**Reulgar Expression in javascript**

**1. Using the Test MethodPassed:**

Input: Checks for a exp within a string.

Syntax: Regexp.test(String)

Output: boolean(true/false)

**2. Match a Literal String with Different Possibilities:**

Input: Checks for several possibilities using a or operator.

Syntax: /option 1|option 2/

Output: boolean(true/false)

**3. Ignore Case While Matching:**

Input: Checks for the regexp ignoring the case.

Syntax: /regexp/i i=ignorecase flag

Output: boolean(true/false)

**4. Extract Matches:**

Input: Not only checks, it also displays/extracts the matched portion.

Syntax: String.match(Regexp)

Output: Matched regexp.

**5. Find More Than the First Match:**

Input: Search for more than one match.

Syntax: /regexp/g g=global flag

Output: All available matches in the String.

**6. Match Anything with Wildcard Period:**

If we don’t know any character in our pattern,we can use wildcard period. It is also known as dot period or wildcard character,it matches for only one character.

Syntax: /x.y/

Ex: /b.g/ can check for big/bag/bag/beg...simply b[a-z0-9]g anything in between b and g.

**7. Match Single Character with Multiple Possibilities:**

If we don’t know any character in our pattern,we can use Character class,thats quite similar to wildcard period,but in some balanced manner,no need to know the exact exp though.

Syntax: /x[a-z]y/

Ex: /b[aeiu]g/ can similarly search for bag/beg/big/bug.

No need know the exact middle letter or use annonymous dot period.

**8. Match Letters of the Alphabet:** [a-z]

**9. Match Numbers and Letters of the Alphabet:** [a-z0-9]

**10. Match Single Characters Not Specified:**

Input: Find a match ignoring a specified regexp.

Syntax: [^regexp]

Ex: /[^aeiou]/ returns everything except the vowels.

**11. Match Characters that Occur One or More Times:**

Input: Find consecutive one or more matches in a string.

Syntax:/regexp+/

Ex: ‘ab’ will find a match and return an array consisting [‘a’]

‘aab’ “ “ “ “ “ “ “ “ “ [‘aa’]

‘abab’” “ two “-es “ “ “ “ “ [‘a’,’a’]

**12. Match Characters that Occur Zero or More Times:**

Input: Same as ‘+’ operator in regexp just it also finds for zero matches. That means the specified character in regexp is not mandatory,it may not appear in strings or can be consecutive in number in a string

Syntax: /regexp\*/

Ex:

let soccerWord = "gooooooooal!";

let gPhrase = "gut feeling";

let oPhrase = "over the moon";

let goRegex = /go\*/;

soccerWord.match(goRegex); // Returns ["goooooooo"]

gPhrase.match(goRegex); // Returns ["g"]

oPhrase.match(goRegex); // Returns null

**13. Find Characters with Lazy Matching:**

In regular expressions, a greedy match finds the longest possible part of a string that fits the regex pattern and returns it as a match. The alternative is called a lazy match, which finds the smallest possible part of the string that satisfies the regex pattern.

Syntax & EX: Lets there is a string “titnic”

/t[a-z0-9]\*c/gi - matches anything that starts with t and ends with c with any letters or numbers in between and returns it in an array with[“titanic”]. Its a greedy approach,regexp are by defualt greedy.

Now /t[a-z0-9]\*?c/gi – It also matches anything that starts with t and ends with c with any letters or numbers in between but only returns an array with only [“tc”].

**14. Match Beginning String Patterns:** /^regexp/

**15. Match Ending String Patterns:** /regexp$/

**15. Match All Letters and Numbers:** /\w/

**16. Match Everything But Letters and Numbers:** /\W/

**17. Match All Numbers:** /\d/

**18. Match All Non Numbers:** /\D/

**19. Match Whitespaces:** /\s/

**20. Match All Non Whitespace Characters:** /\S/

**21. Specify Upper and Lower Number of Matches:**

We use the plus sign + to look for one or more characters and the asterisk \* to look for zero or more characters. These are convenient but sometimes we want to match a certain range of patterns.

We can specify the lower and upper number of patterns with quantity specifiers. Quantity specifiers are used with curly brackets ({ and }). We put two numbers between the curly brackets - for the lower and upper number of patterns.

Syntax & Ex : /{lowerlimit,upperlimit}/

let ohStr = "Ohhh no";

let ohRegex = /oh{3,6} no/gi; // Checks for “oh no” string with let result = ohRegex.test(ohStr);//‘h’ amounting between 3 & 6.

**22. Specify Lower Number of Matches:** /{lowerlimit,}/

**23. Specify Exact Number of Matches:** /{exactlimit}/

**24. Check for All or None:**

Sometimes the patterns we want to search for may have parts of it that may or may not exist. However, it may be important to check for them nonetheless.

We can specify the possible existence of an element with a question mark, ?. This checks for zero or one of the preceding element. We can think of this symbol as saying the previous element is optional.

Syntax & Ex: /regexp?/

let favWordBritish = "favorite";

let favWordAmerican = "favourite";

let favRegex = /favou?rite/;

let british = favRegex.test(favWordBritish);//returns true for british english

let amrican = favRegex.test(favWordAmerican);//returns true for american english

**25. Positive and Negative Lookahead:**

Lookaheads are patterns that tell JavaScript to look-ahead in your string to check for patterns further along. This can be useful when you want to search for multiple patterns over the same string.

There are two kinds of lookaheads: positive lookahead and negative lookahead.

Syntax:

A positive lookahead will look to make sure the element in the search pattern is there, but won't actually match it. A positive lookahead is used as (?=...) where the ... is the required part that is not matched.

On the other hand, a negative lookahead will look to make sure the element in the search pattern is not there. A negative lookahead is used as (?!...) where the ... is the pattern that you do not want to be there. The rest of the pattern is returned if the negative lookahead part is not present.

Ex:

let sampleWord = "";

let pwRegex = /(?=^\D+\w{5,})(?=\w\*\d\d)/g;

let result = pwRegex.test(sampleWord);

Here is a simple password checker,that match passwords that are greater than 4 characters long, do not begin with numbers, and have two consecutive digits at the end.

**26. Check For Mixed Grouping of Characters:**

Its similar to no 2 doc,that is **Match a Literal String with Different Possibilities**, added feature is just we can also search for a group of exp, like: /regexp1(regexp\_op1|regex\_op2)regexp2/

Ex:

let myString = "";

let myRegex = /(Eleanor|Franklin).\*Roosevelt/;

let result = myRegex.test(myString);

**28. Reuse Patterns Using Capture Groups:**

Some patterns you search for will occur multiple times in a string. It is wasteful to manually repeat that regex. There is a better way to specify when you have multiple repeat substrings in your string.

You can search for repeat substrings using capture groups. Parentheses, ( and ), are used to find repeat substrings. You put the regex of the pattern that will repeat in between the parentheses.

To specify where that repeat string will appear, you use a backslash (\) and then a number. This number starts at 1 and increases with each additional capture group you use. An example would be \1 to match the first group.

Using the .match() method on a string will return an array with the string it matches, along with its capture group.

Ex:

let repeatNum = "42 42 42";

let reRegex = /^(\d+)\s\1\s\1$/;

let result = reRegex.test(repeatNum);

console.log(result)

This reRegex match numbers that are repeated only three times in a string, each separated by a space.

**29. Use Capture Groups to Search and Replace:**

Searching is useful. However, we can make searching even more powerful when it also changes (or replaces) the text we match.

We can search and replace text in a string using .replace() on a string. The inputs for .replace() is first the regex pattern you want to search for. The second parameter is the string to replace the match or a function to do something.

Ex:

let wrongText = "The sky is silver.";

let silverRegex = /silver/;

wrongText.replace(silverRegex, "blue")

// Returns "The sky is blue."

This replace() takes the string, search for the regexp, and replace it by the given exp.

We can also access capture groups in the replacement string with dollar signs ($).

"Sayan Ghosh".replace(/(\w+)\s(\w+)/, '$2 $1');

// Returns "Ghosh Sayan"