UNIDAD 9 (2). Formularios - Expresiones regulares

Desarrollo Web en Entorno Cliente

2º DAW

Contenidos

Expresiones regulares

Las expresiones regulares son un elemento de uso muy habitual en la mayoría de lenguajes de programación. Se utilizan principalmente, para definir patrones que reconocen cadenas de caracteres específicas. Estas condiciones nos facilitan la tarea de búsqueda de textos dentro de otros textos, validación de formularios o extracción avanzada de subcadenas.

Las expresiones regulares de JavaScript son objetos de tipo **`RegExp`** que se pueden crear delimitándolas entre dos barras (*/*) al principio y al final. Tras las barras se pueden indicar controles o banderas para crear expresiones regulares avanzadas. También se pueden crear utilizando su constructor. La sintaxis sería la siguiente:

```
let expReg = /expresión/[controles];
let expReg = new RegExp("expresión", [controles]);
```

Una expresión regular se puede crear a través de su literal o. Ejemplo:

```
let expReg1 = /\w+/;
let expReg2 = new RegExp("\\w+");
```

Métodos para trabajar con expresiones regulares

Las expresiones regulares se utilizan fundamentalmente con dos métodos de la clase **`String`**:

Método	Devuelve	Descripción
`cadena.match(/expReg/)`	`[String]`	Devuelve un array con primer substring que casa o `null`. Algunos navegadores devuelven un array con más parámetros, pero conviene utilizar solo el primero.
`cadena.replace(/expReg/, repuesto)`	`String`	Devuelve un string resultado de sustituir el primer substring que casa por *repuesto*.

Ejemplos:

```
"Es hoy".match(/hoy/) => devuelve ["hoy"]
"Número: 142719".replace(/1/, "x") => devuelve "Número: x42719"
```

Cuando se validan campos de formularios, es muy frecuente tener que comprobar si una cadena tiene un formato determinado. Esto se puede hacer comprobando si una cadena casa con una expresión regular utilizando el siguiente método de la clase **`RegExp`**:

Método	Devuelve	Descripción
`expReg.test(cadena)`	`Boolean`	Devuelve `true` si la cadena casa con la expresión regular, `false` en otro caso.
`exReg.exec(cadena)`	`[String]`	Devuelve un array con primer substring que casa o `null`. Se comporta como `match()`.

Ejemplo:

```
let expReg = /[0-9]{8}[a-zA-Z]/
expReg.test("012345678A")   => devuelve true
```

Construcción de expresiones regulares

Algunos patrones básicos

Carácter	Significado
c	Carácter. Siendo c un carácter cualquiera, encaja solamente con ese carácter.
cde	Secuencia. Siendo c, d, y e tres caracteres cualquiera, encaja solamente si esos caracteres aparecen de esa manera. Ejemplo: `/hola/` encaja con textos que contengan la palabra hola
^	Principio de cadena. Ejemplo: `/^A/` encaja con la "A" en "Asturias" pero no encaja con la "A" en "Es Asturias".
$	Final de cadena. Ejemplo: `/s$/` encaja con última "s" en "Asturias" pero no con la primera (segunda letra).
.	Cualquier carácter excepto fin de línea.

Ejemplos:

```
"Es hoy".match(/hoy$/) => devuelve ['hoy'] (está al final)
"Es hoy".match(/^hoy/) => devuelve null (no está al principio)
"Es hoy".match(/^..../) => devuelve los 4 primeros caracteres:
['Es h']
```

Clases y rangos de caracteres

Carácter	Significado
[xyz]	Clase de caracteres. Encaja con uno de los caracteres que están entre los corchetes. Ejemplos: `/chic[ao]/` cadenas `"chico"` y `"chica"`. `/[aeiou]/` cualquier vocal (minúscula).
[^xyz]	Clase de caracteres negada. Encaja con un carácter que NO esté entre los corchetes. Ejemplo: `/[^aeiou]/` no debe ser vocal (minúscula).
[a-z]	Rango de caracteres. En este corresponde al rango “a-z” de letras ASCII. Ejemplo: `/[0-9][0-9][0-9]/` las cadenas que contienen tres números seguidos, como `"123"` y `"676"` encajan con el patrón.
[^a-z]	Rango de caracteres negado. Ejemplo: `/[^c-z]a/` las cadenas `"aa"` y `"ba"` encajan, pero la cadena `"za"` no encaja.

Ejemplos:

```
"canciones".match(/[aeiou]/) => devuelve ['a']
"canciones".match(/c[aeiou]/) => devuelve ['ca']
"canciones".match(/n[aeiou]/) => devuelve ['ne']
```

Otras clases y rangos de caracteres

Carácter	Significado
\d	Cualquier dígito. Equivale a `[0-9]`. Ejemplo: `/\d/` o `/[0-9]/` encaja con "2" en "B2".
\D	Cualquier carácter que no sea un dígito. Equivale a `[^0-9]`. Ejemplo: `/\D/` o `/[^0-9]` encaja con "B" en "B2".
\w	Cualquier carácter alfanumérico, incluido el subrayado. Equivale a [A-Za-z0-9_]. Ejemplo: `/\w/` encaja con "a" en "apple", "5" en "$5.28" y "3" in "3D".
\W	Cualquier carácter que no es un carácter alfanumérico o subrayado. Equivale a [^A-Za-z0-9_] Ejemplo: `/\W/` encaja con "." y con "$" en "$5.28".
\s	Cualquier carácter de tipo espacio en blanco, como espacio en blanco, tabulador, salto de línea. Reconoce separadores [\f\n\r\t\v\u00a0\u1680 .....]. Ejemplo: `/\s\w*/` encaja con " bar" en "foo bar".
\S	Cualquier carácter que no sea de tipo espacio en blanco.
\t	Tabulador horizontal.
\r	Salto de línea.
\n	Nueva línea.
\v	Tabulador vertical.
\f	Salto de página.
****	Para escapar un carácter que tiene un significado especial, como [,], /, \, etc. Por ejemplo \\ es la forma de representar la barra invertida.
\uffff	Permite indicar un carácter Unicode mediante su código hexadecimal.
\p{PropUnicode}	Cualquier carácter que pertenezca a la propiedad Unicode indicada.
\P{PropUnicode}	Cualquier carácter que no pertenezca a la propiedad Unicode.

Controles i, g, m

Carácter	Significado
i	Búsqueda insensible a mayúsculas.
g	Búsqueda global de todos los substrings que casan con el patrón (no solamente con el primero).
m	Búsqueda multilínea, donde ^ y $ representan principio y fin de línea.

Ejemplos con match():

```
"canciones".match(/[aeiou]/g) => devuelve ['a', 'i', 'o', 'e']
"canciones".match(/c[aeiou]/g) => devuelve ['ca', 'ci']
"Hoy dice hola".match(/ho/i) => devuelve ['Ho']
"Hoy dice hola".match(/ho/ig) => devuelve ['Ho', 'ho']
"Hola Pepe\nHoy vás".match(/^Ho/g) => devuelve ['Ho']
"Hola Pepe\nHoy vás".match(/^ho/gim) => devuelve ['Ho', 'Ho']
```

Ejemplos con replace():

```
"Número: 142719".replace(/1/, 'x') => devuelve 'Número: x42719'
"Número: 142719".replace(/1/g, 'x') => devuelve 'Número: x427x9'
```

```
"Número: 142719".replace(/[0-9]+/, '<número>') => devuelve
'Número: <número>'
```

Operadores de repetición

Carácter	Significado
*	Cero o más veces. Ejemplo: `/a*/` encaja con: “”, “a”, “aa”, “aaa”, ...
+	Una o más veces. Ejemplo: `/a+/` encaja con: “a”,“aa”, “aaa”, …
?	Cero o una vez. Ejemplo: `/a?/` encaja solo con: “” y “a”
{n}	n veces. Ejemplo: `/a{2}/` encaja solo con: “aa”
{n,}	n o más veces. Ejemplo: `/a{2,}/` encaja con: “aa”, “aaa”, “aaaa”, ...
{n, m}	Entre n y m veces. Ejemplo: `/a{2,3}/` encaja solo con: “aa” y “aaa”

Ejemplos:

```
"tiene".match(/[aeiou]+/g) => devuelve ['ie', 'e']
// cadenas no vacías de vocales
"tiene".match(/[aeiou]?/g) => devuelve ['', 'i', 'e', '', 'e', '']
// vocal o nada
"tiene".match(/[aeiou]*/g) => devuelve ['','ie', '', 'e', '']
// cadenas de vocales incluyendo ""
"Había un niño.".match(/[a-zñáéíóú]+/ig) => devuelve ['Había',
'un', 'niño']
// palabras en castellano: ASCII extendido con ñ, á, é, í, ó, ú
```

Los operadores de repetición son “ansiosos” y siempre buscarán encajar o casar con la cadena más larga posible, pero pueden volverse “perezosos” añadiendo `?` detrás del operador de repetición, en ese caso casarán con la cadena más corta posible.

Ejemplos:

```
"aaabb".match(/a+/) => devuelve ['aaa']
"aaabb".match(/a+?/) => devuelve ['a']
"ccaaccbccaa".match(/.+cc/) => devuelve ['ccaaccbcc']
"ccaaccbccaa".match(/.+?cc/) => ['ccaacc']
```

Estos dos patrones se utilizan mucho:

- `.+` cualquier cadena con longitud uno o más
- `.*` cualquier cadena con longitud cero o más

Patrones alternativos

Carácter	Significado

x|y	Encaja con x o con y. Ejemplo: `/green|red/` encaja con "green" en "green apple" y con "red" in "red apple".

Ejemplos:

```
"canciones".match(/ci|ca/) => devuelve ['ca']
"canciones".match(/ci|ca/g) => devuelve ['ca','ci']
"1 + 2 --> tres".match(/[a-z]+|[0-9]+/g) => devuelve ['1','2','tres']
```

Subpatrones

Carácter	Significado
(x)	Subpatrón. Encaja con x y recuerda la cadena encajada. El patrón y los subpatrones pueden reutilizarse referenciándolos con: o $0 representa todo el patrón o $1 representa el primer subpatrón o $2 el match el segundo subpatrón y así sucesivamente Ejemplos: `/(c)([aeiou])/` patrón c seguido de vocal, delimitando la c como primer subpatrón ($1) y las vocales como segundo subpatrón ($2) `/([0-9]+)(,[0-9]*)?/` patrón que representa números enteros o decimales, tiene dos subpatrones `([0-9]+)` y `(,[0-9]*)`

El método `match()` realmente busca patrones y subpatrones, por lo tanto, `cadena.match(/patrón/)` devolverá el match completo seguido de los subpatrones: `[match$0, macth$1, match$2,…]`

Ejemplos con match():

```
"canciones".match(/(c)([aeiou])/) => devuelve ['ca','c','a']
"canciones".match(/c([aeiou])n/) => devuelve ['can','a']
"canciones".match(/(..)..(..)/) => devuelve ['cancio','ca','io']
```

Ejemplos con replace():

```
//Sustituye por el primer subpatrón
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '$1') => devuelve
'Número: 142'
//Sustituye por 0 y segundo supbatrón
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '0$2') =>
devuelve 'Número: 0,719'
//Sacamos la , del subpatrón para que la sustituya por el .
"Número: 142,719".replace(/([0-9]+),([0-9]*)?/, '$1.$2') =>
devuelve 'Número: 142.719'
```

Ejemplo con test() y exec():

```
//Patrón que representa un código postal (formado por 5 números
del 00000 al 52999)
let cp = /^((5[012])|([0-4][0-9]))([0-9]{3})$/;
cp.test("49345") => devuelve true
```

```
cp.test("53345") => devuelve false
cp.exec("49345") => devuelve ['49345']
cp.exec("53345") => devuelve null
```