Day 12 - Regular Expressions

Regular Expressions

A RegExp is a small programming languages that helps in finding pattern in data. Can be used to check if some pattern exists in different data types. We use the RegExp constructor or declare a RegExp pattern using two forward slashes followed by a flag to use RegExp in JS. Pattern can be created in two ways.

To declare a string, we use a single quote, double quote, or a backtick. To declare a regular expressions, we use two forward slashes and an optional flag. The flag can be g, i, m, s, u or y.

RegExp parameters

A regular expressions takes two parameters, one being a required search pattern and the other an optional flag.

Pattern

Could be text or any form of pattern with some sort of similarity. For example, the word spam in an email could be a pattern we are interested to look for, or a phone number format.

Flags

Optional parameters in a regular expression that determines the type of searching:

- g: a global flag, meaning the whole text will be searched for the pattern.
- i: case insensitive, searches for both lowercase and uppercase
- m: multiline

Creating a pattern with RegExp Constructor

Declaring regular expression without global flag and case insensitive flag

```
// without flag
let pattern = 'love'
let regEx = new RegExp(pattern)
```

Declaring regular expression with global flag and case insensitive flag

```
let pattern = 'love'
let flag = 'gi'
let regEx = new RegExp(pattern, flag)
```

Declaring a regex pattern using RegExp object. Writing pattern and flag inside RegExp constructor

```
let regEx = new RegExp('love', 'gi')
```

Creating a pattern without RegExp Constructor

Declaring regular expression with global flag and case insensitive flag

```
let regEx = /love/gi
// is the same as
let regEx = new RegExp('love','gi')
```

RegExp Object Methods

Testing for a match

test(): Test for a match in a string. Returns true or false

```
const str = 'I love JavaScript'
const pattern = /love/
const result = pattern.test(str)
console.log(result) // true
// can also do /love/.test(str)
```

Array containing all of the match

match(): Returns array containing all of the matches, including capturing groups, or null if no match is found. If global flag not used, match() returns an array containing the pattern, index, input and group.

```
const str = 'I love JavaScript'
const pattern = /love/
const result = str.match(pattern)
console.log(result) // ["love", index: 2, input: "I love JavaScript", groups: undefined]
```

```
const str = 'I love JavaScript'
const pattern = /love/g
const result = str.match(pattern)
console.log(result) // ["love"]
```

search(): Tests for a match in a string. Returns the index of the match, or -1 if the search fails.

```
const str = 'I love JavaScript'
const pattern = /JavaScript/g
const result = str.search(pattern)
console.log(result) // 7
```

Replacing a substring

replace(): Executes search for a match in a string, and replaces matched substring with a replacement substring.

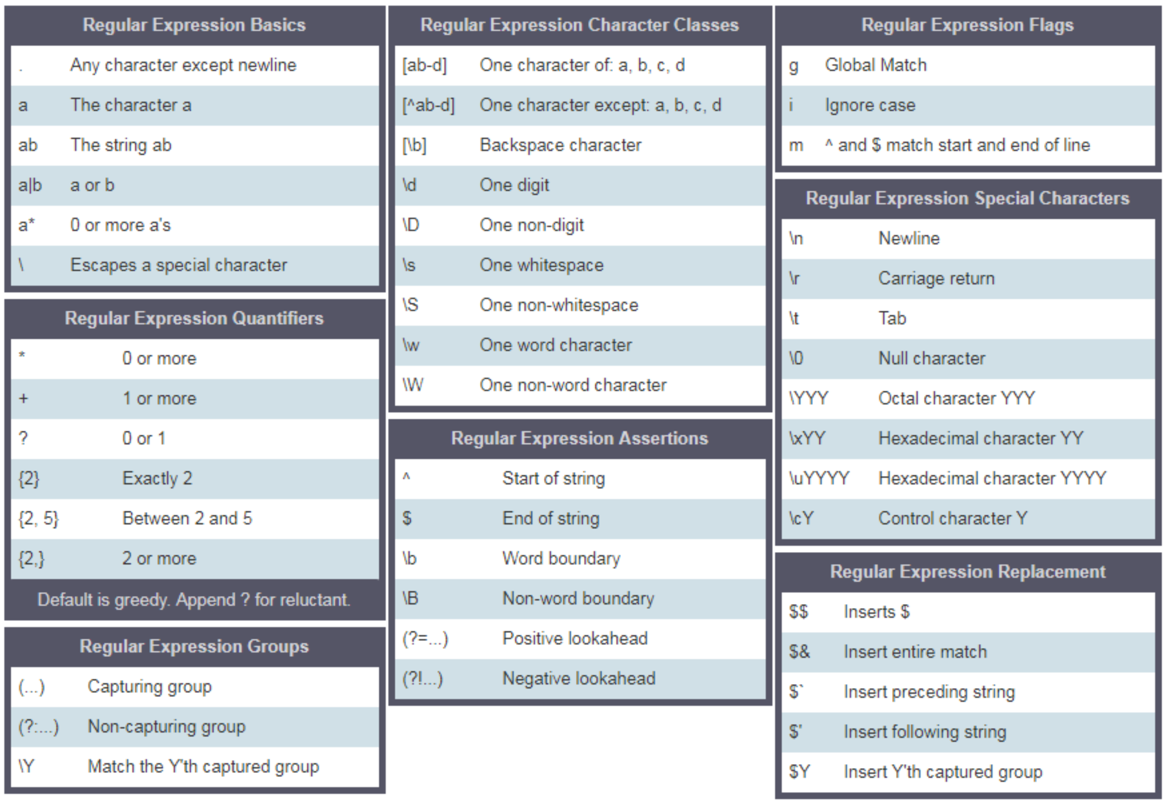
```
const txt = 'Python is the most beautiful language that a human begin has ever created.\
I recommend python for a first programming language'

const matchReplaced = txt.replace(/Python|python/, 'JavaScript')
console.log(matchReplaced)
// JavaScript is the most beautiful language that a human begin has ever created.I recommend python for a first programming language
// Changing all occurences of 'python'
const matchReplaced2 = txt.replace(/Python|python/g, 'JavaScript')
console.log(matchReplaced2)
// JavaScript is the most beautiful language that a human begin has ever created.I recommend JavaScript for a first programming language
// Using case insensitive flag
const matchReplaced3 = txt.replace(/Python/gi, 'JavaScript')
console.log(matchReplaced3)
// JavaScript is the most beautiful language that a human begin has ever created.I recommend JavaScript for a first programming language
```

```
const txt = '%I a%m te%%a%%che%r% a%n%d %% I l%o%ve te%ach%ing.\
T%he%re i%s n%o%th%ing as m%ore r%ewarding a%s e%duc%at%i%ng a%n%d e%m%p%ow%er%ing \
p%e%o%ple.\
I fo%und te%a%ching m%ore i%n%t%er%%es%ting t%h%an any other %jobs.\
D%o%es thi%s m%ot%iv%a%te %y%o%u to b%e a t%e%a%cher.'

const matches = txt.replace(/%/g, '')
console.log(matches)
// I am teacher and  I love teaching.There is nothing as more rewarding as educating and empowering people.I found teaching more interestin
```

- []: A set of characters
    - [a-c] means a, b, or c
    - [a-z] means any letter a to z
    - [A-Z] means any letter A to Z
    - [0-3] means 0, 1, 2, or 3
    - [0-9] means any number 0 to 9
    - [A-Za-z0-9] means any character A to Z, a to z, or 0 to 9
- \: uses to escape special characters
    - \d means to match where the string contains digits (numbers from 0-9)
    - \D means to match where the string does not contain digits.
- .: Any character except new line character (\n)
- ^: starts with
    - r'^love' a sentence which starts with the word love
    - r'[^abc] means not a, b, or c
- $: ends with
    - r'love$' a sentence which ends with the world love
- *: zero or more times
    - r'[a]*' means a is optional or it can occur many times
- +: one or more times
    - r'[a]+' means at least once or more times
- ?: zero or one times
    - r'[a]?' means zero times or once
- \b: word bounder, matches with the beginning or ending of a word
- {3}: Exactly 3 characters
- {3,}: At least 3 characters
- {3,8}: 3 to 8 characters
- |: Either or
    - r'apple|banana' mean either an apple or a banana
- (): Capture and group



Regular Expression Basics	
.	Any character except newline
a	The character a
ab	The string ab
a|b	a or b
a*	0 or more a's
\	Escapes a special character

Regular Expression Quantifiers	
*	0 or more
+	1 or more
?	0 or 1
{2}	Exactly 2
{2, 5}	Between 2 and 5
{2,}	2 or more
Default is greedy. Append ? for reluctant.	

Regular Expression Groups	
(...)	Capturing group
(?:...)	Non-capturing group
\Y	Match the Y'th captured group

Regular Expression Character Classes	
[ab-d]	One character of: a, b, c, d
[^ab-d]	One character except: a, b, c, d
[\b]	Backspace character
\d	One digit
\D	One non-digit
\s	One whitespace
\S	One non-whitespace
\w	One word character
\W	One non-word character

Regular Expression Assertions	
^	Start of string
$	End of string
\b	Word boundary
\B	Non-word boundary
(?=...)	Positive lookahead
(?!...)	Negative lookahead

Regular Expression Flags	
g	Global Match
i	Ignore case
m	^ and $ match start and end of line

Regular Expression Special Characters	
\n	Newline
\r	Carriage return
\t	Tab
\0	Null character
\YYY	Octal character YYY
\xYY	Hexadecimal character YY
\uYYYY	Hexadecimal character YYYY
\cY	Control character Y

Regular Expression Replacement	
$$	Inserts $
$&	Insert entire match
$`	Insert preceding string
$'	Insert following string
$Y	Insert Y'th captured group

Square Bracket

We'll use square bracket to include upper and lower case.

```
const pattern = '[Aa]pple' // square bracket means either A or a
const txt = 'Apple and banana are fruits. An old cliche says an apple a day keeps the  doctor way has been replaced by a banana a day keeps
const matches = txt.match(pattern)

console.log(matches)
// ["Apple", index: 0, input: "Apple and banana are fruits. An old cliche says an apple a day keeps the  doctor way has been replaced by a
```

```
const pattern = /[Aa]pple/g
const txt = 'Apple and banana are fruits. An old cliche says an apple a day a doctor way has been replaced by a banana a day keeps the doct
const matches = txt.match(pattern)

console.log(matches)
["Apple", "apple"]
```

If we want to look for banana, we write this pattern:

```
const pattern = /[Aa]pple|[Bb]anana/g
const txt = 'Apple and banana are fruits. An old cliche says an apple a day a doctor way has been replaced by a banana a day keeps the doct
const matches = txt.match(pattern)

console.log(matches)
// ["Apple", "banana", "apple", "banana", "Banana"]
```

Using the square bracket and or operator, we were able to extract, Apple, apple, Banana and banana.

Escape character(\) in RegExp

```
const pattern = /\d/g // d is a special character which means digits
const txt = 'This regular expression example was made in January 12, 2020.'
const matches = txt.match(pattern)

console.log(matches)
// ["1", "2", "2", "0", "2", "0"], this is not what we want
```

One or more times(+)

```
const pattern = /\d+/g
const txt = 'This regular expression example was made in January 12,  2020.'
const matches = txt. match(pattern)
console.log(matches)  // ["12", "2020"]
```

Period(.)

```
const pattern = /[a]./g // square bracket means a and . means any character except new line
const txt = 'Apple and banana are fruits'
const matches = txt.match(pattern)

console.log(matches)  // ["an", "an", "an", "a ", "ar"]
```

```
const pattern = /[a].+/g // . means any character, + means any character one or more times
const txt = 'Apple and banana are fruits'
const matches = txt.match(pattern)

console.log(matches) // ['and banana are fruits']
```

Zero or more times(*)

Zero or many times. Pattern may not occur or it can occur many times.

```
const pattern = /[a].*/g // . any character, + any character one or more times
const txt = 'Apple and banana are fruits'
const matches = txt.match(pattern)

console.log(matches) // ['and banana are fruits']
```

Zero or one times(?)

Zero or one times. Pattern may not occur or it may occur once.

```
const txt = 'I am not sure if there is a convention how to write the word e-mail.\
Some people write it email others may write it as Email or E-mail.'
const pattern = /[Ee]-?mail/g // ? means optional (- is optional)
matches = txt.match(pattern)

console.log(matches) // ["e-mail", "email", "Email", "E-mail"]
```

Quantifier in RegExp

We can specify length of substring we look for in a text using a curly bracket. Imagine we are interested in a substring whose length is 4 characters.

```
const txt = 'This regular expression example was made in December 6,  2019.'
const pattern = /\\b\w{4}\b/g // exactly four character words
const matches = txt.match(pattern)
console.log(matches) // ['This', 'made', '2019']
```

```js
const txt = 'This regular expression example was made in December 6,  2019.'
const pattern = /\b[a-zA-Z]{4}\b/g  //  exactly four character  words without numbers
const matches = txt.match(pattern)
console.log(matches)  //['This', 'made']
```

```js
const txt = 'This regular expression example was made in December 6,  2019.'
const pattern = /\d{4}/g  // a number and exactly four digits
const matches = txt.match(pattern)
console.log(matches)  // ['2019']
```

```js
const txt = 'This regular expression example was made in December 6,  2019.'
const pattern = /\d{1,4}/g   // 1 to 4
const matches = txt.match(pattern)
console.log(matches)  // ['6', '2019']
```

Cart ^

Starts with

```js
const txt = 'This regular expression example was made in December 6,  2019.'
const pattern = /^This/ // ^ means starts with
const matches = txt.match(pattern)
console.log(matches)  // ['This']
```

Negation

```js
const txt = 'This regular expression example was made in December 6,  2019.'
const pattern = /[^A-Za-z,. ]+/g  // ^ in set character means negation, not A to Z, not a to z, no space, no comma no period
const matches = txt.match(pattern)
console.log(matches)  // ["6", "2019"]
```

Exact match

Should have ^ starting and $ which is an end

```js
let pattern = /^[A-Z][a-z]{3,12}$/;
let name = 'Chris';
let result = pattern.test(name)

console.log(result) // true
```