UT03.5: Expresiones Regulares

Contenido

1. Introducción

Las expresiones regulares son patrones que se utilizan para reconocer combinaciones de caracteres en un *string*, o para chequear si un texto tiene un formato determinado. Un ejemplo podría ser el formato de una fecha, o un correo electrónico. También podemos extraer información de la cadena.

2. Crear expresiones regulares

Las expresiones regulares en JavaScript tienen su propio constructor new `RegExp()`, aunque lo habitual es utilizarlo mediante las barra. En el siguiente ejemplo vemos como crear expresiones regulares mediante barras.

La expresión `/Mancha/` es una expresión regular que se define como una secuencia de caracteres determinada, es decir, esta expresión nos permite localizar si existen esta secuencia de caracteres en un *string*.

El método `test()` de una expresión regular nos dice si un string contiene el patrón o no. En la función vemos un ejemplo.

En cuanto al método `exec()` devuelve un array con la primera coincidencia del patrón. El array contiene dos **propiedades**:

- *index*: con la posición donde se ha localizado el patrón.
- *input*: con la cadena de entrada.

```
function createRegularExpression() {
  const str = 'En un lugar de la Mancha.';
  const reg1 = /Mancha/; // Declaración de una expresión regular.
  const reg2 = /España/;

  // Evaluación de expresión regular.
  $$result.log(`"Mancha" está en "En un lugar de la Mancha.":
${reg1.test(str)}`); // true
  $$result.log(`"España" está en "En un lugar de la Mancha.":
${reg2.test(str)}`); // false

  $$result.log(`Patrón reconocido: ${reg1.exec(str)[0]}`); // "Mancha"
  $$result.log(`Posición en la cadena: ${reg1.exec(str).index}`); // 18
  $$result.log(`Cadena de entrada: ${reg1.exec(str).input}`); // "En un
lugar de la Mancha."
  $$result.log(`Patrón reconocido: ${/de/.exec(str)[0]}`); // "de"
  $$result.log(`Posición en la cadena: ${/de/.exec(str).index}`); // 12
  $$result.log(`Patrón reconocido: ${reg2.exec(str)}`); // null
}
```

3. Caracteres opcionales

Tenemos dos métodos de definir opcionalidad a la hora de definir una expresión regular. Los corchetes nos permiten elegir uno de los caracteres que puede contener un patrón. En este ejemplo `/[Mm]ancha/` el patrón representa una secuencia que puede comenzar con “M” o “m”.

Con los corchetes podemos utilizar rangos de caracteres en base su codificación ASCII. Esta expresión `/[0-9]/` representa cualquier carácter numérico. Podemos negar también la opcionalidad. En este ejemplo `/[^0-9]/` representa cualquier carácter que no sea un número.

El segundo método para la opcionalidad es el uso del carácter "|" y se aplica a secuencia de caracteres. En esta expresión /azul|verde/ el patrón podría ser o bien "azul" o bien "verde".

```
function optionalCharacters() {
  $$result.logBold('Caracteres opcionales');
  const reg = /[Mm]ancha/; // Los corchetes nos dan la posibilidad de
elegir uno de los caracteres que contienen.
  $$result.log(reg.exec('En un lugar de la Mancha')[0]); // "Mancha"
  $$result.log(reg.exec('En un lugar de la mancha')[0]); // "mancha"
  // Podemos especificar rangos de caracteres en función de su código
Unicode.
  $$result.logBold('Rangos de caracteres');
  $$result.log(/[0-9]/.exec('En un lugar 7de la Mancha')[0]); // "7"
  // El acento circunflejo niega los caracteres que estamos buscando
  $$result.log(/[^0-9]/.exec('3.1416')[0]); // "."
  // | índica también opcionalidad
  $$result.logBold('Secuencia de caracteres opcionales');
  $$result.log(/azul|verde/.exec('rojo, azul, verde')[0]); // "azul"
}
```

Tenemos rangos de caracteres predefinidos. En la siguiente función podemos ver los más importantes.

```
function commonSets() {
  // \w equivale a [a-zA-Z0-9]
  // No inclye tildes ni ñ
  $$result.logBold('\\w equivale a [a-zA-Z0-9]');
  $$result.log(/\w/.exec('----a----')[0]); // "a"
  // \W equivale a [^a-zA-Z0-9]
  $$result.logBold('\\W equivale a [^a-zA-Z0-9]');
  $$result.log(/\W/.exec('1111-aaaa')[0]); // "-"
  // \d equivale a [0-9]
  $$result.logBold('\\d equivale a [0-9]');
  $$result.log(/\d/.exec('aaaa1aaaa')[0]); // "1"
  // \D equivale a [^0-9]
  $$result.logBold('\\D equivale a [^0-9]');
  $$result.log(/\D/.exec('1111+1111')[0]); // "+"
  // \s para espacios en blanco (espacios, tabuladores, etc).
  $$result.logBold('\\s para espacios en blanco (espacios, tabuladores,
etc).');
  $$result.log(/\s/.exec('1111 1111')[0].length); // "1"
  // \S para NO espacios en blanco.
  $$result.logBold('\\S para NO espacios en blanco.');
  $$result.log(/\S/.exec('    a    ')[0]); // "a"
}
```

4. Cuantificadores

Podemos definir el número de repeticiones que queremos que recoja el patrón. Siendo *n* y *m* números enteros:

- *{n}*: Define una repetición de n veces.
- *{n,m}*: Define un número de repeticiones entre *n* y *m*.
- *{n,}*: Define un número de repeticiones de al menos *n* caracteres.

En la siguiente función vemos un ejemplo de este tipo de cuantificadores.

```
function repeatCharacters() {
  const str = 'En un lugar 12345de la Mancha';
  // Esta expresión busca 4 digitos seguidos.
  $$result.logBold('Expresión con 4 caracteres seguidos');
  $$result.log(/\d\d\d\d/.exec(str)[0]); // "1234"
  // \d{4} Las llaves indican que busque cuatro digitos seguidos
  $$result.logBold('Igual que la anterior pero utilizando { }');
  $$result.log(/\d{4}/.exec(str)[0]); // "1234"
  // \d{1,3} Busca entre 1 y 3 digitos.
  $$result.logBold('Rangos');
  $$result.log(/\d{1,3}/.exec(str)[0]); // "123"
  // \d{2,} Busca dos dígitos o más en el string
  $$result.logBold('Rangos infinitos');
  $$result.log(/\d{2,}/.exec(str)[0]); // "12345"
}
```

La otra clase de cuantificadores que podemos utilizar son los comodines.

- *: Indica una repetición de 0 o más veces.
- +: Indica una repetición de 1 o más veces.
- ?: Indica una repetición de 0 o 1 vez para expresar opcionalidad.

Veamos algunos ejemplos de cómo utilizar este tipo de cuantificadores. En esta función estamos declarando tres expresiones regulares similares, básicamente la palabra “texto” seguida de una combinación de números y un cuantificador. El cuantificador es aplicado a carácter que tiene delate, en este caso el dígito.

El método `match()` de un string determina si en el string se encuentra el patrón definido con la expresión, y cuál es exactamente el patrón que hemos encontrado. En el ejemplo vemos como la primera expresión con * se empareja tanto con el texto con dígitos como sin dígitos. La segunda expresión con + no se empareja con el texto sin dígitos porque al menos necesitamos 1. Por último, la tercera expresión con ? se empareja con un dígito o con ningún digito, expresando la opcionalidad.

```
function useWildcardsV1() {
  const reg1 = /texto\d*/;
  const reg2 = /texto\d+/;
  const reg3 = /texto\d?/;

  $$result.logBold('Comodín *');
  $$result.log('texto12345'.match(reg1)); // texto12345
  $$result.log('texto'.match(reg1)); // texto
  $$result.logBold('Comodín +');
  $$result.log('texto12345'.match(reg2)); // texto12345
  $$result.log('texto'.match(reg2)); // null
  $$result.logBold('Comodín ?');
  $$result.log('texto12345'.match(reg3)); // texto1
  $$result.log('texto'.match(reg3)); // texto
}
```

Veamos otro ejemplo de este tipo de cuantificadores utilizando las mismas cadenas de texto.

- /ab*a/:
  - *"**aa**aabaaaa"*: * nos permite no incluir la letra *“b”* por lo que la primera ocurrencia que encuentra el patrón sería *“aa”*.
  - *"a a **aba**aaa"*: En esta ocasión, al separar las *“aaa”* con espacios, la primera ocurrencia que encuentra es *“aba”* incluyendo la *“b”*.
  - *"a a **abbbbba**aaa"*: Por último, nos encuentra *“abbbbba”* ya que * permite un número indeterminado de *“b”*.
- /ab+a/:
  - *"aaa**aba**aaa"*: + nos encuentra *“aba”* porque la *“b”* es obligatoria.
  - *"a a **aba**aaa"*: Obtenemos el mismo resultado anterior.
  - *"a a **abbbbba**aaa"*: En esta ocasión obtenemos *“abbbbba”* porque la *“b”* puede ocurrir un número indeterminado de ocasiones.
- /ab?a/:
  - *"**aa**aabaaaa"*: ? hace que *“b”* se opcional por eso obtenemos *“aa”*.
  - *"a a **aba**aaa"*: Obtenemos *“aba”* ya que *“b”* puede aparecer una vez.
  - *"a a abbbbb**aa**aa"*: Por último, nos recoge *“aa”* correspondientes a los caracteres que aparecer posteriores a la *“b”*.

```
function useWildcardsV2() {
  // Podemos utilizar también comodines
  // * equivale a 0 o más veces {0,}
  $$result.logBold('Comodín *');
  const reg1 = /ab*a/;
  $$result.log(reg1.exec('aaaabaaaa')[0]); // "aa". Corresponde a los dos
primeros caracteres.
  $$result.log(reg1.exec('a a abaaaa')[0]); // "aba"
  $$result.log(reg1.exec('a a abbbbbaaaa')[0]); // "abbbbba"

  // + equivale a 0 o más veces {1,}
  $$result.logBold('Comodín +');
  const reg2 = /ab+a/;
  $$result.log(reg2.exec('aaaabaaaa')[0]); // "aba"
  $$result.log(reg2.exec('a a abaaaa')[0]); // "aba"
  $$result.log(reg2.exec('a a abbbbbaaaa')[0]); // "abbbbba"

  // * equivale a 0 o más veces {0,1}
  $$result.logBold('Comodín ?');
  const reg3 = /ab?a/;
  $$result.log(reg3.exec('aaaabaaaa')[0]); // "aa". Corresponde a los dos
primeros caracteres.
  $$result.log(reg3.exec('a a abaaaa')[0]); // "aba"
  $$result.log(reg3.exec('a a abbbbbaaaa')[0]); // "aa". Corresponden a
los caracteres posteriores a "bbbbb".
}
```

Tenemos que notar que la expresión siempre localiza el patrón con la cadena de mayor tamaño que encuentre. En ocasiones necesitamos que el emparejamiento sea con la cadena de menor caracteres, esto lo logramos combinando `+?`. Observa el ejemplo en la función.

Para terminar, tenemos el carácter `.` para representar una ocurrencia de cualquier carácter.

```
function useWildcardsV3() {
  // Para evitar que el comodín se empareje con la cadena más larga se
utiliza ?
  $$result.logBold('Comodín +?');
  $$result.log(/ab+/.exec('a a abbbbbaaaa')[0]); // Devuelve "abbbb"
  $$result.log(/ab+?/.exec('a a abbbbbaaaa')[0]); // Devuelve "ab"

  // . equivale a cualquier carácter menos el de nueva línea.
  $$result.logBold('Comodín .');
  $$result.log(/...../.exec('12345')[0]); // Devuelve "12345"
  $$result.log(/.n/.exec('En un lugar de la Mancha')[0]); // Devuelve
"En"
}
```

5. Carácter de escape

Si queremos procesar caracteres utilizados en una expresión regular debemos utilizar el carácter de escape `\` que le indica al intérprete que el siguiente carácter se debe procesar lo haga como un carácter normal.

```
function escapeCharacters() {
  $$result.clear();
  // Algunos de los caracteres que debemos escapar son:  [ ] { } ( ) * .
^ $
  $$result.logBold('Escapando caracteres');
  const result = /\*/.exec('esto es un *');
  $$result.log(result); // ["*"].
}
```

6. Agrupamientos

Los paréntesis permiten agrupar secuencias para aplicarles un cuantificador posterior. La expresión `/(aba){3}/` se utiliza para repetir la cadena *"aba"* tres veces seguidas.

El paréntesis tiene una doble función, ya que permite extraer información del patrón para ser reutilizada posteriormente. Un ejemplo es el uso del método `exec()` que devuelve un array con el patrón localizado. Por cada conjunto de paréntesis el método devolverá una posición del array con el subpatrón representado en los paréntesis. En la función de ejemplo vemos como al procesar una fecha, el array resultante tiene los valores del día, mes y el año en posiciones del array para no tener que volver a procesar la cadena extrayendo esta información.

Los subpatrones con los paréntesis podemos reutilizarlos dentro de la misma expresión. Estos valores se referencian en la expresión como `\1`, `\2`, `\3`, etc en función de su posición. La expresión `/(["']).*\1/` permite localizar un fragmento entrecomillado ya sea con comilla simple o doble. `.*` representa cualquier conjunto de caracteres, y `\1` hace referencia a si hemos escogido una comilla simple, debemos cerrar con la misma comilla, y si hemos comenzado con doble, lo finalizamos con doble.

Por último, si comenzamos los paréntesis con la combinación `?:` permite ignorar el subpatrón del paréntesis porque no necesitamos su información. Utilizamos el paréntesis solo para agrupar la secuencia.

```
function parenthesisExample() {
  $$result.logBold('Agrupamientos con ( )');
  $$result.log(/(aba){3}/.exec('abaabaabaabaaba')[0]); // "abaabaaba"
  $$result.log(/(aba)+/.exec('abaabaabaabaaba')[0]); // "abaabaabaabaaba"
  $$result.log(/(aba)+?/.exec('abaabaabaabaaba')[0]); // "aba"

  // Con el paréntesis también podemos extraer información del patrón
para reutilizarla.
  // El siguiente ejemplo muestra un expresión para una fecha.
  $$result.logBold('Subpatrones');
  const d = new Date();
  $$result.log(/\d{1,2}\/\d{1,2}\/\d{4}/.exec(d.toLocaleDateString())[0])
; // ["dd/mm/aaaa"].
  // Podemos obtener el valor de cada uno de los paréntesis para
reutilizarlos. Se quedan guardados en el array.
  $$result.log(/(\d{1,2})\/(\d{1,2})\/(\d{4})/.exec(d.toLocaleDateString(
)); // ["dd/mm/aaaa", "dd", "mm", "aaaa", index: 0, input:
"01/11/2017"].

  // Los patrones encontrados en los paréntesis pueden ser reutilizado en
otros puntos de la expresión.
  // El valor encontrado en cada paréntesis puede referenciarse como
"\1", "\2", "\3", etc.
  // El siguiente ejemplo muestra como capturar texto entrecomillado con
comilla simple o doble.
  $$result.logBold('Reutilización de subpatrones');
  $$result.log(/(["']).*\1/.exec("En un lugar de la 'Mancha'")[0]); //
"'Mancha'"
  $$result.log(/(["']).*\1/.exec('En un lugar de la "Mancha"')[0]); //
'"Mancha"'
  $$result.log(/(["']).*\1/.exec('En un lugar de la "Mancha\'')); // null

  // Para ignorar lo encontrado en un paréntesis debemos de usar "?:" al
comienzo del paréntesis.
  $$result.logBold('Ignorar un subpatrón');
  $$result.log(/(?:\d{1,2})\/(?:\d{1,2})\/(?:\d{4})/.exec(d.toLocaleDateS
tring())); // ["dd/mm/aaaa", index: 0, input: "01/11/2017"].
  $$result.log(/(?:aba){3}/.exec('abaabaabaabaaba')); // "abaabaaba"
}
```

7. Posición en la expresión

Podemos definir en que lugar de la cadena debe estar localizado el patrón de nuestra expresión. Por ejemplo, el acento circunflejo indica en la expresión que el patrón debe aparecer al inicio de la cadena de texto `/^En un lugar/`. En el caso de querer que el patrón se localice al finalizar la cadena utilizamos el dólar `/Mancha$/`.

```
function positionExpressionV1() {
  $$result.logBold('Comienzo de la cadena ^');
  const str = 'En un lugar de la Mancha';
  // ^ indica que la expresión debe emparejarse al comienzo del string.
  $$result.log(/^En un lugar/.test(str)); // true
  $$result.log(/^En un lugar/.test(`Otra cadena: ${str}`)); // false

  $$result.logBold('Final de la cadena $');
  // $ indica que la expresión debe emparejarse al final del string.
  $$result.log(/Mancha$/.test(str)); // true
  $$result.log(/Mancha$/.test(`${str}Otra cadena`)); // false
}
```

Podemos indicar los delimitadores de palabra con `\b`, estos incluyen espacios y símbolos de puntuación. Con `\B` indicamos que la palabra no debe finalizar aún.

```
function positionExpressionV2() {
  const str = 'En un lugar de la Mancha';

  $$result.logBold('Delimitador de palabra');
  // Con \b nos sirve de delimitador de una palabra para localizarla en
el string.
  $$result.log(/\bJava\b/.exec('Java, es un lenguaje de
programación.')[0]); // "Java"
  $$result.log(/\bJava\b/.exec('JavaScript es un lenguaje de
programación.')); // null
  // También elimina símbolos de puntuación.
  $$result.log(/\bJava\b/.exec('La lista de lenguajes que debo conocer es
JavaScritp, Java, HTML, CSS.')[0]); // "Java"
  // Localiza la primera palabra en un string
  $$result.log(/\b\w+\b/.exec('La lista de lenguajes que debo conocer es
JavaScritp, Java, HTML, CSS.')[0]); // "La"
  // Localiza la primera palabra que comience por M
  $$result.log(/\bM\w*\b/.exec(str)[0]); // "Mancha"

  $$result.logBold('Palabra no finalizada');
  // \B índica que la palabra no debe finalizar todavía.
  $$result.log(/\bJava\B/.exec('JavaScript es un lenguaje de
programación.')[0]); // "Java"
  $$result.log(/\bJava\B/.exec('Java es un lenguaje de programación.'));
// null
  $$result.log(/\BScript\b/.exec('Un lenguaje de Script es
JavaScript.')[0]); // "Script"
}
```

Por último, la expresión *(?=expresión)* permite posicionarnos en el string del patrón, lo que nos permite recoger un dato antes o después de su posición. En este ejemplo, la expresión `/\d+(?=Gb)/` permite recoger los dígitos antes del patrón *"Gb"*. Otro ejemplo sería `/\b[\wáéíóú]*\s*(?=(Pérez|Fernández))/` que recoge la cadena de texto que

precede a "*Pérez*" o "*Fernández*", es decir, obtenemos el nombre antes de cualquiera de estos apellidos.

Para recoger datos después de un patrón, tenemos que volver a consumirlo para que nos permita recoger el dato. La expresión `/(?=\.)\.\d+/` nos permite recoger la parte decimal a partir de la localización del punto decimal. Al tratarse de un dato posterior, debemos volver a consumir el punto, por lo que es indicado en la expresión.

Podemos recoger un dato precedido a la negación de una expresión. El patrón `/\d{3,4}(?!Gb)/` detecta los dígitos precedidos a texto que no sea "*Gb*".

```
function positionExpressionV3() {
  $$result.clear();

  $$result.logBold('Datos antes del patrón');
  // (?=expresion) permite posicionarnos en el string, para poder obtener
un dato antes o después de lo encontrado.
  $$result.log(/\d+Gb/.exec('La memoria RAM puede ser de 1024Mb, 8Gb,
16Gb, 32Gb.')[0]); // "8Gb"
  $$result.log(/\d+(?=Gb)/.exec('La memoria RAM puede ser de 1024Mb, 8Gb,
16Gb, 32Gb.')[0]); // "8"
  $$result.log(/\b[\wáéíóú]*\s*(?=(Pérez|Fernández))/.exec('Luis García,
Juan Fernández, José Fernández.')[0]); // "Juan"

  $$result.logBold('Datos después del patrón');
  // Al posicionarnos en el string con ?=, la expresión no es consumida y
hay que repetirla para recoger la información después de dicha expresión.
  $$result.log(/(?=\.)\d+/.exec('3.1416')); // Devuelve null
  $$result.log(/(?=\.)\.\d+/.exec('3.1416')[0]); // Devuelve ".1416"

  $$result.logBold('Patrón inverso');
  // (?!expresión) es la operación inversa, buscando un patrón donde no
se cumpla la expresión.
  $$result.log(/\d{3,4}(?!Gb)/.exec('La memoria RAM puede ser de 8Gb,
16Gb, 32Gb o 1024Mb.')[0]); // Devuelve "1024"
}
```

8. Modificadores

Los modificadores permiten cambiar el comportamiento de una expresión. Son aplicados una vez definida. Los modificadores son:

- **i**: ignora mayúsculas y minúsculas.
- g: busca globalmente todas las ocurrencias del patrón emparejadas con la expresión.
- m: considera el *string* con multilíneas.

Estos modificadores pueden ser combinados entre sí.

Como vemos en el ejemplo, la expresión `/mancha/i` permite localizar patrones independientemente del uso de mayúsculas o minúscular.

El modificador global aplicado conjuntamente con el método `exec()` nos da la posibilidad de localizar nuevos patrones que cumplen la expresión en la cadena. Cada ejecución del método se

hace a partir de la última ocurrencia del patrón. Cuando no es capaz de encontrar nuevas ocurrencias devuelve `null`.

Para la expresión `/Mancha$/m` localiza el patrón siempre y cuando sea fin de línea.

```
function useModifiers() {
  // Los modificadores permiten cambiar el comportamiento por defecto de una expresión regular.
  const str = 'En un lugar de la Mancha';

  $$result.logBold('Ignorando mayúsculas de minúsculas');
  // i: indica que se ignoren mayúscuals y minúsculas.
  $$result.log(/mancha/i.exec(str)[0]); // "Mancha"

  $$result.logBold('Modificador global');
  // g: Busca globalmente todos los patrones que se emparejen con la expresión
  const reg = /n/g;
  $$result.log(reg.exec(str)); // ["n", index: 1, input: "En un lugar de la Mancha"]
  $$result.log(reg.exec(str)); // ["n", index: 4, input: "En un lugar de la Mancha"]
  $$result.log(reg.exec(str)); // ["n", index: 20, input: "En un lugar de la Mancha"]
  $$result.log(reg.exec(str)); // null

  $$result.logBold('Multilineas');
  // m: considera el string con multilineas, teniendo en cuenta el caracter \n
  // Localiza el nombre "Mancha" al final de una de las líneas.
  $$result.log(/Mancha$/m.exec('En un lugar de la Mancha\nde cuyo nombre no quiero acordarme.')[0]); // "Mancha"
  $$result.log(/^de/m.exec('En un lugar de la Mancha\nde cuyo nombre no quiero acordarme.')[0]); // "de"
}
```

9. Iterar sobre una expresión regular

Siempre que utilicemos el modificador global podremos iterar sobre los patrones localizados en una cadena, sabiendo que la condición de salida será un `null` devuelto por el método. En el siguiente ejemplo como recoger las posiciones de los caracteres *"m"* o *"n"* independientemente de que sean mayúsculas o minúsculas.

```
function iterateOverRegularExpression() {
  $$result.logBold('Posición del patrón en la cadena');
  const str1 = 'En un lugar de la Mancha.';
  const reg1 = /[mn]/ig; // Declaración de una expresión regular.
  let item;
  while (item = reg1.exec(str1)) { // 1, 4, 18, 20
    $$result.log(item.index);
  }
}
```

10. Uso de expresiones con *string*

El tipo *string* tiene métodos que pueden ser utilizados conjuntamente con expresiones regulares.

- search(): Localiza un patrón en la cadena definido por una expresión.
- `match()`: Genera un array con la primera ocurrencia encontrada o si utilizamos el modificador global con todas las que encuentre.
- `split()`: Devuelve un array son las subcadenas separadas por un delimitador definido a partir de una expresión regular.
- `replace()`: Reemplaza un patrón localizado a partir de una expresión regular, por una cadena de texto.

La siguiente función muestra ejemplos de su aplicación.

```
function useRegularExpressionWithStringV1() {
  const str = 'En un lugar de la Mancha';

  $$result.logBold('Método: search');
  // Método: search
  $$result.log(str.search(/mancha/i)); // Devuelve 18

  $$result.logBold('Método: match');
  // Método: match
  // Genera un array con la primera ocurrencia encontrada o si utilizamos
el modificador global con todas las que encuentre.
  $$result.log('La memoria RAM puede ser de 1024Mb, 8Gb, 16Gb,
32Gb.'.match(/\d+(?=Gb)/)); // ["8", index: 36, input: "La memoria RAM
puede ser de 1024Mb, 8Gb, 16Gb, 32Gb."]
  $$result.log('La memoria RAM puede ser de 1024Mb, 8Gb, 16Gb,
32Gb.'.match(/\d+(?=Gb)/g)); // ["8", "16", "32"]

  $$result.logBold('Método: split');
  // Método: split
  // Genera un array con separadores irregulares.
  const list = 'José Pérez ;Manuel Fernández; Juan Gómez ; María López
;Ana Díaz ';
  $$result.log(list.split(/\s*;\s*/)); // ["José Pérez", "Manuel
Fernández", "Juan Gómez", "María López", "Ana Díaz "]
}
```

Como podemos observar en el siguiente ejemplo, podemos reutilizar los subpatrones definidos en los paréntesis en la cadena de sustitución, utilizando la expresión `$n`, siendo "n" el orden de los paréntesis definidos.

Por último, como elemento de sustitución podemos utilizar una *función callback* que será llamada por cada ocurrencia del patrón definido por una expresión regular con modificador global. Esta función recibe como parámetros:

- match: es la ocurrencia encontrada.
- offset: su posición.
- string: la cadena completa.

```
function useRegularExpressionWithStringV2() {
  $$result.logBold('Método: Replace');
  // Método: replace
  $$result.log('Man5cha'.replace(/\d/, 'X')); // "ManXcha"
  $$result.log('La memoria RAM puede ser de 1024Mb, 8Gb, 16Gb,
32Gb.'.replace(/Gb\b/ig, 'Tb')); // "La memoria RAM puede ser de 1024Mb,
8Tb, 16Tb, 32Tb."
  // Podemos reutilizar lo recogido en los paréntesis de la expresión
utilizando $n, siendo n la posición del paréntesis.
  $$result.log('Man5cha'.replace(/(\d+)/, '-$1-')); // "Man-5-cha"
  // Este ejemplo cambiar de orden el nombre y el apellido.
  $$result.log('Jose Perez'.replace(/(\w+)\s(\w+)/, '$2 $1')); // "Perez
Jose"

  // Podemos utilizar funciones por cada ocurrencia emparejada con la
expresión
  // La función transforma cada ocurrencia a minúsculas.
  function transform(match, offset, string) {
    // Si la posición no es la 0 añade un guión antes de devolver la
ocurrencia en minúsculas.
    // Si es la 0, al transformase en false, solamente devuelve la
ocurrencia en minúsculas.
    // También devuelve la posición
    return `${(offset ? '-' : '') + match.toLowerCase()}(${offset})`;
  }
  $$result.log('En un lugar de la Mancha'.replace(/[A-Z]/g, transform));
// "e(0)n un lugar de la -m(18)ancha"
}
```

11. Ejemplo de uso

Para finalizar este apartado, vemos tres posibles aplicaciones de uso de expresiones regulares. La función `f2c()` permite localizar el patrón de dígitos antes de un carácter *"F"* que representa temperaturas en Fahrenheit para ser remplazadas por su equivalente en Celsius.

```
function f2c(x) {
  function convert(str, p1, offset, s) {
    // p1 contiene el valor recogido en el paréntesis.
    return `${((p1 - 32) * 5 / 9).toFixed(2)}C`;
  }
  const s = String(x);
  const test = /(-?\d+(?:\.\d*)?)F\b/g;
  return s.replace(test, convert);
}
```

La función `styleHyphenFormat()` transforma una propiedad CSS utilizada en un objeto style de un elemento, y definida por un formato *camel case*, a su equivalente en la especificación CSS 3.0 utilizando guiones.

```
function styleHyphenFormat(propertyName) {
  function upperToHyphenLower(match, offset, string) {
    return (offset ? '-' : '') + match.toLowerCase();
  }
  return propertyName.replace(/[A-Z]/g, upperToHyphenLower);
}
```

Por último, `trim()` limpia un texto eliminando del inicio y del final de la cadena los caracteres especificados pasados por parámetro.

```
function trim(str, filter) {
  function ltrim(str, filter) {
    const pattern = new RegExp(`^(${filter})*`, 'g');
    return str.replace(pattern, '');
  }
  function rtrim(str, filter) {
    const pattern = new RegExp(`(${filter})*$`, 'g');
    return str.replace(pattern, '');
  }
  filter || (filter = '\\s|\\ ');
  return ltrim(rtrim(str, filter), filter);
}
```

Invocamos las funciones creadas.

```
$$result.logBold('Transformar temperaturas');
$$result.log(f2c('Viernes: -80.5F, Sábado 90.5F, Domingo 95.5F')); //
"Viernes: -62.22C, Sábado 32.22C, Domingo 35.00C"
$$result.logBold('Propiedades DOM a CSS');
$$result.log(styleHyphenFormat('borderTop')); // "border-top"
$$result.logBold('Trim');
$$result.log(trim('       En un lugar de la Mancha        ')); // "En un
lugar de la Mancha"
$$result.log(trim('-.-.---En un lugar de la Mancha-.-.-.--', '-|\\.'));
// "En un lugar de la Mancha"
```