JavaScript: an overview of the regular expression API

This post gives an overview of the JavaScript API for regular expressions. It does not, however, go into details about regular expression syntax, so you should already be familiar with it.

Regular expression syntax

Listed below are constructs that are hard to remember (not listed are things like * for repetition, capturing groups, etc.).

- Escaping: the backslash escapes special characters, including the slash in regular expression literals (see below) and the backslash itself.
    - If you specify a regular expression in a string you must escape twice: once for the string literal, once for the regular expression. For example, to just match a backslash, the string literal becomes "\\\\".
    - The backslash is also used for some special matching operators (see below).
- Non-capturing group: (?:x) works like a capturing group for delineating the subexpression x, but does not return matches and thus does not have a group number.
- Positive look-ahead: x(?=y) means that x matches only if it is followed by y. y itself is not counted as part of the regular expression.
- Negative look-ahead: x(?!y) the negated version of the previous construct: x must not be followed by y.
- Repetitions: {n} matches exactly n times, {n,} matches at least n times, {n,m} matches at least n, at most m times.
- Control characters: \cX matches Ctrl-X (for any control character X), \n matches a linefeed, \r matches a carriage return.
- Back reference: \n refers back to group n and matches its contents again.

Examples:

```
> /(a+)b\1/.test("aaba")
true
> /^(a+)b\1/.test("aaba")
false
> var tagName = /<([^>]+)>[^<]*<\/\1>/;
> tagName.exec("<b>bold</b>")[1]
'b'
> tagName.exec("<strong>text</strong>")[1]
'strong'
> tagName.exec("<strong>text</stron>")
null
```

Create a regular expression

Regular expression literal:	`var regex = /xyz/;`	(compiled at load time)
Regular expression object:	`var regex = new RegExp("xzy");`	(compiled at runtime)

Flags modify matching behavior.

g global	The given regular expression is matched multiple times. Matters mainly for search and replace.
i ignoreCase	Case is ignored when trying to match the given regular expression.
m multiline	In multiline mode, the begin and end operators ^ and $ work for each line, instead of for the complete input string.

Examples:

```
> /abc/.test("ABC")
false
> /abc/i.test("ABC")
true
```

Simple matching

Use the following methods to find out whether a regular expression matches a string.

- `regex.test([str])`: returns a boolean indicating whether the match succeeded. If the argument is omitted, the last input string is used again.
- `str.search(regex)`: returns the index where a match was found, `-1` otherwise.

Examples:

```
> var regex = /^(a+)b\1$/;
> regex.test("aabaa")
true
> regex.test("aaba")
false
```

Capture groups, optionally repeatedly

Find all matches for a given regular expression in an input string. The following regular expression method and string method are largely equivalent.

- `var matchData = regex.exec([str])`
  Missing parameter means: reuse argument from last invocation
- `var matchData = str.match(regex)`

`regex`: has the following properties.

- Flags: boolean values indicating what flags are set.
    - `global`: is flag `g` set?
    - `ignoreCase`: is flag `i` set?
    - `multiline`: is flag `m` set?
- Data about the last match:
    - `source`: the complete input string.
    - `lastIndex`: the index where to continue the search next time.

`matchData: null` if there wasn't a match. Otherwise, an array and two additional properties.

- Properties:
    - `input`: The complete input string.
    - `index`: The index where the match was found.
- Array: whose length is the number of capturing groups plus one.
    - 0: The match for the complete regular expression (group 0, if you will).
    - n ≥ 1: The capture of group n.

Invoke once: Flag `global` is not set.

```
> var regex = /a(b+)a/;
    > regex.exec("_abbba_aba_")
    [ 'abbba'
    , 'bbb'
    , index: 1
    , input: '_abbba_aba_'
    ]
    > regex.lastIndex
    0
```

Invoke repeatedly: Flag `global` is set.

```
> var regex = /a(b+)a/g;
    > regex.exec("_abbba_aba_")
    [ 'abbba'
    , 'bbb'
    , index: 1
    , input: '_abbba_aba_'
    ]
    > regex.lastIndex
    6
    > regex.exec()
    [ 'aba'
    , 'b'
    , index: 7
    , input: '_abbba_aba_'
    ]
    > regex.exec()
    null
```

Loop over matches.

```
 var regex = /a(b+)a/g;
    var str = "_abbba_aba_";
    while(true) {
        var match = regex.exec(str);
        if (!match) break;
        console.log(match[1]);
    }
```

Output:

```
bbb
    b
```

Search and replace

Invocation: `str.replace(search, replacement)`.

- `search`:
  - either a string (to be found literally, has no groups)
  - or a regular expression.
- `replacement`:
  - either a string describing how to replace what has been found
  - or a function that computes a replacement, given matching information.

Replacement is a string. The dollar sign $ is used to indicate special replacement directives:

- `$$` inserts a dollar sign $.
- `$&` inserts the complete match.
- `$`` inserts the text before the match.
- `$'` inserts the text after the match.
- `$n` inserts group n from the match. n must be at least 1, `$0` has no special meaning.

Examples:

```
> "a1b_c1d".replace("1", "[$`-$&-$']")
'a[a-1-b_c1d]b_c1d'
> "a1b_c1d".replace(/1/, "[$`-$&-$']")
'a[a-1-b_c1d]b_c1d'
> "a1b_c1d".replace(/1/g, "[$`-$&-$']")
'a[a-1-b_c1d]b_c[a1b_c-1-d]d'
```

Replacement is a function. The replacement function has the following signature.

```
function(completeMatch, group_1, ..., group_n, offset, inputStr) { ...
```

`completeMatch` is the same as `$&` above, `offset` indicates where the match was found, and `inputStr` is what is being matched against. Thus, the special variable `arguments` inside the function starts with the same data as the result of the `exec()` method.

Example:

```
> "I bought 3 apples and 5 oranges".replace(
      /[0-9]+/g,
      function(match) { return 2 * match; })
'I bought 6 apples and 10 oranges'
```

Splitting strings

Signature:

```
str.split(separator, [limit])
```

In a string, find the substrings between the separators and return them in an array.

- `separator` can be
  - a string: separators are matched verbatim
  - a regular expression: for more flexible separator matching. Many JavaScript implementations include the first capturing group in the result array, if there is one.
- `limit` optionally specifies a maximum length for the returned array. A value less than 0 allows arbitrary lengths.

Examples:

```
> "aaa*a*".split("a*")
    [ 'aa', '', '' ]
    > "aaa*a*".split(/a*/)
    [ '', '*', '*' ]
    > "aaa*a*".split(/(a*)/)
    [ '', 'aaa', '*', 'a', '*' ]
```

Sources:

- ECMAScript Language Specification, 5th edition.
- Regular Expressions at the Mozilla Developer Network Doc Cente

References

Your rating: Axel Rauschmayer is a research assistant at the computer science department of the University of Munich. His interests include software engineering, semantic web, and web development. Axel is a DZone MVB and is not an employee of DZone and has posted 80 posts at DZone. You can read more from them at their website.

(Note: Opinions expressed in this article and its replies are the opinions of their respective authors and not those of DZone, Inc.)