1 Expresiones Regulares con JavaScript

Una expresión regular es un objeto que describe un patrón de caracteres. En JavaScript, las expresiones regulares se representan mediante objeto **RegExp**. Evidentemente, estos objetos se pueden crear con la constructora **RegExp()** pero normalmente se crean utilizando una sintaxis especial. Del mismo modo que los literales de cadena son caracteres especificados entre comillas, los literales de las expresiones regulares se especifican como caracteres dentro de un par de barras inclinadas (/). Por tanto, nuestro código JavaScript podría contener líneas como esta:

```
var patron = /[A-Z]+$/;
```

Otro modo de definirla:

```
var patron = new RegExp(?[A-Z]+$?);
```

La tarea más difícil de definir una expresión regular es, justamente, describir el patrón de caracteres deseado utilizando la sintaxis de las expresiones regulares (puedes ver una pequeña ayuda en el siguiente enlace).

> **Para seguir este apartado sobre Expresiones Regulares configura una pequeña web con los siguientes elementos: una caja de texto para introducir la expresión regular, una caja del tipo** textarea **para introducir el texto de trabajo, un botón para aplicar el patrón sobre el texto de trabajo y una etiqueta donde se mostrará la salida del programa/función.**

1.1 Sumario

1.2 Indicadores

Además de la gramática de las expresiones regulares, existe un elemento más, **los indicadores**, que especifican reglas de coincidencia de patrones. Veamos los existentes y qué realizan:

- **i** : Ejecuta una coincidencia que no distingue entre minúsculas y mayúsculas.
- **g** : Ejecuta una coincidencia global, es decir, encuentra todas las coincidencias en lugar de detenerse tras la primera.
- **m** : Modo de múltiples líneas. Recordar que **^** coincide con el principio de una línea o de una cadena y **$** coincide con el final de una línea o de una cadena.

La expresión regular, con indicadores, se definiría, por ejemplo, del modo que vemos en el siguiente código, donde se define una expresión regular para buscar todas las palabras que empiezan por ?**c**? y terminan por ?**a**? en un documento multilínea y buscando todas las coincidencias sin diferenciar entre mayúsculas y minúsculas:

```
let strMyReg = ?\bc\w*a\b?;
let myRe = new RegExp(expreReg, "gmi");
```

Otro modo de hacerlo sería el siguiente:

```
let myRe = /\bc\w*a\b/gmi;
```

1.3 Métodos de cadena para coincidencia de patrones

Los objetos **String** admiten cuatro métodos que utilizan expresiones regulares.

1.3.1 search()

El más simple es **search()**. Este método acepta un argumento de expresión regular y devuelve la posición del carácter inicial de la primera subcadena coincidente, o **-1** si no existe ninguna coincidencia. Por ejemplo, la siguiente llamada devuelve 4:

```
?JavaScript?.search(/script/i);
```

Decir que, **search()** no admite búsquedas globales.

1.3.2 replace()

Otro método que tenemos es replace(), este método ejecuta una operación de búsqueda y reemplazo. Acepta una expresión regular como primer argumento y una cadena de reemplazo como segundo argumento. Si la expresión regular tiene establecido el indicador **g**, el método **replace()** reemplaza todas las coincidencias de la cadena con la cadena de reemplazo; en caso contrario, sólo reemplaza la primera coincidencia que encuentra.

```
var texto = `Si quieres hacer en javascript
un texto multilinea y no te acuerdas,
utiliza las comillas javascript inclinadas`
texto.replace(/javascript/gi, ?JavaScript?);
```

1.3.3 match()

El método **match()** es el más general de los métodos de expresión regular del objeto **String**. Acepta una expresión regular como su único argumento y devuelve una matriz que contiene el resultado de la coincidencia. Si se establece el indicador **g**, el método devuelve una matriz con todas las coincidencias que aparecen en la cadena.

En el siguiente ejemplo se guardan en un *array* los tres números existentes en el texto:

```
var texto = `El número 1,
o el número 20, o, como no
el número 323.`
console.log(texto.match(/\d+/g));     //[?1?, ?20?, ?323?]
```

Recordar agregar el indicador **g** para que se realice una búsqueda global y, así, cada elemento de la matriz será una coincidencia total. Si no se añade ese indicador **g**, el primer elemento de la matriz será la coincidencia completa y, los demás elementos de la matriz serán resultados de las subexpresiones entre paréntesis de la completa.

1.3.4 split()

El último método de expresión regular del objeto **String** es **split()**, ya conocido por nosotros. Este método desglosa la cadena en una matriz de subcadenas, utilizando el argumento como separador. Por ejemplo:

```
?123, 456, 789?.split(?,?); //Devuelve [?123?, ?456?, ?789?]
```

El método **split()** también acepta una expresión regular como argumento. Así es mucho más eficaz. En el siguiente ejemplo se especifica una **?,?** por separador con un número arbitrario de espacios en blanco en ambos lados:

```
?1, 2,  3, 4,   5?.split(/\s*,\s*/);
//Devuelve [?1?,?2?,?3?,?4?,?5?]
```

1.4 Objeto RegExp

Como sabemos, las expresiones regulares se representan como objetos **RegExp**. Así, tenemos, la función constructora **RegExp()**, tres métodos y

diversas propiedades que veremos ahora.

Podemos ver un ejemplo de la constructora **RegExp()**, donde se buscan los números de cinco dígitos en una cadena. Tener en cuenta que las cadenas y las expresiones regulares utilizan el carácter \ para secuencias de escape por lo que, cuando se pasa una expresión regular a **RegExp()** como literal de cadena, hay que reemplazar cada carácter \ por \\.

```
let patron = new RegExp(?\\d{5}?, ?g?);
```

¡Ojo!, porque si la expresión regular viene de una variable no hará falta esa doble \\. Ese detalle es interesante si leemos la expresión regular de una caja de texto o similar.

```
let regex = "\d{5}";
let patron = new RegExp(regex, ?g?);
```

1.4.1 Métodos RegExp para coincidencia de patrones

Los objetos **RegExp** definen dos métodos que ejecutan operaciones de coincidencia de patrones.

1.4.1.1 exec()

El método principal de coincidencia de patrones de **RegExp** es **exec()**, que es similar al método **match()** de **String**. En este caso, si no existe ninguna coincidencia, devuelve **null**. Si encuentra alguna coincidencia, devuelve un array con un comportamiento igual al método **match()** de **String**.

Así y todo, para recuperar con **exec()** todas las coincidencias tal y como se realizó con **match()** tendríamos que hacer del siguiente modo:

```
let texto = `El número 1,
o el número 20, o, como no
el número 323`;
let patron = new RegExp("\\d+", "g");
let arraySal = new Array();
let arrayMatch = new Array(); //Este array tendrá todas las coincidencias
let sal = "";

while ((arraySal = patron.exec(texto)) != null) {
    arrayMatch.push(arraySal[0]); //Vamos creando el array con todas las coincidencias
sal +=
" - Coincidencia : " +
arraySal[0] +
" --> en posición : " +
arraySal.index +
"\n";
}
console.log(sal);
console.log(arrayMatch);
```

1.4.1.2 test()

El otro método de **RegExp()** es **test()**. Este método acepta una cadena y devuelve **true** si ésta contiene una coincidencia para la expresión regular. Podemos aquí ver un ejemplo:

```
salida = patron.test(texto);
```

Con **test()** podemos ir recorriendo todo el **String** a estudiar tal y como vimos para el método **exec()**.

Tanto para **test()** como para **exec()** tener en cuenta que existe la propiedad **lastIndex** que nos indica dónde se encuentra la siguiente coincidencia de la encontrada en esos momentos. Así que, si en algún momento queremos ?volver a empezar? tendremos que configurar esa propiedad a **0**.

1.4.2 Propiedades de los objetos RegExp

Los objetos **RegExp** tienen cinco propiedades:

- **Source** - cadena de sólo lectura que contiene el texto de la expresión regular.
- **Global** - valor booleano de sólo lectura que especifica si la expresión regular tiene el indicador **g**.
- **IgnoreCase** - valor booleano de sólo lectura que especifica si la expresión regular tiene el indicador **i**.

- **Multiline** - valor booleano de sólo lectura que especifica si la expresión regular tiene el indicador **m**.
- **lastIndex** - entero de lectura/escritura. En patrones con el indicador **g**, esta propiedad guarda la posición de la cadena en la que se inicia la siguiente búsqueda.

1.5 Patrones complejos

A partir de los patrones sencillos ya conocidos, se pueden crear expresiones regulares mucho más complejas. Estos patrones están formados por grupos, referencias inversas, búsquedas directas y otras funciones complejas.

1.5.1 Agrupar

Las agrupaciones se usan mediante la inclusión entre paréntesis de un conjunto de caracteres. Por ejemplo, si deseamos buscar la cadena ?toctoc?, podemos hacerlo del siguiente modo:

```
var expReg = /toctoc/g;
```

Para modificar este patrón empleando agrupaciones lo haremos del siguiente modo:

```
var expReg = /(toc){2}/g;
```

Los paréntesis hacen una agrupación y, a continuación, entre llaves, indicamos el número de veces que queremos se repita (en este caso 2).

Sin embargo, no estamos limitados al uso de llaves con grupos, existen otros cuantificadores:

```
var expReg1 = /(toc)?/g; // 0 o 1 instancias de ?toc?
var expReg2 = /(toc)*/g; // 0 o más instancias de ?toc?
var expReg3 = /(toc)+/g; // 1 o más instancias de ?toc?
```

Anexo cuantificadores donde vemos la diferencia entre utilizar o no el símbolo ?**?**? después de ?***? y ?**+**? :

```
var texto = "tocxtocxxtocxxxxtoc";
var expreReg1 = new RegExp(".*toc", "g");
var expreReg2 = new RegExp(".*?toc", "g");
var expreReg3 = new RegExp(".+toc", "g");
var expreReg4 = new RegExp(".+?toc", "g");

console.log("Texto : " + texto);
console.log("Patrón : '.*toc'->Salida : " + texto.match(expreReg1));
console.log("Patrón : '.*?toc'->Salida : " + texto.match(expreReg2));
console.log("Patrón : '.+toc'->Salida : " + texto.match(expreReg3));
console.log("Patrón : '.+?toc'->Salida : " + texto.match(expreReg4));

/* Salida:
Texto : tocxtocxxtocxxxxtoc
Patrón : '.*toc'->Salida : tocxtocxxtocxxxxtoc
Patrón : '.*?toc'->Salida : toc,xtoc,xxtoc,xxxxtoc
Patrón : '.+toc'->Salida : tocxtocxxtocxxxxtoc
Patrón : '.+?toc'->Salida : tocxtoc,xxtoc,xxxxtoc
*/
```

Incluso podemos crear grupos más complicados:

```
//1 o más instancias de ?ba?, ?da?, ?bad?, ?dad?
var expReg = /([bd]ad?)+/g;
```

Podemos incluir grupos dentro de otros grupos:

```
//comparar ?mamá? o ?mamá y papá?
var expReg = /(mamá( y papá)?)/g;
```

Con expresiones regulares podremos crear un método **trim()** para cadenas (implementado ya en JavaScript como **texto.trim()**).

```
var expEspExtra = /^\s+(.*?)\s+$/;
```

Al utilizar esta expresión junto al método **replace()** del objeto **String** y empleando la técnica de referencias inversas, que veremos en el siguiente punto, podemos definir nuestro propio método **trim()**:

```
String.prototype.miTrim = function() {
    let expEspExtra = /^\s+(.*?)\s+$/;
    return this.replace(expEspExtra, "$1");
};
// Como veremos a continuación ?$1?
//es la primera referencia inversa
let texto ="    cosa   ";
console.log(texto);
console.log(texto.miTrim());
```

Esta técnica, de modificar los objetos predefinidos en JavaScript, se denomina Monkey patching y no es muy recomendable realizarla.

1.5.2 Referencias inversas

Cuando se evalúa una expresión regular con los métodos **test()**, **match()** o **search()** (también con el método **replace()** de los objetos **String()**), se obtiene un grupo de coincidencias que se almacena en una ubicación especial para su posterior utilización. Este grupo de coincidencias tienen como nombre ?referencias inversas?.

Podemos realizar el siguiente ejemplo para entender como funcionan las referencias inversas:

```
//Definimos la expresión regular
const expEspExtra = /^\s+(\w+)\s+$/;
//Texto con el que vamos a trabajar
const texto = "   cosa   ";
//Aplicamos patrón al texto para buscar coincidencias
let coincidencias = texto.match(expEspExtra);
//Analizamos el objeto de salida
console.log("Contenido del objeto: ");
console.log(coincidencias);
console.log("Tipo de objeto: " + typeof(coincidencias));
console.log("Número de elementos: " + coincidencias.length);
console.log("Iteramos en cada uno de los elementos:");
console.log("\tClave --> Valor");
for(let i in coincidencias) {
    console.log("\t" + i + " --> " + coincidencias[i]);
}
/*Salida:
Contenido del objeto:
Array [ "   cosa   ", "cosa" ]
Tipo de objeto: object
Número de elementos: 2
Iteramos en cada uno de los elementos:
        Clave --> Valor
        0 -->    cosa
        1 --> cosa
        index --> 0
        input -->    cosa
        groups --> undefined
*/
```

Como se comentó antes: al agregar el indicador *g* se realiza una búsqueda global y, así, cada elemento de la matriz será una coincidencia total. Si no se añade ese indicador *g*, el primer elemento de la matriz será la coincidencia completa y, los demás elementos de la matriz serán resultados de las subexpresiones entre paréntesis de la completa.

Vemos que podemos acceder a las referencias inversas del siguiente modo:

- Primera referencia inversa : coincidencias[1]
- Segunda referencia inversa : coincidencias[2]
- ...

Otro modo de acceder a ellas será utilizando el objeto **RegExp**:

- Primera referencia inversa : RegExp.$1

- Segunda referencia inversa : RegExp.$2
- ...

Por último, las referencias inversas pueden utilizarse con el método **replace()** de **String** con las secuencias de caracteres especiales **$1**, **$2**, etc. El mejor ejemplo para ilustrarlo cosiste en invertir el orden de dos elementos de una cadena. Imagina que deseas cambiar la cadena "1234 5678" por "5678 1234". Podríamos hacerlo del siguiente modo:

```
const textoBase = `Esta es una secuencia
 de números 1234 5678 muy interesante.`
const patron = /(\d{4})\s+(\d{4})/;
let textoNuevo = textoBase.replace(patron, "$2 $1");
//Vemos cómo se realiza el cambio:
console.log("Texto base: ");
console.log(textoBase);
console.log("Coincidencias con el patrón :" + patron);
console.log(textoBase.match(patron));
console.log("Con el método replace() cambiamos de orden")
console.log(`Así $1: ${RegExp.$1} y $2: ${RegExp.$2}`);
console.log("El texto ahora queda así: ");
console.log(textoNuevo);
/*Salida:
Texto base:
Esta es una secuencia
 de números 1234 5678 muy interesante.
Coincidencias con el patrón :/(\d{4})\s+(\d{4})/
Array(3) [ "1234 5678", "1234", "5678" ]
Con el método replace() cambiamos de orden
Así $1: 1234 y $2: 5678
El texto ahora queda así:
Esta es una secuencia
 de números 5678 1234 muy interesante.
*/
```

1.5.3 Alternar

Imagina que quieres buscar en un texto la existencia de las palabras "roja" y "negra"con una única expresión. Estas palabras no tienen caracteres en común, por lo que deberías aplicar el operador **OR** en nuestra expresión regular. El modo de hacerlo es utilizando la barra (|) tal y como podemos ver en el siguiente ejemplo. En él se utiliza el texto de la Wikipedia sobre el origen de la bandera de Albania y buscamos en él la existencia o no de las palabras "negra/o" o "roja/o":

```
const texto = `La bandera de Albania es una bandera roja con un águila bicéfala negra en el centro. Tiene su origen en un sello simi
const patronRojoNegro = /(roj[oa]|negr[oa])/;
console.log(texto);
console.log(patronRojoNegro.test(texto)); //Devuelve true
```

1.5.4 Grupos sin captura

Los grupos que crean referencias inversas se denominan también "grupos de captura". En expresiones regulares muy extensas, el almacenamiento de referencias inversas ralentiza el proceso de búsqueda. JavaScript permite realizar "grupos sin captura" para no sobrecargar el sistema. Para realizarlo sólo tendremos que acompañar el paréntesis de apertura de un interrogante y dos puntos "**(?: ...)**". Como en el siguiente ejemplo:

```
const textoBase = `Ésta es una secuencia
 de números 1234 5678 muy interesante.`
const patronCC = /(\d{4})\s+(\d{4})/;
const patronSC = /(?:\d{4})\s+(?:\d{4})/;
//Comprobamos la existencia o no de Referencias inversas
console.log("Con Captura:");
console.log(textoBase.match(patronCC));
console.log("Sin Captura:");
console.log(textoBase.match(patronSC));
/*Salida:
Con Captura:
Array(3) [ "1234 5678", "1234", "5678" ]
Sin Captura:
Array [ "1234 5678" ]
*/
```

1.5.5 Búsquedas directas

En ocasiones es necesario capturar un determinado grupo de caracteres sólo si aparecen antes que otro grupo de caracteres. El uso de búsquedas directas facilita esta operación.

Existen búsquedas directas negativas y positivas. Para crear una **búsqueda directa positiva** es necesario incluir un patrón entre **(?=** y **)**. Ojo con estos paréntesis, pues no representan un grupo. Para crear una **búsqueda directa negativa** es necesario incluir un patrón entre **(?!** y **)**. Podemos ver un ejemplo para que quede más claro el concepto:

```
//Texto de trabajo
let texto = `En Compostela se realiza compostaje.`;
//Expresiones Regulares
let rePositiva = /(compos(?=tela))/gmi;
let reNegativa = /(compos(?!taje))/gmi;
//Búsqueda de coincidencias
console.log(texto.search(rePositiva));
//Salida: 3  (Posición inicio Compostela)
console.log(texto.search(reNegativa));
//Salida: 3  (Posición inicio Compostela)

//Cambiamos las Expresiones Regulares
rePositiva = /(compos(?=taje))/gmi;
reNegativa = /(compos(?!tela))/gmi;
//Búsqueda de coincidencias
console.log(texto.search(rePositiva));
//Salida: 25  (Posición inicio compostaje)
console.log(texto.search(reNegativa));
//Salida: 25  (Posición inicio compostaje)
```

1.6 Patrones comunes

En la web las expresiones regulares se suelen utilizar para validar entradas del usuario antes de enviar los datos al servidor. Alguno de los patrones más utilizados en Web son los siguientes:

1.6.1 Fechas

Formato dd/mm/aaaa genérico y con "grupos con captura" para acceder al día, al mes y al año independientemente:

```
let texto = "Naciste el día 14/12/2020, día de Nochebuena";
let reFecha = /(\d{1,2})\/(\d{1,2})\/(\d{4})/g;
let coincidencias = texto.match(reFecha);
console.log(coincidencias);
//Salida: "14/12/2020"
//Prueba cuál sería la salida si no pones el parámetro 'g'
```

El problema es que podemos tener días y meses fuera de rango : 55/42/2004

Podemos ir diseñando una mejor RegExp para las fechas:

- Patrón lógico para días:

```
let reDia = /[0-3]?[0-9]/;
```

- Así y todo, existen días erróneos como del 32 al 39. Para solucionarlo:

```
let reDia = /0[1-9]|[12][0-9]|3[01]/;
```

- El patrón de meses será muy similar:

```
let reMes = /0[1-9]|1[0-2]/;
```

- Para el año, podemos centrarnos entre el 1900 y 2099:

```
let reAno = /19|20\d{2}/;
```

Así, podemos combinar todos los patrones y, además, utilizando "grupos sin captura" para cada parte. Aunque también podríamos dejarlos con captura si fuese necesario para algo.

```
let reFecha = /(?:0[1-9]|[12][0-9]|3[01])\/(?:0[1-9]|1[0-2])\/(?:19|20\d{2})/;
```

Por último, es mucho más sencillo utilizar una función para comprobar la validez de una fecha, por lo que utilizaremos esta expresión regular para crear la función **esFechaValida()**:

```
//Esta función devuelve 'true' o 'false' dependiendo si el texto es una fecha.
function esFechaValida(texto) {
    let reFecha = /(?:0[1-9]|[12][0-9]|3[01])\/(?:0[1-9]|1[0-2])\/(?:19|20\d{2})/;
    return reFecha.test(texto); }
// -- //
//Sería interesante realizar una función que, dado un texto,
// devuelva un array con todas las fechas existentes
```

1.6.2 Tarjetas de crédito

Hoy en día, el uso de las tarjetas de crédito está muy extendido para el pago *online*. Así que, será necesario la validación de dichos identificadores en muchos formularios.

Para empezar, podemos utilizar un número de **MasterCard**, que debe tener 16 dígitos. De éstos, los dos primeros números deben estar comprendidos entre 51 y 55. El patrón será el siguiente:

```
let reMasterCard = /^5[1-5]\d{14}$/;
```

Este patrón es correcto pero los números de **MasterCard** se pueden introducir con espacios o guiones entre grupos de cuatro dígitos, como en 5555-5555-5555-5555, algo que también debemos tener en cuenta:

```
let reMasterCard = /^5[1-5]\d{2}([\s-]?\d{4}){3}$/;
```

La validación real de números de tarjeta de crédito requiere utilizar el **algoritmo de Luhn**. Para aplicarle el algoritmo a cada uno de los números, es necesario extraerlos de la entrada del usuario, lo que implica utilizar grupos de captura en la expresión regular:

```
let reMasterCard = /^(5[1-5]\d{2})[\s\-]?(\d{4})[\s\-]?(\d{4})[\s\-]?(\d{4})$/;
```

Ya podemos empezar a crear una función para validar un número de **MasterCard**. El primer paso consiste en probar una determinada cadena en función del patrón. Si la cadena coincide, añadimos los grupos de cuatro dígitos a la misma (por ejemplo : ?5432-1234-5678-9012? se convertiría en ?5432123456789012?).

Esa secuencia de números es la que le tenemos que pasar a la función que implementa el **algoritmo de Luhn** y que podemos bajar de la web.

El código de las dos funciones quedaría del siguiente modo:

```
//Función implementación algoritmo de Luhn
function valid_credit_card(value) {
    // Accept only digits, dashes or spaces
    if (/[^0-9-\s]+/.test(value)) return false;
    // The Luhn Algorithm. It's so pretty.
    let nCheck = 0, bEven = false;
    value = value.replace(/\D/g, "");

    for (var n = value.length - 1; n >= 0; n--) {
        var cDigit = value.charAt(n), nDigit = parseInt(cDigit, 10);
        if (bEven && (nDigit *= 2) > 9) nDigit -= 9;
        nCheck += nDigit;
        bEven = !bEven;
    }
    return (nCheck % 10) == 0;
}

//Función programada por nosotros que valida
//la Tarjeta de Crédito llamando a la función de Luhn
function esValidaMasterCard(sText) {
    let reMasterCard = /^(5[1-5]\d{2})[\s\-]?(\d{4})[\s\-]?(\d{4})[\s\-]?(\d{4})$/;
    if (reMasterCard.test(sText)) {
        let sNumTar = RegExp.$1 + RegExp.$2 + RegExp.$3 + RegExp.$4;
        //Comprobamos alguna salida
```

```
            console.log("Array salida método match:");
            console.log(sText.match(reMasterCard));
            console.log("Texto que se le pasa a Luhn: " + sNumTar);
            //Realizamos aquí llamada a la función
            //que implementa el algoritmo de Luhn
            return valid_credit_card(sNumTar); }
        else { return false; }
    }

    //Hacemos una comprobación
    let numTarjeta = "5432-1234-5678-9012";
    let valida = esValidaMasterCard(numTarjeta);
    console.log("La tarjeta con número " + numTarjeta + " es válida? " + valida);
```

En el caso de otras tarjetas de crédito, como vemos, la comprobación de la validez de su número va a ser similar, pues casi todas se basan en que cumplan el algoritmo de Luhn.

1.6.3 Direcciones de correo electrónico

La creación de un patrón para comparar todas las direcciones válidas de correo eléctrónico es una labor compleja. Hay que saber que, la especificación que define la validez de una dirección de correo es RFC 2822, que considera válidos los siguientes patrones:

- andrea@dominio.com
- andrea.torres@dominio.com
- Andrea Torres <andrea.torres@dominio.com>
- "andrea.torres"@dominio.com
- andrea@[10.1.3.1]

Está claro que, de todas éstas, los usuarios suelen utilizar las dos primeras en sitios y aplicaciones Web, por lo que nos centraremos en la validación de estos patrones. Así, tendremos en cuenta las siguientes normas:

- Una dirección de correo electrónico está formada por una serie de caracteres (números, letras, guiones, puntos o cualquier otro menos espacios), seguidos por un símbolo **@**, seguido por más caracteres.
- Por delante del símbolo @ debe haber, al menos, un carácter.
- Por detrás del símbolo @ debe haber, al menos, tres caracteres, el segundo de los cuales ha de ser un punto (**a@a.b** es una dirección válida, mientras que **a@a** no lo es).
- El texto que aparece por delante y por detrás del símbolo @ nunca puede comenzar ni terminar en punto, y no puede tener dos puntos seguidos.

Así pues, la expresión regular sería la siguiente:

```
let reEmail = /^(?:\w+\.?)*\w+@(?:\w+\.)+\w+$/;
```

Y así, la función de validación quedaría del siguiente modo:

```
function esValidoEmail(sText) {
    let reEmail = /^(?:\w+\.?)*\w+@(?:\w+\.)+\w+$/;
    return reEmail.test(sText);
}
```

Volver