Regular Expressions In JavaScript

Published on March 13, 2020

Regular expressions also known as RegExp are special formatted characters that are used to match patterns in strings of text. For example they can be used to validate whether a user entered correct data in a form, before it is processed by the server. They are also used to manupulate strings in various ways such as search and replace.

Various programming languages such JavaScript, PHP, Python and Perl use regular expressions to process and manipulate text. While there's a lot in common in how each of these languages utilize regular expressions, their specific implemention may vary.

In this tutotial, we are going to use JavaScript to explore the usage of regular expressions.

Regular Expression Syntax

A regular expression in JavaScript can be build in one of two ways:

1. Regular expression literal
2. Regular expression constructor

Regular expressions in both approaches consists of a `pattern` and optional `flag(s)`. Whichever approach one uses, the result will be a regular expression object.

Regular Expression Literal

Regular expression literal syntax consists of a `pattern` enclosed in forwardslashes `/.../` .

Syntax: **`/pattern/flag`**

Example: **`var regex = /abc/;`** without flags

Example: **`var regex = /abc/i;`** with flag *i* making it case insensitive.

Use the literal syntax if the regular expression will remain constant through out the script.

Regular Expression Constructor

Regular expression constructor uses a constructor function.

Syntax: **`new RegExp('pattern', 'flags');`**

Example: **`var regex = new RegExp('abc');`** without flags.

Example: **`var regex = new RegExp('abc','m');`** with flag *m* enabling it to search a multi-line string.

Use the constructor syntax to create dynamic regular expressions or in situations where the expression will come from the user input, in which case the literal syntax will not work.

Special Characters in Regular Expressions

Regular expressions use characters that have special meaning in the expressions. That's why *regular expressions* are like a programming language of its own. The following characters have special meaning in regular expressions: `. ? * + [ ] ( ) { } ^ | \`.

In case you want to use these characters in a regular expression in their literal sense, you must

escape them with a backslash "\". For example, if you want to search for a question mark in a string then the regular expression could be `/\?/`. In this case the '?' is treated as a question mark and not as a special character.

The table below lists the special characters and their meaning when used in a regular expression. Later on, we shall explore how we can use them in searching and matching strings.

RegExp	Description
`.`	The dot matches any single character except new line.
`*`	Matches the preceding character 0 or more times. Same as {0,}
`+`	Matches the preceding character 1 or more times. Same as {1,}
`?`	Matches the preceding character 0 or 1 time. Same as {0,1}
`^`	Matches the proceding character if it is beginning the string.
`$`	Matches the preceding character if it is ending the string.
`[abc]`	A character set that matches any one of the letters a, b or c
`[^abc]`	Matches any character other than a, b or c
`[a-z]`	Matches any lowercase letter between a and z
`[0-9]`	Matches any digit between 0 and 9
`[a-z0-9]`	Matches any character between a and z or 0 to 9
`{n}`	Matches the preceding character(s)exactly n times.
`{n,}`	Matches the preceding character(s) least n times.
`{,n}`	Matches the preceding character(s) at most n times.
`{n,m}`	Matches the preceding character(s) at least n and at most m times.
`(x)`	Matches x and remembers the match. Also called *capturing parenthesis*
`(?:x)`	Matches x but does not remember the match. Also called *non-capturing parenthesis*
`x(?=y)`	Matches x only if x is followed by y. Also called *positive look ahead.*
`x(?!y)`	Matches x only if x is not followed by y. Also called *negated look ahead.*
`(?<=y)x`	Matches x only if x is preceded by y. Also called *positive look behind.*
`(?<!y)x`	Matches x only if x is not preceded by y. Also called negated look behind.
`(x|y)`	Matches either x or y.
`\`	A backslash preceding a non-special character indicates that the proceding character should be treated as special. For example `\t` matches tab space instead of `t`.
`\b`	Matches word boundary. Finds a match at the beginning or ending of a word.

\s	Matches a whitespace character including space, tab, new line, carriage return. Equivalent to [\t\n\r]
\w	Matches a word character, that is any letter, digit or underscore. Equivalent to [a-zA-Z0-9_]
\d	Matches any digit character. Equivalent to [0-9]
\S	Matches a non-whitespace character.
\W	Matches any non-word character. Same as [^a-zA-Z0-9_]
\D	Matches any non-digit character. Same as [^0-9]

As said before, incase you want to use any of these special characters literally, you must escape the character with a backslash. If for instance, you want to search for "a" followed by "+" then followed by "b", you would type the regular expression as `/a\+b/` . Then "+" will not be treated as a special character.

However, as we shall see later, when these special characters are used in a character set "[ ]" they are treated literally, and need not be escaped. For example, if you wanted to match either "a+b" or "a-b", you could type the regular expression as `/a[+-]b/`. No need to escape them.

We are going to use the JavaScript Engine in the bowser Console panel, located in Web Developer tools to practise with regular expressions. To access Web Developer tools, use the specific browser's menu or short-cut keys. FireFox and Microsoft Edge use `Shift+F12`; Chrome and Opera browsers use `Ctrl+Shift+I` as short-cut to Developer tools. If the Develop menu on Safari is not displayed, use ***Preferences -> Advanced*** tab and check the ***Show Develop menu*** box.

Flags

Sometimes regular expressions include flags. Flags change the way the search or match will be performed. Whether to use flags or not, depends on what you want to achieve. The flags listed below can be used in JavaScript.

Flag	Description
i	Makes the search case-insensitive i.e. treats a and A equally.
g	Global serch; looks for all matches in a string.
m	Multi-line mode; looks for a match even in new lines.
u	Enables full unicode support (support surrogate pairs).
y	Sticky mode; looks for a match at the specified position.

Methods to Work with Regular Expressions

In JavaScript regular expressions are implemented using methods. Methods can be called from a built-in **`RegExp class`** such as `test` and `exec` or from **`string methods:`** such as `match`

, `search` , `replace` and `split`.

In most of the examples in this tutotial we will use the shorter literal syntax `/.../` to practise with the above methods. But first, let's look at these methods seperately with simple regular expressions, before we come to the special characters in detail.

regexp.test(str)

The `regexp.test(str)` method searches for a match and returns `true` if the match is found or `false` if there is no match.

Run the following code in the console of your browser and see the result.

```
1. var str = "In the animal kingdom lion is king";
2. var regex = /king/;
3. regex.test(str); // true
```

The *regex.test* found a match ("king") in the string and returned *true*.

regex.exec(str)

The `regex.exec(str)` searches for all matches and returns an array of all matched groups.

```
1. var str = "In the animal kingdom lion is king";
2. regex = /king/;
3. regex.exec(str);
4. // returns ["king" index: 14 input: "In the animal kingdom lion is king" length: 1
   ]
5. //"king" -> is the matched pattern
6. //index: -> is the position the regular expression starts
7. //input: -> is the actual string searched.
8. //length: number of matches (1)
```

From the above result it shows that it is the first part of (king)dom that was matched.

str.search(regex)

This method always returns the postion of the first match and -1 if a match was not found.

```
1. var str = "In the animal kingdom lion is king";
2. regex = /king/;
3. str.search(regex); //returns 14 (first position of a match)
```

We can't use `search` method to find the next match. We need to use other methods to do that.

Before we move on, let's illustrate how to use the *RegExp* constructor syntax to create a dynamic regular expression.

```
1. var search = prompt("What do you want to find?","king");
2. regex = new RegExp(search);
3. alert("In the animal kingdom lion is king".search(regex));
```

You specify what you want to search at the prompt such as "king", "lion", "dom" and the alert will return the position of the first match. Here the user is the one to determine what is to be matched.

If you just want to know whether a pattern exists in a string, then use the `regex.test(str)` or `str.search(regex)` method.

str.match(regex)

The `str.match()` method behavior varies depending on the presence or absence of the "g" flag. `str.match(regex)` without "g" returns an array with the match and other properties similar to `regex.exec(str)`.

```
1. var str = "In the animal kingdom lion is king";
2. regex = /king/;
3. str.match(regex);
4. //returns [0: "king" index: 14 input: "In the animal kingdom lion is king" length: 1]
```

When the "g" flag is used in the regular expression, `str.match()` method just returns all the matches with no additional properties.

```
1. var str = "In the animal kingdom lion is king";
2. regex = /king/g;
3. str.match(regex); //returns [0: "king" 1: "king" length: 2]
4. //0: -> the first match, "king"
5. //1: -> the second match, "king"
6. //length: number of matches (2)
```

If there are no matches the `str.match()` method returns a `null` (not the same as empty).

Both `str.match(regex)` and `regex.exec(str)` return an array of the match with other properties if the match is successful, otherwise they return *null*. You can use either of the methods to get more information about a match.

str.split(regex|substr, limit)

This method splits a string using *regexp* or *substr* as delimiter. Suppose we want to split the date

9/24/2019 using forwardslashes as delimeter, run the following code in your browser console.

```
1. alert("9/24/2019".split("/")); // returns [9, 24, 2019]
```

In the above code we have specified the delimeter in a substring. If we used a regular expression to specify the delimeter, we would have to escape the forwardslash delimeter like the code below.

```
1. alert("9/24/2019".split(/\//)); // returns [9, 24, 2019]
```

So, both give the same result.

str.replace(str|regex, str|func)

This method performs search and replace. The first argument is what is to be searched and the second argument is the replacement. Let's replace the dashes with a colon in the date format in the code below.

```
1. alert("22-09-40".replace(/-/g, ":")); // returns 22:09:40
```

If you don't use the "g" flag in the regular expression above, the search will replace the first dash and stop there, resulting in 22:09-40.

We can also use the following special characters to specify positions of insertion.

Symbol	Inserts
`$$`	$
`$&`	the whole word
`$``	apart of the string before the match.
`$'`	apart of the string after the match.
`$n`	if n is a 1-2 digit number, then it means the contents of n-th parentheses counting from left to right

We are going to use `$&` to replace all entries of *Lucy* with *Ms Lucy* in the code below.

```
1. var str = "Lucy Doe, Lucy Soi, Lucy Roe";
2. alert(str.replace(/Lucy/g, "Ms $&"));
3. //returns Ms Lucy Doe, Ms Lucy Soi, Ms Lucy Roe
```

The expression below will interchange the two words "King" and "Lion"

```
1. var str = "King Lion";
2. alert(str.replace(/(King) (Lion)/, "$2,$1")); //return Lion,King
```

In the above illustration we have included capturing groups `(King)` and `(Lion)` but we will explain more on that later.

Well, now let's turn to testing various regular expressions, highlighting the usage of various special

characters.

Character Set [abc]

Using a character set `[abc]` in a regular expression, tells it to match any of the letters "a", "b" or "c" in the provided string in a single position. To make it clear, let's suppose we want to match the words "bear", "rear", "wear" and "Tear". Then our regular expression would look like `/[brtw]ear/`. Let's do several matches in the browser console.

```
1. var regexp = /[brtw]ear/;
2. console.log(regexp.test("bear")); // returns true
3. console.log(regexp.test("rear")); // true
4. console.log(regexp.test("wear")); // true
5. console.log(regexp.test("Tear")); // false - regexp is case sensitive
6. console.log(regexp.test("dear")); // false - d is not part of the character set
7. console.log(regexp.test("ear"));  //false - before "ear" there must one of these: b, r, t or w
8. console.log(regexp.test("bearing")); // true - it matched "bear" in the (bear)ing
9. console.log(regexp.test("rebear")); // true - it matched "bear" in the re(bear)
```

From the matches above, you can see that so long as the string you are testing has a word consisting a cluster of four letters `b|ear`, `r|ear`, `t|ear`, `w|ear`, it won't matter whether there are other letters before or after the cluster, there will be a match. If we wanted to match both lower and upper cases then the regular expression could have been `/[brtw]ear/i` , making it case-insensitive.

Negated Character Set [^abc]

Negated set helps in matching anything but not what is in the set. Let's negate the earlier character set.

```
1. var regexp = /[^brtw]ear/;
2. console.log(regexp.test("bear")); // false
3. console.log(regexp.test("fear")); // true
4. console.log(regexp.test("Tear")); // true - regexp is case sensitive (t != T)
5. console.log(regexp.test("ear"));  //false - before "ear" there must a letter other than: b, r, t or w
6. console.log(regexp.test("rebear")); // false - because of b.
```

Take note that the caret `^` has a different meaning when placed outside the set such as `/^[abc]/`. That would mean that the string to match must begin with any of the three letters *a*, *b* or *c*.

Character Set Ranges, [a-z] or [0-9]

Sometimes the characters you want to use in a set follow each other consecutively. In such cases you can use a character range. For example, if we had used the range [a-zA-Z] in matching "bear", "rear", "wear" and "Tear". All of them would have returned true. Let's try it.

```
1. var regexp = /[a-zA-Z]ear/;
2. console.log(regexp.test("bear")); // true
3. console.log(regexp.test("Tear")); // true - the regexp includes uppercase A-Z
4. console.log(regexp.test("dear")); // true
5. console.log(regexp.test("ear"));  // false - before "ear" there must a letter
6. console.log(regexp.test("Gear")); // true
7. console.log(regexp.test("HEAR")); // false - the regexp is case sensitive
   outside the set
```

Sometimes instead of using character set ranges, you could replace them with named sets, also called *meta-characters* like `\d` instead of `[0-9]`, `\w` instead of `[a-zA-Z0-9_]`. You could also negate by `\D` instead of `[^0-9]` or `\W` instead of `[^a-zA-Z0-9_]` .

Quantifiers

The special characters `+ * ? {n} {n,} {,n} {n,m}` are also called quantifiers. They dictate how many times the preceding character or group of characters should be matched. Let's look at some examples.

`+` :- Matches the preceding character once or more times. It is equivalent to `{1,}`. Let's use the `\d+` to match one or more digits.

```
1. var regexp = /\d+/;
2. console.log(regexp.test("777")); // true
3. console.log(regexp.test("2")); // true
4. console.log(regexp.test("k")); // false - expects a digit
5. console.log(regexp.test("")); //false - expects at least a single digit
```

We would have got the same result if we used `{1,}`. Let's run the matches again.

```
1. var regexp = /\d{1,}/;
2. console.log(regexp.test("777")); //true
3. console.log(regexp.test("2")); // true
4. console.log(regexp.test("k")); // false
5. console.log(regexp.test("")); //false
```

`*` :- Matches the preceding character 0 or more times. It is equivalent to `{0,}`. If we use

`/fo*d/`, the "*" makes the word to be matched whether the "o" is there or not. It also matches for multiple "o".

```
1. var regexp = /fo*d/;
2. console.log(regexp.test("fd")); //true - missig "o" is expected
3. console.log(regexp.test("fod")); // true
4. console.log(regexp.test("fooood")); // true - 0 or more "o" is expected
```

We could have used `{0,}` and got the same result.

```
1. var regexp = /fo{0,}d/;
2. console.log(regexp.test("fd")); //true
3. console.log(regexp.test("fod")); // true
4. console.log(regexp.test("fooood")); // true
```

`?` :- Matches the preceding character 0 or 1 time. It is equivalent to `{0,1}`

```
1. var regexp = /foo?d/;
2. console.log(regexp.test("fd")); //false - there must be "fo" to start with
3. console.log(regexp.test("fod")); // true - the second "o" is optional
4. console.log(regexp.test("food")); // true - double "o" is allowed
5. console.log(regexp.test("foood")); // false -the second "o" cannot be repeated
```

Try with `{0,1}`

```
1. var regexp = /foo{0,1}d/;
2. console.log(regexp.test("fd")); //false
3. console.log(regexp.test("fod")); // true
4. console.log(regexp.test("food")); // true
5. console.log(regexp.test("foood")); // false
```

Please, note that these special characters we have just explained when used within character sets like `[+*?.]`, they cease to be special and assume the literal meaning. In this case `[+*?.]` would match any of these symbols `+, *, ? or .` in a single position.

Position Anchors: String Boundary Characters

There are two special characters `^` and `$` that help in matching the beginning and the end of a string, respectively. They are common in regular expressions.

`^` :- When used it means, for a match to occur, the string must begin with the character(s) placed after "^". Let's use `/^food/` expressioin to test some strings.

```
1. var regexp = /^food/;
2. console.log(regexp.test("food for the hungry")); //true
```

```
3. console.log(regexp.test("the hungry need food")); // false -food is not starting the string
4. console.log(regexp.test("foolish act")); // false -food is expected to begin the string
5. console.log(regexp.test("foodish")); // true -it matched food in (food)ish
```

In the above tests a match occured only where the string began with "food". That was the case in line 2 and 5.

> When the caret `^` is used inside a character set like, *[^xyz]* it means exclude *xyz* in the match. However, placing the caret outside the character set like, `^[xyz]` it matches a string the starts with any these character *x*, *y*, or *z*. So, `/^[xyz]ox/` will match "xox", "yox", "xoxes" but not "wox" or anything that does not start with x, y or z.

`$` :- This symbol is used to mark the end of the string to be matched. For a match to occur the string must end with the character(s) placed before *"$"*. See the example below.

```
1. var regexp = /good$/;
2. console.log(regexp.test("good food")); //false - end of the string is not "good"
3. console.log(regexp.test("the food was good")); // true
4. console.log(regexp.test("goody goody")); // false -the string ends with "goody" instead of "good"
```

Word Boundary Character

A word boundary is denoted by `\b` . A word boundary does not denote a character but rather a boundary between characters. For example `/\bking\b/` will match *"king"* in *"lion is king"* but not in *"animal kingdom"*

```
1. var regexp = /\bking\b/;
2. console.log(regexp.test("king fisher")); // true
3. console.log(regexp.test("kingfish")); // false
4. console.log(regexp.test("in the animal kingdom")); // false
5. console.log(regexp.test("in the animal kingdom \n lion is king")); // true
```

The word boundary `\b` is used to match standalone English words. As you can see in the 5th line it can search beyond the end of a line to new lines of text. The special character `\n` breaks the line of text into a new line.

The Dot (.)

The dot `.` is a specail character that matches any character except *new line* `(\n)`. Suppose we wanted to find exactly a three-character string. We could use the knowledge we have already gained and build the regular expression `/^.{3}$/`. Let's use it as below.

```
1. var regexp = /^.{3}$/;
2. console.log(regexp.test("123")); // true
3. console.log(regexp.test("5547")); // false - expects exactly three characters
4. console.log(regexp.test("pot")); // true
5. console.log(regexp.test("sip cup ")); // false - expects one three-letter word
6. console.log(regexp.test(" cup")); // false - a space is a character
```

The regular expression above can be explained as follows: the `.{3}` searches for a three-character word in a string of text. Then to restrict the match to only a three-character-word string, we enclose the expression in the string boundary characters `^` and `$`. Hence the `/^.{3}$/` regular expression.

If we wanted to search for a number with a decimal point, then we could build a regular expression like `/\d+\.\d+/` .

```
1. var regexp = /\d+\.\d+/;
2. console.log(regexp.test("123")); // false - no decimal point
3. console.log(regexp.test(".55")); // false - there must be at least one digit
   before decimal point
4. console.log(regexp.test("64.00")); // true - because \d stands for 0-9
5. console.log(regexp.test("0.99")); // true
6. console.log(regexp.test("798.407")); // true
```

`\d+` means match 1 or more digits and `\.` stands for the normal decimal point or period, because we have escaped it.

> Remember the dot . is a special character in regular expressions and *d* is not. When they are escaped with the backslash "\" they change meaning depending on where they are being used.

Suppose we want to search for an opening HTML-tag without attributes. An opening HTML-tag starts with `<` and ends with `>`. Then the name of the tag must start with a letter or letters and sometimes it can be followed by a number. Examples are `<p>`, `<h1>` tags. Our regular expression, therefore can be expressed as the one below.

```
1. var regexp = /<[a-z][a-z0-9]*>/i;
2. console.log(regexp.test("<body>")); // true
3. console.log(regexp.test("<H1>")); // true
4. console.log(regexp.test("<!doctype>")); // false - ! is not part of the regexp
```

The above regular expression starts and ends with the angle brackets. The element name must begin with a letter `[a-z]` and can be followed by nothing or more letters or digits `[a-z0-9]*`. Finally we have made it case insensitive `/i` because HTML tags are not restricted to lowercase only.

The regular expression below incorporates opening and closing HTML tags. What makes a closing HTML tag is the forwardslash placed immediately after the left angle bracket. So we have improved the previous *regexp* by adding `\/?` at the beginning the regexp.

```
1. var regexp = /<\/?[a-z][a-z0-9]*>/i;
2. console.log(regexp.test("<body>")); // true
3. console.log(regexp.test("</body>")); // true
4. console.log(regexp.test("</H2>")); // true
```

The forwardslash had to be escaped `\/` and the `/?` makes the forwardslash optional in the regular expression.

Greedy Vs Non-greedy Quantifiers

The quantifiers `* + ? {}` are by themselves referred to as *greedy quantifiers.* They try to match as many characters as possible. Let's illustrate that by trying to match the string *75489Ave* using `/\d+/` regular expression.

```
1. var str = "75489Ave";
2. var regex = /\d+/;
3. str.match(regex); // returns 75489
```

Let's try to match a bigger string with the regexp having a global flag `g`

```
1. var str = "75489Ave abd2874950 123abc345 wxy12xyz";
2. var regexp = /\d+/g;
3. str.match(regexp); // returns "75489", "2874950", "123", "345", "12"
```

You can see from the above illustrations that `/d+/` is greedy. It tries to match as many digits as it can find.

To make these quantifiers *non-greedy* or *lazy* we use `?` immediately after the quantifier. So whenever you see expressions such as `+?, *?, ??, {}?`, they are the same quantifiers made *non-greedy* to match the fewest characters as possible. Now let's run the previous code with the non-greedy quantifier `+?` and see what we will get.

```
1. var str = "75489Ave";
2. var regex = /\d+?/;
3. str.match(regex); // returns 7
```

You can see it matched the first digit and ignored the rest. Let's try a longer string while incorporating the global flag `g`.

```
1. var str = "75489Ave abd2874950 123abc345 wxy12xyz";
2. var regexp = /\d+?/g;
3. str.match(regexp); // "7", "5", "4", "8", "9", "2", "8", "7", "4", "9", ]
```

The global search is even more interesting, it matches each digit it finds in the string as a seperate entity. It makes sense because we have made it to be "non-greedy" to match the fewest digits as possible, and yet it must continue to the end of the string.

Capturing (Parentheses) Groups

When a pattern or part of a pattern of a regular expression is placed within parenthesis `( )` it is called a *capturing group*. The parenthesis matches what is placed within it and remembers the match. It also helps in creating sub expressions within the regular expression. If a quantifier is placed after the parenthesis, the quantifier applies to the whole group.

For example, when we use a pattern like `/(no)+/` it will look for one or more *"no"*. So it can match "no", "nono", "nonono" and so on. It would mean something different if we used `/no+/`. The latter pattern means, look for "n" that may or may not be followed by *"o"*. So it can match "no", "noo", "nooo" and so on.

Conclusion

Well, we have covered some good ground in exploring regular expressions in JavaScript. You now need to start using them in your programming to get familiar with them. Whenever you see a regular expression in somebody's code, try to interpret it before you look it up. The next tutorial I intend to do is, *validating forms using regular expressions*. I wish you happy coding.