Chapter 31: Regular expressions

Flags	Details
g	**g**lobal. All matches (don't return on the first match).
m	**m**ulti-line. Causes **^** & **$** to match the begin/end of each line (not only begin/end of string).
i	**i**nsensitive. Case insensitive match (ignores case of [a-zA-Z]).
u	**u**nicode : Pattern strings are treated as **UTF-16**. Also causes escape sequences to match Unicode characters.
y	stick**y**: matches only from the index indicated by the lastIndex property of this regular expression in the target string (and does not attempt to match from any later indexes).

Section 31.1: Creating a RegExp Object

Standard Creation

It is recommended to use this form only when creating regex from dynamic variables.

Use when the expression may change or the expression is user generated.

```
var re = new RegExp(".*");
```

With flags:

```
var re = new RegExp(".*", "gmi");
```

With a backslash: (this must be escaped because the regex is specified with a string)

```
var re = new RegExp("\\w*");
```

Static initialization

Use when you know the regular expression will not change, and you know what the expression is before runtime.

```
var re = /.*/;
```

With flags:

```
var re = /.*/gmi;
```

With a backslash: (this should not be escaped because the regex is specified in a literal)

```
var re = /\w*/;
```

Section 31.2: RegExp Flags

There are several flags you can specify to alter the RegEx behaviour. Flags may be appended to the end of a regex literal, such as specifying `gi` in `/test/gi`, or they may be specified as the second argument to the `RegExp` constructor, as in `new RegExp('test', 'gi')`.

`g` - Global. Finds all matches instead of stopping after the first.

`i` - Ignore case. `/[a-z]/i` is equivalent to `/[a-zA-Z]/`.

`m` - Multiline. `^` and `$` match the beginning and end of each line respectively treating `\n` and `\r` as delimiters instead of simply the beginning and end of the entire string.

Version ≥ 6

`u` - Unicode. If this flag is not supported you must match specific Unicode characters with `\uXXXX` where `XXXX` is the character's value in hexadecimal.

`y` - Finds all consecutive/adjacent matches.

Section 31.3: Check if string contains pattern using .test()

```
var re = /[a-z]+/;
if (re.test("foo")) {
    console.log("Match exists.");
}
```

The `test` method performs a search to see if a regular expression matches a string. The regular expression `[a-z]+` will search for one or more lowercase letters. Since the pattern matches the string, “match exists” will be logged to the console.

Section 31.4: Matching With .exec()

Match Using `.exec()`

`RegExp.prototype.exec(string)` returns an array of captures, or `null` if there was no match.

```
var re = /([0-9]+)[a-z]+/;
var match = re.exec("foo123bar");
```

`match.index` is 3, the (zero-based) location of the match.

`match[0]` is the full match string.

`match[1]` is the text corresponding to the first captured group. `match[n]` would be the value of the *n*th captured group.

Loop Through Matches Using `.exec()`

```
var re = /a/g;
var result;
while ((result = re.exec('barbatbaz')) !== null) {
    console.log("found '" + result[0] + "', next exec starts at index '" + re.lastIndex + "'");
}
```

Expected output

> found 'a', next exec starts at index '2'
> found 'a', next exec starts at index '5'
> found 'a', next exec starts at index '8'

Section 31.5: Using RegExp With Strings

The String object has the following methods that accept regular expressions as arguments.

- `"string".match(...`
- `"string".replace(...`
- `"string".split(...`
- `"string".search(...`

Match with RegExp

```
console.log("string".match(/[i-n]+/));
console.log("string".match(/(r)[i-n]+/));
```

Expected output

> Array ["in"]
> Array ["rin", "r"]

Replace with RegExp

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Expected output

> strfoog

Split with RegExp

```
console.log("stringstring".split(/[i-n]+/));
```

Expected output

> Array ["str", "gstr", "g"]

Search with RegExp

`.search()` returns the index at which a match is found or -1.

```
console.log("string".search(/[i-n]+/));
console.log("string".search(/[o-q]+/));
```

Expected output

> 3
> -1

Section 31.6: RegExp Groups

JavaScript supports several types of group in its Regular Expressions, *capture groups*, *non-capture groups* and *look-aheads*. Currently, there is no *look-behind* support.

Capture

Sometimes the desired match relies on its context. This means a simple *RegExp* will over-find the piece of the *String*

that is of interest, so the solution is to write a capture group `(pattern)`. The captured data can then be referenced as...

- String replacement `"$n"` where `n` is the *n th* capture group (starting from `1`)
- The *n th* argument in a callback function
- If the *RegExp* is not flagged `g`, the *n+1 th* item in a returned `str.match` *Array*
- If the *RegExp* is flagged `g`, `str.match` discards captures, use `re.exec` instead

Say there is a *String* where all `+` signs need to be replaced with a space, but only if they follow a letter character. This means a simple match would include that letter character and it would also be removed. Capturing it is the solution as it means the matched letter can be preserved.

```
let str = "aa+b+cc+1+2",
    re = /([a-z])\+/g;

// String replacement
str.replace(re, '$1 '); // "aa b cc 1+2"
// Function replacement
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

Non-Capture

Using the form `(?:pattern)`, these work in a similar way to capture groups, except they do not store the contents of the group after the match.

They can be particularly useful if other data is being captured which you don't want to move the indices of, but need to do some advanced pattern matching such as an OR

```
let str = "aa+b+cc+1+2",
    re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Look-Ahead

If the desired match relies on something which follows it, rather than matching that and capturing it, it is possible to use a look-ahead to test for it but not include it in the match. A positive look-ahead has the form `(?=pattern)`, a negative look-ahead (where the expression match only happens if the look-ahead pattern did not match) has the form `(?!pattern)`

```
let str = "aa+b+cc+1+2",
    re = /\+(?=[a-z])/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Section 31.7: Replacing string match with a callback function

`String#replace` can have a function as its second argument so you can provide a replacement based on some logic.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
```

```
  }
});
// will return Start string End
```

One line template library

```
let data = {name: 'John', surname: 'Doe'}
"My name is {surname}, {name} {surname}".replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);

// "My name is Doe, John Doe"
```

Section 31.8: Using Regex.exec() with parentheses regex to extract matches of a string

Sometimes you doesn't want to simply replace or remove the string. Sometimes you want to extract and process matches. Here an example of how you manipulate matches.

What is a match ? When a compatible substring is found for the entire regex in the string, the exec command produce a match. A match is an array compose by firstly the whole substring that matched and all the parenthesis in the match.

Imagine a html string :

```
<html>
<head></head>
<body>
  <h1>Example</h1>
  <p>Look at this great link : <a href="http://goalkicker.com">goalkicker</a>
http://anotherlinkoutsideatag</p>
  Copyright <a href="https://stackoverflow.com">Stackoverflow</a>
</body>
```

You want to extract and get all the links inside an `a` tag. At first, here the regex you write :

```
var re = /<a[^>]*href="https?:\/\/.*"[^>]*>[^<]*<\/a>/g;
```

But now, imagine you want the `href` and the `anchor` of each link. And you want it together. You can simply add a new regex in for each match **OR** you can use parentheses :

```
var re = /<a[^>]*href="(https?:\/\/.*)"[^>]*>([^<]*)<\/a>/g;
var str = '<html>\n    <head></head>\n    <body>\n        <h1>Example</h1>\n        <p>Look at this
great link: <a href="http://goalkicker.com">goalkicker</a> http://anotherlinkoutsideatag</p>\n\n
     Copyright <a href="https://stackoverflow.com">Stackoverflow</a>\n    </body>\';\n';
var m;
var links = [];

while ((m = re.exec(str)) !== null) {
    if (m.index === re.lastIndex) {
        re.lastIndex++;
    }
    console.log(m[0]); // The all substring
    console.log(m[1]); // The href subpart
    console.log(m[2]); // The anchor subpart

    links.push({
      match : m[0],    // the entire match
      href : m[1],     // the first parenthesis => (https?:\/\/.*)
```

```
      anchor : m[2],  // the second one => ([^<]*)
    });
}
```

At the end of the loop, you have an array of link with `anchor` and `href` and you can use it to write markdown for example :

```
links.forEach(function(link) {
  console.log('[%s](%s)', link.anchor, link.href);
});
```

To go further :

- Nested parenthesis