29th October 2019

Capturing groups

A part of a pattern can be enclosed in parentheses `(...)`. This is called a “capturing group”.

That has two effects:

1. It allows to get a part of the match as a separate item in the result array.
2. If we put a quantifier after the parentheses, it applies to the parentheses as a whole.

Examples

Let’s see how parentheses work in examples.

Example: gogogo

Without parentheses, the pattern `go+` means `g` character, followed by `o` repeated one or more times. For instance, `goooo` or `gooooooooo`.

Parentheses group characters together, so `(go)+` means `go`, `gogo`, `gogogo` and so on.

```
alert( 'Gogogo now!'.match(/(go)+/i) ); // "Gogogo"
```

Example: domain

Let’s make something more complex – a regular expression to search for a website domain.

For example:

```
mail.com
users.mail.com
smith.users.mail.com
```

As we can see, a domain consists of repeated words, a dot after each one except the last one.

In regular expressions that’s `(\w+\.)+\w+`:

```
let regexp = /(\w+\.)+\w+/g;

alert( "site.com my.site.com".match(regexp) ); // site.com,my.site.com
```

The search works, but the pattern can't match a domain with a hyphen, e.g. `my-site.com`, because the hyphen does not belong to class `\w`.

We can fix it by replacing `\w` with `[\w-]` in every word except the last one: `([\w-]+\.)+\w+`.

Example: email

The previous example can be extended. We can create a regular expression for emails based on it.

The email format is: `name@domain`. Any word can be the name, hyphens and dots are allowed. In regular expressions that's `[-.\w]+`.

The pattern:

```
let regexp = /[-.\w]+@([\w-]+\.)+[\w-]+/g;

alert("my@mail.com @ his@site.com.uk".match(regexp)); // my@mail.com, his@sit
```

That regexp is not perfect, but mostly works and helps to fix accidental mistypes. The only truly reliable check for an email can only be done by sending a letter.

Parentheses contents in the match

Parentheses are numbered from left to right. The search engine memorizes the content matched by each of them and allows to get it in the result.

The method `str.match(regexp)`, if `regexp` has no flag `g`, looks for the first match and returns it as an array:

1. At index `0`: the full match.
2. At index `1`: the contents of the first parentheses.
3. At index `2`: the contents of the second parentheses.
4. …and so on…

For instance, we'd like to find HTML tags `<.*?>`, and process them. It would be convenient to have tag content (what's inside the angles), in a separate variable.

Let's wrap the inner content into parentheses, like this: `<(.*?)>`.

Now we'll get both the tag as a whole `<h1>` and its contents `h1` in the resulting array:

```
let str = '<h1>Hello, world!</h1>';

let tag = str.match(/<(.*?)>/);

alert( tag[0] ); // <h1>
alert( tag[1] ); // h1
```

Nested groups

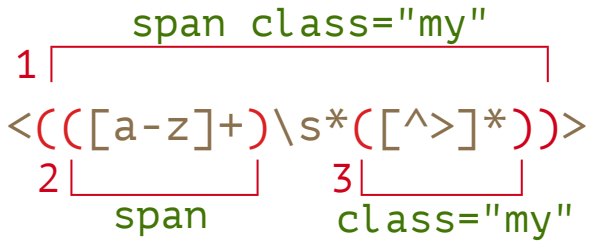
Parentheses can be nested. In this case the numbering also goes from left to right.

For instance, when searching a tag in `<span class="my">` we may be interested in:

1. The tag content as a whole: `span class="my"`.
2. The tag name: `span`.
3. The tag attributes: `class="my"`.

Let's add parentheses for them: `<(([a-z]+)\s*([^>]*))>`.

Here's how they are numbered (left to right, by the opening paren):



In action:

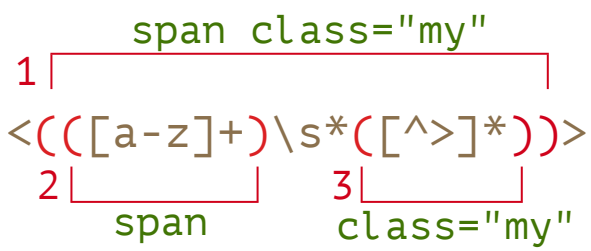
```
1  let str = '<span class="my">';
2
3  let regexp = /<(([a-z]+)\s*([^>]*))>/;
4
5  let result = str.match(regexp);
6  alert(result[0]); // <span class="my">
7  alert(result[1]); // span class="my"
8  alert(result[2]); // span
9  alert(result[3]); // class="my"
```

The zero index of `result` always holds the full match.

Then groups, numbered from left to right by an opening paren. The first group is returned as `result[1]`. Here it encloses the whole tag content.

Then in `result[2]` goes the group from the second opening paren `([a-z]+)` – tag name, then in `result[3]` the tag: `([^>]*)`.

The contents of every group in the string:



Optional groups

Even if a group is optional and doesn't exist in the match (e.g. has the quantifier `(...)?`), the corresponding `result` array item is present and equals `undefined`.

For instance, let's consider the regexp `a(z)?(c)?`. It looks for `"a"` optionally followed by `"z"` optionally followed by `"c"`.

If we run it on the string with a single letter `a`, then the result is:

```
let match = 'a'.match(/a(z)?(c)?/);

alert( match.length ); // 3
alert( match[0] ); // a (whole match)
alert( match[1] ); // undefined
alert( match[2] ); // undefined
```

The array has the length of `3`, but all groups are empty.

And here's a more complex match for the string `ac`:

```
let match = 'ac'.match(/a(z)?(c)?/)

alert( match.length ); // 3
alert( match[0] ); // ac (whole match)
alert( match[1] ); // undefined, because there's nothing for (z)?
alert( match[2] ); // c
```

The array length is permanent: `3`. But there's nothing for the group `(z)?`, so the result is `["ac", undefined, "c"]`.

Searching for all matches with groups: matchAll

> **`matchAll` is a new method, polyfill may be needed**
>
> The method `matchAll` is not supported in old browsers.
>
> A polyfill may be required, such as https://github.com/ljharb/String.prototype.matchAll.

When we search for all matches (flag `g`), the `match` method does not return contents for groups.

For example, let's find all tags in a string:

```
let str = '<h1> <h2>';

let tags = str.match(/<(.*?)>/g);

alert( tags ); // <h1>,<h2>
```

The result is an array of matches, but without details about each of them. But in practice we usually need contents of capturing groups in the result.

To get them, we should search using the method `str.matchAll(regexp)`.

It was added to JavaScript language long after `match`, as its "new and improved version".

Just like `match`, it looks for matches, but there are 3 differences:

1. It returns not an array, but an iterable object.
2. When the flag `g` is present, it returns every match as an array with groups.
3. If there are no matches, it returns not `null`, but an empty iterable object.

For instance:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

// results - is not an array, but an iterable object
alert(results); // [object RegExp String Iterator]

alert(results[0]); // undefined (*)

results = Array.from(results); // let's turn it into array

alert(results[0]); // <h1>,h1 (1st tag)
alert(results[1]); // <h2>,h2 (2nd tag)
```

As we can see, the first difference is very important, as demonstrated in the line `(*)`. We can't get the match as `results[0]`, because that object isn't pseudoarray. We can turn it into a real `Array` using `Array.from`. There are more details about pseudoarrays and iterables in the article Iterables.

There's no need in `Array.from` if we're looping over results:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

for(let result of results) {
  alert(result);
  // первый вывод: <h1>,h1
  // второй: <h2>,h2
}
```

…Or using destructuring:

```
let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

Every match, returned by `matchAll`, has the same format as returned by `match` without flag `g`: it's an array with additional properties `index` (match index in the string) and `input` (source string):

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

let [tag1, tag2] = results;

alert( tag1[0] ); // <h1>
alert( tag1[1] ); // h1
alert( tag1.index ); // 0
alert( tag1.input ); // <h1> <h2>
```

Why is a result of `matchAll` an iterable object, not an array?

Why is the method designed like that? The reason is simple – for the optimization.

The call to `matchAll` does not perform the search. Instead, it returns an iterable object, without the results initially. The search is performed each time we iterate over it, e.g. in the loop.

So, there will be found as many results as needed, not more.

E.g. there are potentially 100 matches in the text, but in a `for..of` loop we found 5 of them, then decided it's enough and make a `break`. Then the engine won't spend time finding other 95 mathces.

Named groups

Remembering groups by their numbers is hard. For simple patterns it's doable, but for more complex ones counting parentheses is inconvenient. We have a much better option: give names to parentheses.

That's done by putting `?<name>` immediately after the opening paren.

For example, let's look for a date in the format "year-month-day":

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "2019-04-30";

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

alert(groups.year); // 2019
alert(groups.month); // 04
alert(groups.day); // 30
```

As you can see, the groups reside in the `.groups` property of the match.

To look for all dates, we can add flag `g`.

We'll also need `matchAll` to obtain full matches, together with groups:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30 2020-01-01";

let results = str.matchAll(dateRegexp);

for(let result of results) {
  let {year, month, day} = result.groups;

  alert(`${day}.${month}.${year}`);
  // first alert: 30.10.2019
  // second: 01.01.2020
}
```

Capturing groups in replacement

Method `str.replace(regexp, replacement)` that replaces all matches with `regexp` in `str` allows to use parentheses contents in the `replacement` string. That's done using `$n`, where `n` is the group number.

For example,

```
let str = "John Bull";
let regexp = /(\w+) (\w+)/;

alert( str.replace(regexp, '$2, $1') ); // Bull, John
```

For named parentheses the reference will be `$<name>`.

For example, let's reformat dates from "year-month-day" to "day.month.year":

```
let regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30, 2020-01-01";

alert( str.replace(regexp, '$<day>.$<month>.$<year>') );
// 30.10.2019, 01.01.2020
```

Non-capturing groups with ?:

Sometimes we need parentheses to correctly apply a quantifier, but we don't want their contents in results.

A group may be excluded by adding `?:` in the beginning.

For instance, if we want to find `(go)+`, but don't want the parentheses contents (`go`) as a separate array item, we can write: `(?:go)+`.

In the example below we only get the name `John` as a separate member of the match:

```
let str = "Gogogo John!";

// ?: exludes 'go' from capturing
let regexp = /(?:go)+ (\w+)/i;

let result = str.match(regexp);

alert( result[0] ); // Gogogo John (full match)
alert( result[1] ); // John
alert( result.length ); // 2 (no more items in the array)
```

Summary

Parentheses group together a part of the regular expression, so that the quantifier applies to it as a whole.

Parentheses groups are numbered left-to-right, and can optionally be named with `(?<name>...)`.

The content, matched by a group, can be obtained in the results:

- The method `str.match` returns capturing groups only without flag `g`.

- The method `str.matchAll` always returns capturing groups.

If the parentheses have no name, then their contents is available in the match array by its number. Named parentheses are also available in the property `groups`.

We can also use parentheses contents in the replacement string in `str.replace`: by the number `$n` or the name `$<name>`.

A group may be excluded from numbering by adding `?:` in its start. That's used when we need to apply a quantifier to the whole group, but don't want it as a separate item in the results array. We also can't reference such parentheses in the replacement string.

Tasks

Check MAC-address

MAC-address of a network interface consists of 6 two-digit hex numbers separated by a colon.

For instance: `'01:32:54:67:89:AB'`.

Write a regexp that checks whether a string is MAC-address.

Usage:

```
let regexp = /your regexp/;

alert( regexp.test('01:32:54:67:89:AB') ); // true

alert( regexp.test('0132546789AB') ); // false (no colons)

alert( regexp.test('01:32:54:67:89') ); // false (5 numbers, must be 6)

alert( regexp.test('01:32:54:67:89:ZZ') ) // false (ZZ ad the end)
```

solution

Find color in the format #abc or #abcdef

Write a RegExp that matches colors in the format `#abc` or `#abcdef`. That is: `#` followed by 3 or 6 hexadecimal digits.

Usage example:

```
let regexp = /your regexp/g;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(regexp) ); // #3f3 #AA00ef
```

P.S. This should be exactly 3 or 6 hex digits. Values with 4 digits, such as `#abcd`, should not match.

solution

Find all numbers

Write a regexp that looks for all decimal numbers including integer ones, with the floating point and negative ones.

An example of use:

```
let regexp = /your regexp/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(regexp) ); // -1.5, 0, 2, -123.4
```

solution

Parse an expression

An arithmetical expression consists of 2 numbers and an operator between them, for instance:

- `1 + 2`
- `1.2 * 3.4`
- `-3 / -6`
- `-2 - 2`

The operator is one of: `"+"`, `"-"`, `"*"` or `"/"`.

There may be extra spaces at the beginning, at the end or between the parts.

Create a function `parse(expr)` that takes an expression and returns an array of 3 items:

1. The first number.
2. The operator.
3. The second number.

For example:

```
let [a, op, b] = parse("1.2 * 3.4");

alert(a); // 1.2
alert(op); // *
alert(b); // 3.4
```

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)