Regular expressions with Javascript – notes

Using regular expressions in Javascript & flags:

Basic regex pattern

let string = "This is a simple Javascript string";

let rex = /Javascript/;

Using regex to match and test strings in Javascript

Test method:

let check = rex.test(string);

document.write(/Javascript/.test("This is a simple Javascript string"));

(or)

console.log(/Javascript/.test("This is a simple Javascript string"));

Match method:

let check = string.match(rex);

console.log(check);

Regular expression modifiers (flags)

Global flag – gets all instances of the regular expression from the given string

let rex = /Javascript/g;

Case insensitive flag – Makes the search case insensitive (a and A are the same)

let rex = /Javascript/gi;

Multiline flag – can be used to search across multiple lines

let rex = /Javascript/m;

Regular expressions in search and replace

let string = "This is my first javascript Regex. Javascript javascript Javascript";

```
let rex = /Javascript/gi;

document.write(string.search(rex));
```

Now, the replace function.

```
document.write(string.replace(rex, "Typescript"));
```

Advanced regular expression patterns using brackets

Brackets are a great way to start creating advanced regular expressions. You can use brackets to specify a lot of options in your search.

[…] – letters or numbers inside brackets to specify any one of them inside.

```
let string = "Hello there!";

let rex = /[aeiou]/gi;

document.write(string.match(rex));
```

```
let string = "Cat cut the cot";

let rex = /c[aou]t/gi;
```

You can do the same with numbers

```
let string = "124 860 3786 27";

let rex = /[123]/gi;
```

Let’ say we only want numbers that start with 1 and end with 3. Then, this is how it’ll go:

```
let string = "123 145 143 167 153 173 175 857 564";

let rex = /1[0-9]3/gi;
```

Can be any range:

```
let rex = /1[3-5]3/gi;
```

You can do the same with letters too. Specify a range of letters.

let string = "Cat caught the rat on the mat and gave it a pat";

let rex = /[a-z]at/gi;

Again, can be any sequence:

let rex = /[a-p]at/gi;

Can specify the capital letter version of the same as well. That is, specify multiple ranges.

let rex = /[a-zA-Z]at/g;

let string = "Cat caught the rat on the mat and gave it a pat 1at 2at";

let rex = /[a-zA-Z0-9]at/g;

Doesn't have to be in a specific sequence. You can mention any of the ranges anywhere and get the same result.

let rex = /[A-Z0-9a-z]at/g;

Can use the pipe character to specify options as well. To mention that any one of the specified word or letter or letters should be there.

This time, not square brackets, but parenthesis.

let rex = /(cat|mat|pat)/gi;

Can specify letters too:

let rex = /(c|r|m|p)at/gi;

We can also specify letters or sequences or patterns we don't want inside square brackets.

let string = "Cat caught the rat.";

let rex = /[aeiou]/gi;

All vowels in the string displayed.

If you use the carat sign before the letters in the square bracket, it'll take it as negation. So, anything other than the vowels will be displayed.

let rex = /[^aeiou]/gi;

Another example:

```
let string = "cat cut cot cet cyt cbt cit.";

let rex = /c[^eyb]t/gi;
```

You can use carat with number ranges too.

```
let string = "Cat caught the rat on the mat and gave it a pat 1at 2at";

let rex = /[^0-9]at/g;
```

If you don't want the 1at and 2at to be displayed, do the above.

Instead of ranges, can do numbers as well.

```
let rex = /[^123]at/g;
```

Multiple numbers or letters:

So, a string with 3 numbers and a letter at the end:

```
let string = "cat mat 12t 3g 45a cA5 64A";

let rex = /[0-9][0-9][a-Z]/g;

document.write(string.match(rex));
```

Use Metacharacters to enhance your regular expressions

You can use metacharacters in regular expressions to specify something similar to what we saw in the earlier module.

First of all, there is the wildcard character. It is depicted by a single dot. It is used to specify any single character, could be alphabets, numbers or special characters, or even spaces. All of that, except newline or line terminator.

```
let string = "cat";

let rex = /c.t/;

document.write(rex.test(string));
```

Gives true.

```
let string = "c\nt";
```

let string = `c

t`;

Last two gives false.

\w is for all words, a-z, A-Z, 0-9 and underscore, and nothing else.

let string = `Hello world!`;

let rex = /\w/g;

document.write(string.match(rex));

Space and ! is not in the array result.

\W – opposite of word character – everything except words, that is special characters, periods, spaces, newlines etc.

\d – digits

All numbers – every single number is separate array entry

\D – anything except numbers, including special characters, spaces and words

\s – whitespace characters – space, tab, carriage return, newline, vertical tab, form feed

\S – anything except whitespaces (like using ^ sign)

\b – find the rest of the pattern in the beginning or end of the word (not string)

let string = "good cat";

let rex = /\bca/g;

document.write(string.search(rex));

Found “Ca” in the 5th position in the string

In search, if no match is found, it returns -1

Quantifiers, Anchors & More

Anchors – start and end anchors

^pattern – string has to start with pattern

let string = "Good cat";

let rex = /^ca/gi;

document.write(rex.test(string));

Above is false.

let string = "Cat is good";

Above is true, even though C is caps because of the i flag.

Can give patterns too.

Should start with underscore:

let rex = /^_/gi;

Should start with a digit:

let rex = /^\d/gi;

Pattern$ - string ends with pattern

^pattern$ - starts and ends with pattern – contains only that pattern

If a string should contain what's inside the pattern, and only that, then use this. Perfect for password verification, username verification, like that.

Only contain cat in the string:

let rex = /^cat$/i;

Doesn't need global flag at all

Only have 3 numbers, and nothing else:

let rex = /^\d\d\d$/i;

Quantifiers:

a+ - one or more of the character in string

Until infinity:

Say we want string to start with underscore, have one or more letters and end with a digit, then:

let rex = /^_[a-zA-Z]+[0-9]$/i;

\w won't work because we just want letters. But, we can use \d instead of [0-9]

a* - zero or more of the character

a? – optional – 0 or 1

a{2,3} – 2 or 3 of that – ranges – includes both, so a can be repeated 2 or 3 times

Example:

Can have 2 or 3 letters (only small) and end with 3 to 5 digits.

let string = "aaa999";

let rex = /^[a-z]{2,3}[0-9]{3,5}$/i;

Can use \d too

let rex = /^[a-z]{2,3}\d{3,5}$/i;

a{2} – only 2 of that

a{2,} – 2 to infinity

Literal/Special meta characters & Unicode

Matching literal/special characters

Special characters have meaning in regular expressions -> '+' is used for one or more in quantifiers, for example.

So, in order to use them as such in a pattern, i.e, as literal characters, you need to escape them.

For example, if I want to find a++ in my pattern, I can't just say:

/a++/ because + has a special meaning in Javascript regular expressions, so I need to escape each +, like this:

/a\+\+/

There are 12 such special characters that need to be escaped to be used literally in a regular expression. They are:

\ - backslash, ^ - carat, $ - dollar, [– opening square brackets (not the closing one because it usually comes after the opening, so if used alone, it'll be considered a literal character), ( ) – opening & closing parentheses, + - plus sign, * - star sign, . – period, ? – question mark, | - pipe symbol, { - opening flower bracket

{ is mostly treated as a literal character unless it's part of a complete set ({ })

Don't escape other characters with a backlash because then you might accidentally create a metacharacter - \d, \s etc

RegExp object, properties & methods

RegExp object

```
let string = "Cat cat cat";

let rex = new RegExp("cat","gi");

document.write(string.match(rex));
```

```
let rex = new RegExp("[0-9]{2}[a-z]{2,3}","g");
```

[0-9], [a-z] works, but \d, \w etc doesn't work here and they'll display literal letters d, w etc.

```
let string = "Cat pat rat mat mittens bite";
```

```
let rex = new RegExp("(c|p|m)at","gi");
```

RegExp properties

RegExp constructor:

Constructor property returns constructor function of the regular expression:

It only returns a reference to the function, not the function itself.

```
let rex = new RegExp("(c|p|m)at","gi");
let cons = rex.constructor;
document.write(cons);
```

Output: function RegExp() { [native code] }

Global property returns true if the global property is set, false otherwise.

```
let gl = rex.global;
document.write(gl);
```

ignoreCase – checks if “i” modifier is set

lastIndex

Returns the position after the last character in the match, unlike search which returns the position of the first character in the match

Works with exec() and test() properties

Returns 0 if no match is found

Only works if the global variable is set

```
let str = "Cat sat on the mat and ate the rat"
let rex = new RegExp("(c|m|r)at","gi");
while(rex.test(str) == true) {
        document.write(rex.lastIndex + "<br/>");
}
```

Multiline – checks whether multiline flag is set

Source - Returns the text of the regular expression, as such, without the flags

```
let rex = new RegExp("(c|m|r)at","gi");
let src = rex.source;
document.write(src);
```

flags – returns all flags in the regex

```
let rex = new RegExp("(c|m|r)at","gi");
document.write(rex.flags);
```

S flag

By default, the . wildcard character will match every character other than the newline character. With the s flag, you can make it match the newline as well, so literally ANY character.

```
document.write(/./s.test("\n"));
```

DotAll property (google search)

Regex ES2018 update

Specifies whether the s flag is used or not

Returns true if the dotAll property is set, that is, if the . is used and the s flag is set.

Matches “.” with line terminators if the s flag is used.

dotAll only affects the dot, like multiline property only affects ^ and $

```
document.write(/./s.dotAll); //True
document.write(/./.dotAll); //False
```

RegExp methods

toString – returns string value of the regular expression

```
let rex = new RegExp("(c|m|r)at","gi");
let src = rex.toString();
document.write(src);

let rex = /(c|m|r)at/gi;
let src = rex.toString();
document.write(src);
```

Same result for both of the above examples

Exec() – searches for match in string and returns first match

Similar to match method – returns null if no match, array of results if there is match

```
let str = "Cat sat on the mat and ate the rat"
let rex = new RegExp("(c|m|r)at","gi");
document.write(rex.exec(str));
```

Output: Cat, c

If “i” flag is taken out, then will return mat,m

Lazy and greedy matches

```
let str = "Hello there (This is a greeting), how are you? (This is also a greeting)";
let rex = /\(.+\)/gi;
document.write(str.match(rex));
```

Output: (This is a greeting), how are you? (This is also a greeting)

Greedy match by default. Should have gotten 2 different outputs in an array and not the inbetween words, instead got only one that encompassed the whole thing.

Because browser will by default take the maximum possible match in the given string, and not the minimum. That’s called greedy matching.

Lazy matches take the most minimum possible result in the string.

Lazy mode is opposite of greedy mode. If ? is used after quantifier (only works with quantifiers), we are activating lazy mode, and it'll basically instruct the browser to repeat that particular character or set minimum number of times, instead of maximum (which was the default before).

let rex = /\(.+?\)/gi;
Output: (This is a greeting), (This is also a greeting)

Instead of using lazy match, the below will work too:

let rex = /\([^\(]+\)/gi;

In regex, multiple ways of doing the same thing. Just understand and solve the problem in whichever way you choose.

Not always the right choice:

To check phone number that's written in the following pattern:

let str = "123 456 7890";

let rex = /\d+? \d+? \d+?/gi;

document.write(str.match(rex));

Output is 123 456 7

So, lazy match is not always the right choice.

Grouping and Capture groups

Grouping in Javascript Regex (Capture groups)

Capture groups lets you apply quantifiers to groups of characters, rather than just one, as is usual.

For example:

If I want multiple he's to emulate laugh in textual format.

He, hehe, hehehe, hehehehe etc

Then, this is how I'll do it:

let str = "Heheh";

```
let rex = /(he)+/gi;

document.write(rex.test(str));
```

Hehe, he, heh, heheheh, etc

With capture groups, can get a part of the group as a match in the array.

match[0] – full match

match[1] – contents of first parenthesis

match[2] – contents of second parenthesis

```
let str = "Hehehe! Hohoho!";

let rex = /(he)+! (ho)+!/i;

let match = str.match(rex);

document.write(match[0]);
```

match[0] is entire string, match[1] is he, and match[2] is ho

Should not use g flag for the groups to work.

```
let str = "Jane Doe";

let rex = /(\w+) (\w+)/;

let match = str.match(rex);

document.write(match[2]);
```

Nested groups:

Parenthesis can be nested:

Create an example string where there can be sequence of numbers, any number of numbers, separated by spaces, and multiple numbers like that.

```
let str = "123 456 7890";

let rex = /(([0-9]+)\s*)+/i; //starts with numbers, can have spaces or not (last sequence of numbers won't have spaces), and can end with number sequences of not.

let match = str.match(rex);

document.write(match[0]);
```

Then match[1], match[2], etc

match.length is 3

let str = "123 abc";

let rex = /(([0-9]+)\s*)+\s*([a-z]*)/i;

match[0] is for the entire match 123 abc

match[1] is for the first entire one, including \s*, which is 123

match[1] is for just ([0-9]+) which is 123

match[2] is for ([a-z]*) which is abc

Named capture groups

If there are multiple capture groups, remembering what each are is very difficult. So, naming them is convenient.

Immediately after the open parenthesis, place a ?<name>, where name is any name.

Example as phone number format:

Country code, area code, number part 1, number part 2

+1 (425) 555-2551

let str = "+1 (425) 555-2551"; //Should work if +1 is removed as well

let rex = /(?<int>\+[0-9]\s)?(?<area>\([0-9]{3}\)) (?<num1>[0-9]{3})-(?<num2>[0-9]{4})/i; //International code and first space is optional, area code with () needed, then a space (not anything else, so use space and not \s, then first num, then - and then second number and those have fixed ranges.)

document.write(rex.test(str));

Can separate the above by groups as well:

Groups reside in the groups property of the match.

let match = str.match(rex).groups;

document.write(match.int + "
");

document.write(match.area + "
");

document.write(match.num1 + "
");

document.write(match.num2 + "
");

Use them in search and replace too.

Let's rewrite the name format, like this:

```
let str = "Jane Doe";

let rex = /(\w+) (\w+)/;

document.write(str.replace(rex, '$2, $1'))
```

Result is Doe, Jane

$2 is 2nd capture group, $1 is first capture group and we rewrote it in the format we wanted.

Non capturing parentheses

Used to exclude a capture group from results.

"Hehehe Funny!"

If we want to exclude Hehehe, this is how you'll do. Specify ?: at the start of the parenthesis of the capture group.

It'll still be a part of the complete match but won't be a separate array item.

```
let str = "Hehehe Funny!";

let rex = /(?:he)+ (\w+)!/i;

let match = str.match(rex);

document.write(match[0]);
```

match[0[will be Hehehe Funny!

Match[1] will be Funny

Match[2] is undefined

Match.length is 2 because Hehehe is not an array item anymore.

If we use ?: it means that particular group alone is excluded from being capture. It can't be referenced in any way.

Backreferencing a group:

By name: \N

So, if you want a pattern repeated as such, without any change among the options given, then you can use backreferencing.

Example:

Brackets:

@,#,* - the other side should have the same as well

let str = "*Hello*";

let rex = /[@#*]\w+[@#*]/i;

*Hello# etc will match too.

So, to match identical on either side, put the options in capture groups and use backreferencing by number at the end.

let rex = /([@#*])\w+\1/i;

document.write(rex.test(str));

\2 is second capture group, \3 is 3rd capture group and so on.

Can reference by name as well:

let rex = /(?<spl>[@#*])\w+\k<spl>/i;

\k<name>, no need to use ? at the end.

Positive and negative lookahead & Lookbehinds

Positive lookahead

Only matches something that is followed by something else.

let string = "30cm 40in 55mm 60i";

let rex = /\d+(?=in)/;

document.write(string.match(rex));

Negative lookahead

Positive lookbehind

Negative lookbehind

Projects

Project 1: Remove extra spaces at start or end of string or both without using trim()

```
let str = "   Hello  ";

let rex = /(^\s+|\s+$)/g;

document.write(str.replace(rex, ''));
```

Project 2: E-mail validation

//words@words.words or words@words.words.words

Should have word - \w

But can have words.words or words-words as well but those are optional – words as such is compulsory

Then should compulsorily have @

Then, should have domain name – words is compulsory, again, words-words, words.words for domain is optional

Finally end with top level extension – can be 2 to 3 letters.

.com. .in etc

This will match for .co.in, .co.au etc as well (since we gave that option before the final capture group.

```
let string = "a@g.co";
```

```
let rex = /^\w+([\.-]?\w)*@\w+([\.-]?\w)*(\.\w{2,})$/gi;

document.write(rex.test(string));
```

Notes: period is wildcard character, so to specify period as such in pattern, use backslash

Project 3: Mobile number validation

Examples:

+1 (555) 555-5555

(555) 555-5555

555-555-5555

555.555.5555

555 555 5555

All of the above should match, with or without international code

So, international code is optional

Then, (, space or – or . then the numbers in fixed ranges.

```
let string = "(555) 555-5555";

let rex = /^(\+[0-9]+)?[. -]?(\(?[0-9]{3}\)?)[. -]?([0-9]{3})[. -]?([0-9]{4})$/;

document.write(rex.test(string));
```

So, above, +1 or +91 or something is optional. Then space or . or – optional. Then area code within optional (), then numbers with delimiters.

But, if we want the same -,. or space combo throughout, then we can use numbering groups

```
let string = "+1 (555) 555 5555";

let rex = /^(\+[0-9])?([. -]?)(\(?[0-9]{3}\)?)\2?([0-9]{3})\2?([0-9]{4})$/;

document.write(rex.test(string));
```

But the above will only work if the first one after +1 is there in the string. Otherwise, will return false