establece una alternativa: lo que está a la izquierda o lo que está a la derecha.

Por último, tenemos el «pipe» `|`, con el que podemos establecer alternativas. Veamos un ejemplo aplicado a esto, que se verá más claro:

```
const r = /[aeiou]/i; // RegExp que acepta vocales (mayús/minús JS)
r.test("a"); // true (es vocal)
r.test("E"); // true (es vocal, y tiene flag «i»)
r.test("t"); // false (no es vocal)

const r = /^[^aeiou]/i; // RegExp que acepta lo que no sea vocal
r.test("a"); // false
r.test("E"); // false
r.test("T"); // true
r.test("m"); // true

const r = /casa|cama/; // RegExp que acepta la primera o la seg
r.test("casa"); // true
r.test("cama"); // true
r.test("capa"); // false
```



En el interior de los corchetes, si establecemos dos caracteres separados por guión, por ejemplo `[0-9]`, se entiende que indicamos el rango de caracteres entre `0` y `9`, sin tener que escribirlos todos explícitamente.

De esta forma podemos crear rangos como `[A-Z]` (*mayúsculas*) o `[a-z]` (*minúsculas*), o incluso varios rangos específicos como `[A-Za-z0-9]`:

Caracter especial	Alternativa	Descripción
<code>[0-9]</code>	<code>\d</code>	Un dígito del 0 al 9.
<code>[^0-9]</code>	<code>\D</code>	No exista un dígito del 0 al 9.
<code>[A-Z]</code>		Letra mayúscula de la <b>A</b> a la <b>Z</b> . Excluye <b>ñ</b> o letras acentuadas.
<code>[a-z]</code>		Letra minúscula de la <b>a</b> a la <b>z</b> . Excluye <b>ñ</b> o letras acentuadas.
<code>[A-Za-z0-9]</code>	<code>\w</code>	Carácter alfanumérico (letra mayúscula, minúscula o dígito).
<code>[^A-Za-z0-9]</code>	<code>\W</code>	No exista carácter alfanumérico (letra mayúscula, minúscula o dígito).
<code>[\t\r\n\f]</code>	<code>\s</code>	Carácter de espacio en blanco (espacio, <b>TAB</b> , <b>CR</b> , <b>LF</b> o <b>FF</b> ).
<code>[^\t\r\n\f]</code>	<code>\S</code>	No exista carácter de espacio en blanco (espacio, <b>TAB</b> , <b>CR</b> , <b>LF</b> o <b>FF</b> ).
	<code>\xN</code>	Carácter hexadecimal número <b>N</b> .
	<code>\uN</code>	Carácter Unicode número <b>N</b> .

Observa que en esta tabla tenemos una notación **alternativa** que es equivalente al caracter especial indicado. Por ejemplo, es lo mismo

escribir `[0-9]` que `\d`. Algunos programadores encuentran más explicativa la primera forma y otros más cómoda la segunda.

## » Anclas

Dentro de las expresiones regulares, las **ancclas** son un recurso muy importante, ya que permiten delimitar los patrones de búsqueda e indicar si empiezan o terminan por caracteres concretos, siendo mucho más específicos al realizar la búsqueda:

Caracter especial	Descripción
<code>^</code>	Ancla. Delimita el inicio del patrón. Significa <b>empieza por</b> .
<code>\$</code>	Ancla. Delimita el final del patrón. Significa <b>acaba en</b> .
<code>\b</code>	Posición de una palabra limitada por espacios, puntuación o inicio/final.
<code>\B</code>	Opuesta al anterior. Posición entre 2 caracteres alfanuméricos o no alfanuméricos.

Las dos primeras son bastante útiles cuando sabemos que el texto que estamos buscando termina o empieza de una forma concreta. De este modo podemos hacer cosas como las siguientes:

```
const r = /^mas/i;

r.test("Formas"); // false (no empieza por "mas")
r.test("Master"); // true
r.test("Masticar"); // true

const r = /do$/i;

r.test("Vívido"); // true
r.test("Dominó"); // false
```

JS



Por otro lado, `\b` nos permite indicar si el texto adyacente está seguido o precedido de un límite de palabra (*espacio*), puntuación (*comas o puntos*) o inicio o final del `STRING` :

```
const r = /fo\b/;

r.test("Esto es un párrafo de texto."); // true (tras "fo" hay
r.test("Esto es un párrafo."); // true (tras "fo" hay un signo
r.test("Un círculo es una forma."); // false (tras "fo" sigue l
r.test("Frase que termina en fo"); // true (tras "fo" termina e
```

Por último, `\B` es la operación opuesta a `\b`, por lo que podemos utilizarla cuando nos interesa que el texto no esté delimitado por una palabra, puntuación o string en sí.

## » Cuantificadores

En las **expresiones regulares** los cuantificadores permiten indicar cuántas veces se puede repetir el carácter inmediatamente anterior. Existen varios tipos de cuantificadores:

Caracter especial	Descripción
<code>*</code>	El carácter anterior puede aparecer <b>0</b> o más veces.
<code>+</code>	El carácter anterior puede aparecer <b>1</b> o más veces.
<code>?</code>	El carácter anterior puede aparecer o no aparecer.
<code>{n}</code>	El carácter anterior aparece <b>n</b> veces.
<code>{n,}</code>	El carácter anterior aparece <b>n</b> o más veces.



Caracter especial	Descripción
<code>{n,m}</code>	El carácter anterior aparece de <b>n</b> a <b>m</b> veces.

Veamos algunos ejemplos para aprender a aplicarlos. Comencemos con **\*** (*0 o más veces*):

```
// 'a' aparece 0 o más veces en el string
const r = /a*/;

r.test(""); // true ('a' aparece 0 veces)
r.test("a"); // true ('a' aparece 1 veces)
r.test("aa"); // true ('a' aparece 2 veces)
r.test("aba"); // true ('a' aparece 2 veces)
r.test("bbb"); // true ('a' aparece 0 veces)
```

JS

El cuantificador **+** es muy parecido a **\***, sólo que con el primero es necesario que el carácter anterior aparezca al menos una vez:

```
// 'a' aparece 1 o más veces (equivalente a /aa*/)
const r = /a+/;

r.test(""); // false ('a' aparece 0 veces)
r.test("a"); // true ('a' aparece 1 veces)
r.test("aa"); // true ('a' aparece 2 veces)
r.test("aba"); // true ('a' aparece 2 veces)
r.test("bbb"); // false ('a' aparece 0 veces)
```

JS

El cuantificador **?** se suele utilizar para indicar que el carácter anterior es opcional (*puede aparecer o puede no aparecer*). Normalmente se utiliza cuando quieres indicar que no importa que aparezca un carácter opcional:



```
const r = /disparos?/i;
```

JS

```
r.test("Escuché disparos en la habitación."); // true  
r.test("Efectuó un disparo al sujeto."); // true  
r.test("La pistola era de agua."); // false
```

Los tres cuantificadores siguientes, se utilizan cuando necesitamos concretar más el número de repeticiones del caracter anterior. Por ejemplo, **{n}** indica un número exacto, **{n,}** indica al menos **n** veces y **{n,m}** establece que se repita de **n** a **m** veces.

```
// Un número formado de 2 dígitos (del 0 al 9)  
const r = /[0-9]{2}/;
```

JS

```
r.test(42); // true  
r.test(88); // true  
r.test(1); // false  
r.test(100); // true
```

Observa que el último aparece como **true**. Esto ocurre porque en la expresión regular no se han establecido **anclas** que delimiten el inicio y/o el final del texto. Si las añadimos, es más estricto con las comprobaciones:

```
const r = /^[0-9]{2}$/;
```

JS



```
r.test(4); // false  
r.test(55); // true  
r.test(100); // false
```

```
const r = /^[0-9]{3,}$/;
```

```
r.test(33); // false  
r.test(4923); // true
```



```
const r = /^[0-9]{2,5}$/;  
  
r.test(2); // false  
r.test(444); // true  
r.test(543213); // false
```

Si quieres profundizar con las **expresiones regulares**, puedes jugar a [RegEx People](#) , un pequeño y básico juego para aprender a utilizar las expresiones regulares y buscar patrones, con su código fuente disponible en [GitHub](#) .

Recuerda también que aunque **test()** espera un STRING por parámetro, en caso de enviarle otro objeto, lo pasará a STRING mediante el método **toString()** que existe en todos los objetos de Javascript

## Captura de patrones

Pero con las **expresiones regulares** no sólo podemos realizar búsquedas de patrones. Una de las características más importantes de las expresiones regulares es lo potente y versátil que resultan las **capturas de patrones**.

Toda expresión regular que utilice la **parentización** (*englobe con paréntesis fragmentos de texto*) está realizando implícitamente una captura de texto, que es muy útil para obtener rápidamente información.

Para ello, dejamos de utilizar el método **test(str)** y comenzamos a utilizar **exec(str)**, que funciona exactamente igual, sólo que devuelve un array con las capturas realizadas. Antes de empezar a utilizarlo, necesitamos saber detalles sobre la **parentización**:



Caracter especial	Descripción
(x)	El patrón incluido dentro de paréntesis se captura y se guarda en <b>\$1</b> o sucesivos.
(?:x)	Si incluimos <b>?:</b> al inicio del contenido de los paréntesis, evitamos capturar ese patrón.
x(?:y)	Busca sólo si <b>x</b> está seguido de <b>y</b> .
x(?:!y)	Busca sólo si <b>x</b> no está seguido de <b>y</b> .

Así pues, vamos a realizar una captura a través de los paréntesis de una expresión regular:

```
// RegExp que captura palabras de 3 letras.
const r = /\b([a-z]{3})\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que

r.global; // true (el flag global está activado)

r.exec(str); // ['una', 'una']      index: 35
r.exec(str); // ['que', 'que']      index: 46
r.exec(str); // ['ver', 'ver']      index: 60
r.exec(str); // ['que', 'que']      index: 64
r.exec(str); // null
```

El método **exec()** nos permite ejecutar una búsqueda sobre el texto **str** hasta encontrar una coincidencia. En ese caso, se detiene la búsqueda y nos devuelve un array con los `STRING` capturados por la parentización. Si el flag **g** está activado, podemos volver a ejecutar **exec()** para continuar buscando la siguiente aparición, hasta que no encuentre ninguna más, que devolverá **null**.

## » RegEx en Strings

Quizás, generalmente el usuario prefiera utilizar el método `match(reg)` de los `STRING`, que permiten ejecutar la búsqueda de la expresión regular `reg` pasada por parámetro, sobre esa variable de texto. El resultado es que nos devuelve un `ARRAY` con los `STRING` capturados:

```
const r = /\b([a-z]{3})\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que";
str.match(r); // Devuelve ['una', 'que', 'ver', 'que']

const r = /\bv([0-9]+)\.([0-9]+)\.([0-9]+)\b/;
const str = "v1.0.21";
str.match(r); // Devuelve ['v1.0.21', '1', '0', '21']
```

En el caso de no existir **parentización**, el array devuelto contiene un `STRING` con todo el texto capturado. En el caso de existir múltiples parentizaciones (como en el último ejemplo), el array devuelto contiene un `STRING` con todo el texto capturado, y un `STRING` por cada parentización.

Recuerda que los `STRING` tienen varios métodos que permiten el uso de expresiones regulares para realizar operaciones, como por ejemplo, el `replace()`, para hacer reemplazos en todas las ocurrencias:

```
const daenerys = "Javascript es un gran lenguaje";
daenerys.replace(/[aeou]/g, "i"); // 'Jiviscript is in grin lin'
```



Eventos en Javascript

Capítulo anterior



JSON

Capítulo siguiente







# CheatSheets

&lt;HTML&gt;

{CSS}

{JS}

{vue}

\$ terminal

&lt;web components&gt;



functions

Eventos en  
Javascript

JSON

Clases |



## Publicado por Manz

Docente, divulgador informático y freelance. Autor de [Emezeta.com](https://emezeta.com), es profesor en la [Universidad de La Laguna](https://www.universidadde.la.laguna.es/) y dirige el curso de Programación web FullStack y Diseño web FrontEnd de [EOI](https://www.eoi.es/) en Tenerife (Canarias). En sus ratos libres, busca GIF de gatos en Internet.



Twitter



Linkedin



GitHub



CodePen



YouTube

3 comentarios

## Lenguaje JS

Documentación sobre Javascript, su evolución y las mejores herramientas y recursos alrededor de su ecosistema.

Creado y mantenido por [@Manz](https://manz.es/) con ❤️

¡Puedes sugerir temas en el [backlog](#)!

Credits: [Main image: ©ESA/Hubble modified](#)



2.05g. de pistachos con sirope de fresa.

CLS 0 FCP ... FID ... LCP ... TTFB 212ms

