**Regular Expressions:**

Regular expressions are patterns that combination of characters. Regular expressions are used to perform pattern-matching.

**Syntax:**

/pattern/modifiers;

Example:

var patt = /w3public/i;

var re = /ab+c/;

Example explained:

* /w3public/i  is a regular expression.
* w3public  is a pattern (to be used in a search).
* i  is a modifier (modifies the search to be case-insensitive).

Multiline search for "is" at the beginning of each line in a string:

var str = "\nIs th\nis h\nis?";

var patt1 = /^is/gmi;

var result = str.match(patt1);

Multiline search for "is" at the end of each line in a string:

var str = "Is\nthis\nhis\n?";

var patt1 = /is$/gm;

var result = str.match(patt1);

Brackets: Brackets are used to find a range of characters:

|  |  |
| --- | --- |
| Expression | Description |
| [abc] | Find those characters between the brackets followed by a string |
| [^abc] | Don’t find those characters between the brackets |
| [0-9] | Find those characters between the brackets (any digit) |
| [^0-9] | Don’t find those characters between the brackets (find non-digit only) |
| (x|y) | Find x or y followed by a string while x & y may be words or characters |

**Difference between [] and () in Regex:**

[] denotes a character class. And, () denotes a capturing group.

[a-z0-9] -- One character that is in the range of a-z OR 0-9

(a-z0-9) -- Explicit capture of a-z0-9. No ranges.

1. a -- Can be captured by [a-z0-9].
2. a-z0-9 -- Can be captured by (a-z0-9) and then can be referenced in a replacement and/or later in the expression.

. is the regex token for any character.

**Problem 01**: /[\s]/

Solution:

var colors = "yellow, green, red";

var patt1 = /[\s]/g;

var result = colors.split(patt1);

console.log(result);

Output: ["yellow,", "green,", "red"]

**Problem 02**: /[,\s]/

Solution:

var colors = "yellow, green, red";

var patt1 = /[,\s]/g;

var result = colors.split(patt1);

console.log(result);

["yellow", "", "green", "", "red"]

Problem 03: /[,\s]+/

**JavaScript RegExp g Modifier:**

Do a global search for "is":

var str = "Is this all there is?";

var patt1 = /is/g;

The g modifier is used to perform a global match (find all matches rather than stopping after the first match).

Tip: To perform a global, case-insensitive search, use this modifier together with the "i" modifier.

**Syntax:**

new RegExp("regexp", "g")

or simply:

/regexp/g

**JavaScript RegExp i Modifier:**

Do a case-insensitive search for "w3public" in a string:

var str = "Visit w3public";

var patt1 = /w3public/i;

**Syntax:**

new RegExp("regexp", "i")

or simply:

/regexp/i

Example 01:

var str = "Visit w3public";

var patt1 = /w3public/i;

var result = str.match(patt1);

Output: The output of the result is: w3public

N.B, without g after the first match it doesn’t care and return one result

Example 02:

var str = "Visit w3public and w3public";

var patt1 = /w3public/gi;

var result = str.match(patt1);

Output: The output of the result is: w3public, w3public

**N.B**, with g after the all match it returns results by comma seperated

JavaScript multiline:

ECMAScript 6 (ES6) introduces a new type of literal, namely template literals. They have many features, variable interpolation among others, but most importantly they can be multiline.

**ECMAScript 6 (ES6) Syntax:**

A template literal is delimited by backticks:

var html = `

<div>

<p>some text here</p>

</div>

`;

**ES5 Syntax:**

"foo \

bar"

You can escape the literal into newline using ‘\’ or ‘\n’.

**N.B**, “\” doesn’t support all browsers

**JavaScript RegExp m Modifier:**

The m modifier is used to perform a multiline matching.

**Syntax:**

new RegExp("regexp", "i")

or simply:

/regexp/im

Example 01:

var str = "\nIs th\nis h\nis?"; // here, ‘\’ or ‘\n’ same

var patt1 = /^is/gmi;

var result = str.match(patt1);

Output: Is,is,is

Example 02:

var str = "\nIs th\nis h\nis?"; // here, ‘\’ or ‘\n’ same

var patt1 = /^is/gi;

var result = str.match(patt1);

Output: No result

**RegExp Methods:**

|  |  |
| --- | --- |
| Method | Description |
| exec() | Tests for a match in a string. Returns the first match |
| test() | Tests for a match in a string. Returns true or false |
| ToString() | Returns the string value of the regular expression |
| match() | Returns comma seperated result |
| split() | Returns an arry of elements |
| replace() |  |

**N.B,** split() and match() are both inverse with one another.

**Quantifiers**: Quantifiers are used for a complex pattern matching expression. This means we have to find some pattrens from a string using the expression.

|  |  |
| --- | --- |
| Quantifier | Description |
| n+ | Means “one or more”, the same as {1,}. Matches an expression one or multiple times or simply at least one. |
| n\* | Means “zero or more”, the same as {0,}. Matches an expression zero or multiple times. If zero then the expression will evaluate as found and the value will be an empty string. |
| n? | Means “zero or one”, the same as {0,1}. Matches an expression zero or one time. If zero then the expression will evaluate as found and the value will be an empty string. |
| n{X} | {X} quantifier is put after a character and X specifies exactly how many we need. |
| n{X,Y} | {X,Y} quantifier is put after a character and X-Y specifies an inclusive range. Like count as from-to limits. |
| n$ | Matches the n character at the end of a string |
| ^n | Matches the n character at the begening of a string |
| ?=n | Matches any string that is followed by a specific string n |
| ?!n | Matches any string that is not followed by a specific string n |

**N.B**, Here, n a character like /abc\*/, here, n=c and {} is called number of times to repeat or simply quantity.

**Example using n?**

var str = "I’ve got a couple of cars in my house";

var regexp = /cars?/g;

var result = str.match(regexp);

console.log(result);

Output: ["cars"]

If we change the string like so "I’ve got a car in my house" then result will remain the same as found ["car"].

So we can see in the regular expression s is an **optional** character.

We can also change the string like as "I’ve got a cas in my house" and also change the regular expression as follows:

/car?s/g;

Then result will remain the same as found ["cas"]. Here, r is an **optional** character.

There is no available option for c, a and s if we use **r?** in the expression.

**Example using n+**

var str = "Today is 23rd Feb.";

var regexp = /\d+r?d?/g;

var result = str.match(regexp);

console.log(result);

**Output**: ["23rd"]

If we change the string like so "Today is 23d Feb." then result will remain the same as found ["23d"].

Even if we change like so "Today is 23 Feb." the result will remain as well like ["23"]

**Let’s see another example:**

var str = "Today";

var regexp = /\S/g; //any string is \S

var result = str.match(regexp);

console.log(result);

Output: ["T", "o", "d", "a", "y"]

“In above result, we can see by default the regular expression assumes every single character is a string.”

Now we change the expression as /**\S+/g**

Then the result will be evaluated as ["Today"] and only one element.

So we can see in the regular expression \S is repeatingmore times using “+” until the string is finished and evaluating a group of string or a brand new string unlike a word.

**Example using n\***

var str = "Today";

var regexp = /\S\*/g;

var result = str.match(regexp);

console.log(result);

Output: ["Today", ""]

We can see there is an extra empty string value and that is evaluated as found.

Basically n+ and n\* are same but a little bit difference is as follows:

\* - Matches an Expression 0 OR More Times.

+ - Matches an Expression 1 OR More Times

Let’s see another example:

If we change the string like as var str = ""; and regular expression as **/\S\*/g**

It has been evaluated as found like [“”] although the string was zero length.

But if we use the expression as **/\S+/g**

Then the result will be evaluated as **null.**

**Example using quantity {}**

var str = "AAAABBBBCCCCDDDD";

var regexp = /A/g;

var result = str.match(regexp);

console.log(result);

Output: ["A", "A", "A", "A"]

But we want only 3 times and then the result would be as follows:

var regexp = /A{3}/g;

Output: ["AAA"]

We can see the above expression a kind of range.

Now we change the string as str = "AAAABBBBAAAACCCCDDDD";

Output: ["AAA","AAA"]

**Now at least 3 times:**

var regexp = /A{3,}/g;

Output: ["AAAA", "AAAA"]

As we can see not exact 3 times but at least 3 times.

**And now at least 3 but not more than 5 times:**

var regexp = /A{3,5}/g;

And change the string as well as follows:

str = "AAAAAABBBBAAAAAACCCCDDDD";

Output: ["AAAAA", "AAAAA"]

As we can see minimum 3 and at most 5 times.

**More Examples:**

Exact count: {5}

\d{5} denotes exactly 5 digits, the same as \d\d\d\d\d.

The example below looks for a 5-digit number:

**Shorthands**

Most often needed quantifiers have shorthands:

**+** Means “one or more”, the same as {1,}.

For instance, \d+ looks for numbers:

let str = "+(880)-173-0910-539";

alert( str.match(/\d+/g) ); // 7,903,123,45,67

**?** Means “zero or one”, the same as {0,1}. In other words, it makes the symbol optional.

So it can find or in the word color and our in colour:

let str = "Should I write color or colour?";

alert( str.match(/colou?r/g) ); // color, colour

**\*** Means “zero or more”, the same as {0,}. That is, the character may repeat any times or be absent.

The example below looks for a digit followed by any number of zeroes:

alert( "100 10 1".match(/\d0\*/g) ); // 100, 10, 1

Compare it with '+' (one or more):

alert( "100 10 1".match(/\d0+/g) ); // 100, 10

what is the difference between ?:, ?! and ?= in regex?

The difference between ?= and ?! is that the former requires the given expression to match and the latter requires it to not match. For example a(?=b) will match the "a" in "ab", but not the "a" in "ac". Whereas a(?!b) will match the "a" in "ac", but not the "a" in "ab".

The difference between ?: and ?= is that ?= excludes the expression from the entire match while ?: just doesn't create a capturing group. So for example a(?:b) will match the "ab" in "abc", while a(?=b) will only match the "a" in "abc". a(b) would match the "ab" in "abc" and create a capture containing the "b".

**At a glance**:

?: is for non capturing group

?= is for positive look ahead

?! is for negative look ahead

?<= is for positive look behind

?<! is for negative look behind

() capturing group - the regex inside the parenthesis must be matched and the match create a capturing group

(?:) non capturing group - the regex inside the parenthesis must be matched but doesn't not create the capturing group

(?=) positive look ahead - asserts that the regex must be matched

(?!) negative look ahead - asserts that it is impossible to match the regex