Regular expressions

Regular expressions is a powerful way of doing search and replace in strings.

Patterns and flags

Regular expressions are patterns that provide a powerful way to search and replace in text.

In JavaScript, they are available via the RegExp object, as well as being integrated in methods of strings.

Regular Expressions

A regular expression (also “regexp”, or just “reg”) consists of a *pattern* and optional *flags*.

There are two syntaxes that can be used to create a regular expression object.

The “long” syntax:

```
regexp = new RegExp("pattern", "flags");
```

And the “short” one, using slashes `"/"`:

```
regexp = /pattern/; // no flags
regexp = /pattern/gmi; // with flags g,m and i (to be covered soon)
```

Slashes `/.../` tell JavaScript that we are creating a regular expression. They play the same role as quotes for strings.

In both cases `regexp` becomes an instance of the built-in `RegExp` class.

The main difference between these two syntaxes is that pattern using slashes `/.../` does not allow for expressions to be inserted (like string template literals with `${...}`). They are fully static.

Slashes are used when we know the regular expression at the code writing time – and that’s the most common situation. While `new RegExp`, is more often used when we need to create a regexp “on the fly” from a dynamically generated string. For instance:

```
let tag = prompt("What tag do you want to find?", "h2");
```

Searching: str.match

As mentioned previously, regular expressions are integrated with string methods.

The method `str.match(regexp)` finds all matches of `regexp` in the string `str`.

It has 3 working modes:

1. If the regular expression has flag `g`, it returns an array of all matches:

   ```
   let str = "We will, we will rock you";

   alert( str.match(/we/gi) ); // We,we (an array of 2 substrings that match)
   ```

   Please note that both `We` and `we` are found, because flag `i` makes the regular expression case-insensitive.

2. If there's no such flag it returns only the first match in the form of an array, with the full match at index `0` and some additional details in properties:

   ```
   let str = "We will, we will rock you";

   let result = str.match(/we/i); // without flag g

   alert( result[0] );     // We (1st match)
   alert( result.length ); // 1

   // Details:
   alert( result.index );  // 0 (position of the match)
   alert( result.input );  // We will, we will rock you (source string)
   ```

   The array may have other indexes, besides `0` if a part of the regular expression is enclosed in parentheses. We'll cover that in the chapter Capturing groups.

3. And, finally, if there are no matches, `null` is returned (doesn't matter if there's flag `g` or not).

   This a very important nuance. If there are no matches, we don't receive an empty array, but instead receive `null`. Forgetting about that may lead to errors, e.g.:

   ```
   let matches = "JavaScript".match(/HTML/); // = null

   if (!matches.length) { // Error: Cannot read property 'length' of null
     alert("Error in the line above");
   }
   ```

If we'd like the result to always be an array, we can write it this way:

```
let matches = "JavaScript".match(/HTML/) || [];

if (!matches.length) {
  alert("No matches"); // now it works
}
```

Replacing: str.replace

The method `str.replace(regexp, replacement)` replaces matches found using `regexp` in string `str` with `replacement` (all matches if there's flag `g`, otherwise, only the first one).

For instance:

```
// no flag g
alert( "We will, we will".replace(/we/i, "I") ); // I will, we will

// with flag g
alert( "We will, we will".replace(/we/ig, "I") ); // I will, I will
```

The second argument is the `replacement` string. We can use special character combinations in it to insert fragments of the match:

Symbols	Action in the replacement string
`$&`	inserts the whole match
`$``	inserts a part of the string before the match
`$'`	inserts a part of the string after the match
`$n`	if `n` is a 1-2 digit number, then it inserts the contents of n-th parentheses, more about it in the chapter Capturing groups
`$<name>`	inserts the contents of the parentheses with the given `name`, more about it in the chapter Capturing groups
`$$`	inserts character `$`

An example with `$&`:

```
alert( "I love HTML".replace(/HTML/, "$& and JavaScript") ); // I love HTML and Java
```

Testing: regexp.test

The method `regexp.test(str)` looks for at least one match, if found, returns `true`, otherwise `false`.

```
let str = "I love JavaScript";
let regexp = /LOVE/i;

alert( regexp.test(str) ); // true
```

Later in this chapter we'll study more regular expressions, walk through more examples, and also meet other methods.

Full information about the methods is given in the article Methods of RegExp and String.

Summary

- A regular expression consists of a pattern and optional flags: `g`, `i`, `m`, `u`, `s`, `y`.
- Without flags and special symbols (that we'll study later), the search by a regexp is the same as a substring search.
- The method `str.match(regexp)` looks for matches: all of them if there's `g` flag, otherwise, only the first one.
- The method `str.replace(regexp, replacement)` replaces matches found using `regexp` with `replacement`: all of them if there's `g` flag, otherwise only the first one.
- The method `regexp.test(str)` returns `true` if there's at least one match, otherwise, it returns `false`.

Character classes

Consider a practical task – we have a phone number like `"+7(903)-123-45-67"`, and we need to turn it into pure numbers: `79035419441`.

To do so, we can find and remove anything that's not a number. Character classes can help with that.

A *character class* is a special notation that matches any symbol from a certain set.

For the start, let's explore the "digit" class. It's written as `\d` and corresponds to "any single digit".

For instance, the let's find the first digit in the phone number:

```
let str = "+7(903)-123-45-67";

let regexp = /\d/;

alert( str.match(regexp) ); // 7
```

Without the flag `g`, the regular expression only looks for the first match, that is the first digit `\d`.

Let's add the `g` flag to find all digits:

```
let str = "+7(903)-123-45-67";

let regexp = /\d/g;

alert( str.match(regexp) ); // array of matches: 7,9,0,3,1,2,3,4,5,6,7

// let's make the digits-only phone number of them:
alert( str.match(regexp).join('') ); // 79035419441
```

That was a character class for digits. There are other character classes as well.

Most used are:

`\d` ("d" is from "digit")

A digit: a character from `0` to `9`.

`\s` ("s" is from "space")

A space symbol: includes spaces, tabs `\t`, newlines `\n` and few other rare characters, such as `\v`, `\f` and `\r`.

`\w` ("w" is from "word")

A "wordly" character: either a letter of Latin alphabet or a digit or an underscore `_`. Non-Latin letters (like cyrillic or hindi) do not belong to `\w`.

For instance, `\d\s\w` means a "digit" followed by a "space character" followed by a "wordly character", such as `1 a`.

A regexp may contain both regular symbols and character classes.

For instance, `CSS\d` matches a string `CSS` with a digit after it:

```
let str = "Is there CSS4?";
let regexp = /CSS\d/
```

```
alert( str.match(regexp) ); // CSS4
```

Also we can use many character classes:

```
alert( "I love HTML5!".match(/\s\w\w\w\w\d/) ); // ' HTML5'
```

The match (each regexp character class has the corresponding result character):

Inverse classes

For every character class there exists an “inverse class”, denoted with the same letter, but uppercased.

The “inverse” means that it matches all other characters, for instance:

`\D`
Non-digit: any character except `\d`, for instance a letter.

`\S`
Non-space: any character except `\s`, for instance a letter.

`\W`
Non-wordly character: anything but `\w`, e.g a non-latin letter or a space.

In the beginning of the chapter we saw how to make a number-only phone number from a string like `+7(903)-123-45-67`: find all digits and join them.

```
let str = "+7(903)-123-45-67";

alert( str.match(/\d/g).join('') ); // 79031234567
```

An alternative, shorter way is to find non-digits `\D` and remove them from the string:

```
let str = "+7(903)-123-45-67";

alert( str.replace(/\D/g, "") ); // 79031234567
```

A dot is “any character”

A dot `.` is a special character class that matches “any character except a newline”.

For instance:

```
alert( "Z".match(/./) ); // Z
```

Or in the middle of a regexp:

```
let regexp = /CS.4/;

alert( "CSS4".match(regexp) ); // CSS4
alert( "CS-4".match(regexp) ); // CS-4
alert( "CS 4".match(regexp) ); // CS 4 (space is also a character)
```

Please note that a dot means “any character”, but not the “absense of a character”. There must be a character to match it:

```
alert( "CS4".match(/CS.4/) ); // null, no match because there's no character for th
```

Dot as literally any character with “s” flag

By default, a dot doesn’t match the newline character `\n`.

For instance, the regexp `A.B` matches `A`, and then `B` with any character between them, except a newline `\n`:

```
alert( "A\nB".match(/A.B/) ); // null (no match)
```

There are many situations when we’d like a dot to mean literally “any character”, newline included.

That’s what flag `s` does. If a regexp has it, then a dot `.` matches literally any character:

```
alert( "A\nB".match(/A.B/s) ); // A\nB (match!)
```

Not supported in Firefox, IE, Edge

Check https://caniuse.com/#search=dotall for the most recent state of support. At the time of writing it doesn't include Firefox, IE, Edge.

Luckily, there's an alternative, that works everywhere. We can use a regexp like `[\s\S]` to match "any character".

```
alert( "A\nB".match(/A[\s\S]B/) ); // A\nB (match!)
```

The pattern `[\s\S]` literally says: "a space character OR not a space character". In other words, "anything". We could use another pair of complementary classes, such as `[\d\D]`, that doesn't matter. Or even the `[^]` – as it means match any character except nothing.

Also we can use this trick if we want both kind of "dots" in the same pattern: the actual dot `.` behaving the regular way ("not including a newline"), and also a way to match "any character" with `[\s\S]` or alike.

Pay attention to spaces

Usually we pay little attention to spaces. For us strings `1-5` and `1 - 5` are nearly identical.

But if a regexp doesn't take spaces into account, it may fail to work.

Let's try to find digits separated by a hyphen:

```
alert( "1 - 5".match(/\d-\d/) ); // null, no match!
```

Let's fix it adding spaces into the regexp `\d - \d`:

```
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, now it works
// or we can use \s class:
alert( "1 - 5".match(/\d\s-\s\d/) ); // 1 - 5, also works
```

A space is a character. Equal in importance with any other character.

We can't add or remove spaces from a regular expression and expect to work the same.

In other words, in a regular expression all characters matter, spaces too.

Summary

There exist following character classes:

- `\d` – digits.
- `\D` – non-digits.
- `\s` – space symbols, tabs, newlines.
- `\S` – all but `\s`.
- `\w` – Latin letters, digits, underscore `'_'`.
- `\W` – all but `\w`.
- `.` – any character if with the regexp `'s'` flag, otherwise any except a newline `\n`.

…But that's not all!

Unicode encoding, used by JavaScript for strings, provides many properties for characters, like: which language the letter belongs to (if it's a letter) it is it a punctuation sign, etc.

We can search by these properties as well. That requires flag `u`, covered in the next article.

Unicode: flag "u" and class \p{...}

JavaScript uses Unicode encoding for strings. Most characters are encoding with 2 bytes, but that allows to represent at most 65536 characters.

That range is not big enough to encode all possible characters, that's why some rare characters are encoded with 4 bytes, for instance like `𝒳` (mathematical X) or `😄` (a smile), some hieroglyphs and so on.

Here are the unicode values of some characters:

Character	Unicode	Bytes count in unicode
a	`0x0061`	2
≈	`0x2248`	2
𝒳	`0x1d4b3`	4
𝒴	`0x1d4b4`	4
😄	`0x1f604`	4

So characters like `a` and `≈` occupy 2 bytes, while codes for `𝒳`, `𝒴` and `😄` are longer, they have 4 bytes.

Long time ago, when JavaScript language was created, Unicode encoding was simpler: there were no 4-byte characters. So, some language features still handle them incorrectly.

For instance, `length` thinks that here are two characters:

```
alert('😄'.length); // 2
alert('𝒳'.length); // 2
```

…But we can see that there's only one, right? The point is that `length` treats 4 bytes as two 2-byte characters. That's incorrect, because they must be considered only together (so-called "surrogate pair", you can read about them in the article Strings).

By default, regular expressions also treat 4-byte "long characters" as a pair of 2-byte ones. And, as it happens with strings, that may lead to odd results. We'll see that a bit later, in the article Sets and ranges [...].

Unlike strings, regular expressions have flag `u` that fixes such problems. With such flag, a regexp handles 4-byte characters correctly. And also Unicode property search becomes available, we'll get to it next.

Unicode properties \p{...}

> **Not supported in Firefox and Edge**
>
> Despite being a part of the standard since 2018, unicode properties are not supported in Firefox (bug) and Edge (bug).
>
> There's XRegExp library that provides "extended" regular expressions with cross-browser support for unicode properties.

Every character in Unicode has a lot of properties. They describe what "category" the character belongs to, contain miscellaneous information about it.

For instance, if a character has `Letter` property, it means that the character belongs to an alphabet (of any language). And `Number` property means that it's a digit: maybe Arabic or Chinese, and so on.

We can search for characters with a property, written as `\p{…}`. To use `\p{…}`, a regular expression must have flag `u`.

For instance, `\p{Letter}` denotes a letter in any of language. We can also use `\p{L}`, as `L` is an alias of `Letter`. There are shorter aliases for almost every property.

In the example below three kinds of letters will be found: English, Georgean and Korean.

```
let str = "A ბ 한";

alert( str.match(/\p{L}/gu) ); // A,ბ,한
alert( str.match(/\p{L}/g) ); // null (no matches, as there's no flag "u")
```

Here's the main character categories and their subcategories:

- Letter `L`:
  - lowercase `Ll`
  - modifier `Lm`,
  - titlecase `Lt`,
  - uppercase `Lu`,
  - other `Lo`.
- Number `N`:
  - decimal digit `Nd`,
  - letter number `Nl`,
  - other `No`.
- Punctuation `P`:
  - connector `Pc`,
  - dash `Pd`,
  - initial quote `Pi`,
  - final quote `Pf`,
  - open `Ps`,
  - close `Pe`,
  - other `Po`.
- Mark `M` (accents etc):
  - spacing combining `Mc`,
  - enclosing `Me`,
  - non-spacing `Mn`.
- Symbol `S`:
  - currency `Sc`,
  - modifier `Sk`,
  - math `Sm`,
  - other `So`.

- Separator `Z`:
  - line `Zl`,
  - paragraph `Zp`,
  - space `Zs`.
- Other `C`:
  - control `Cc`,
  - format `Cf`,
  - not assigned `Cn`, – private use `Co`,
  - surrogate `Cs`.

So, e.g. if we need letters in lower case, we can write `\p{Ll}`, punctuation signs: `\p{P}` and so on.

There are also other derived categories, like:

- `Alphabetic` (`Alpha`), includes Letters `L`, plus letter numbers `Nl` (e.g. Ⅻ – a character for the roman number 12), plus some other symbols `Other_Alphabetic` (`OAlpha`).
- `Hex_Digit` includes hexadecimal digits: `0-9`, `a-f`.
- …And so on.

Unicode supports many different properties, their full list would require a lot of space, so here are the references:

- List all properties by a character: https://unicode.org/cldr/utility/character.jsp .
- List all characters by a property: https://unicode.org/cldr/utility/list-unicodeset.jsp .
- Short aliases for properties: https://www.unicode.org/Public/UCD/latest/ucd/PropertyValueAliases.txt .
- A full base of Unicode characters in text format, with all properties, is here: https://www.unicode.org/Public/UCD/latest/ucd/ .

Example: hexadecimal numbers

For instance, let's look for hexadecimal numbers, written as `xFF`, where `F` is a hex digit (0…1 or A…F).

A hex digit can be denoted as `\p{Hex_Digit}`:

```
let regexp = /x\p{Hex_Digit}\p{Hex_Digit}/u;

alert("number: xAF".match(regexp)); // xAF
```

Example: Chinese hieroglyphs

Let's look for Chinese hieroglyphs.

There's a unicode property `Script` (a writing system), that may have a value: `Cyrillic`, `Greek`, `Arabic`, `Han` (Chinese) and so on, here's the full list.

To look for characters in a given writing system we should use `Script=<value>`, e.g. for Cyrillic letters: `\p{sc=Cyrillic}`, for Chinese hieroglyphs: `\p{sc=Han}`, and so on:

```
let regexp = /\p{sc=Han}/gu; // returns Chinese hieroglyphs

let str = `Hello Привет 你好 123_456`;

alert( str.match(regexp) ); // 你,好
```

Example: currency

Characters that denote a currency, such as `$`, `€`, `¥`, have unicode property `\p{Currency_Symbol}`, the short alias: `\p{Sc}`.

Let's use it to look for prices in the format "currency, followed by a digit":

```
let regexp = /\p{Sc}\d/gu;

let  str = `Prices: $2, €1, ¥9`;

alert( str.match(regexp) ); // $2,€1,¥9
```

Later, in the article Quantifiers +, *, ? and {n} we'll see how to look for numbers that contain many digits.

Summary

Flag `u` enables the support of Unicode in regular expressions.

That means two things:

1. Characters of 4 bytes are handled correctly: as a single character, not two 2-byte characters.
2. Unicode properties can be used in the search: `\p{…}`.

With Unicode properties we can look for words in given languages, special characters (quotes, currencies) and so on.

Anchors: string start ^ and end $

The caret `^` and dollar `$` characters have special meaning in a regexp. They are called “anchors”.

The caret `^` matches at the beginning of the text, and the dollar `$` – at the end.

For instance, let’s test if the text starts with `Mary`:

```
let str1 = "Mary had a little lamb";
alert( /^Mary/.test(str1) ); // true
```

The pattern `^Mary` means: “string start and then Mary”.

Similar to this, we can test if the string ends with `snow` using `snow$`:

```
let str1 = "it's fleece was white as snow";
alert( /snow$/.test(str1) ); // true
```

In these particular cases we could use string methods `startsWith/endsWith` instead. Regular expressions should be used for more complex tests.

Testing for a full match

Both anchors together `^...$` are often used to test whether or not a string fully matches the pattern. For instance, to check if the user input is in the right format.

Let’s check whether or not a string is a time in `12:34` format. That is: two digits, then a colon, and then another two digits.

In regular expressions language that’s `\d\d:\d\d`:

```
let goodInput = "12:34";
let badInput = "12:345";

let regexp = /^\d\d:\d\d$/;
alert( regexp.test(goodInput) ); // true
alert( regexp.test(badInput) ); // false
```

Here the match for `\d\d:\d\d` must start exactly after the beginning of the text `^`, and the end `$` must immediately follow.

The whole string must be exactly in this format. If there’s any deviation or an extra character, the result is `false`.

Anchors behave differently if flag `m` is present. We’ll see that in the next article.

Anchors have “zero width”

Anchors `^` and `$` are tests. They have zero width.

In other words, they do not match a character, but rather force the regexp engine to check the condition (text start/end).

Multiline mode of anchors ^ $, flag "m"

The multiline mode is enabled by the flag `m`.

It only affects the behavior of `^` and `$`.

In the multiline mode they match not only at the beginning and the end of the string, but also at start/end of line.

Searching at line start ^

In the example below the text has multiple lines. The pattern `/^\d/gm` takes a digit from the beginning of each line:

```
let str = `1st place: Winnie
2nd place: Piglet
3rd place: Eeyore`;

alert( str.match(/^\d/gm) ); // 1, 2, 3
```

Without the flag `m` only the first digit is matched:

```
let str = `1st place: Winnie
2nd place: Piglet
3rd place: Eeyore`;

alert( str.match(/^\d/g) ); // 1
```

That’s because by default a caret `^` only matches at the beginning of the text, and in the multiline mode – at the start of any line.

Please note:

“Start of a line” formally means “immediately after a line break”: the test `^` in multiline mode matches at all positions preceeded by a newline character `\n`.

And at the text start.

Searching at line end $

The dollar sign `$` behaves similarly.

The regular expression `\d$` finds the last digit in every line

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

alert( str.match(/\d$/gm) ); // 1,2,3
```

Without the flag `m`, the dollar `$` would only match the end of the whole text, so only the very last digit would be found.

> **Please note:**
>
> “End of a line” formally means “immediately before a line break”: the test `$` in multiline mode matches at all positions succeeded by a newline character `\n`.
>
> And at the text end.

Searching for \n instead of ^ $

To find a newline, we can use not only anchors `^` and `$`, but also the newline character `\n`.

What’s the difference? Let’s see an example.

Here we search for `\d\n` instead of `\d$`:

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

alert( str.match(/\d\n/gm) ); // 1\n,2\n
```

As we can see, there are 2 matches instead of 3.

That’s because there’s no newline after `3` (there’s text end though, so it matches `$`).

Another difference: now every match includes a newline character `\n`. Unlike the anchors `^` `$`, that only test the condition (start/end of a line), `\n` is a character, so it becomes a part of the result.

So, a `\n` in the pattern is used when we need newline characters in the result, while anchors are used to find something at the beginning/end of a line.

Word boundary: \b

A word boundary `\b` is a test, just like `^` and `$`.

When the regexp engine (program module that implements searching for regexps) comes across `\b`, it checks that the position in the string is a word boundary.

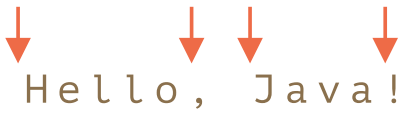
There are three different positions that qualify as word boundaries:

- At string start, if the first string character is a word character `\w`.
- Between two characters in the string, where one is a word character `\w` and the other is not.
- At string end, if the last string character is a word character `\w`.

For instance, regexp `\bJava\b` will be found in `Hello, Java!`, where `Java` is a standalone word, but not in `Hello, JavaScript!`.

```
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, JavaScript!".match(/\bJava\b/) ); // null
```

In the string `Hello, Java!` following positions correspond to `\b`:



So, it matches the pattern `\bHello\b`, because:

1. At the beginning of the string matches the first test `\b`.
2. Then matches the word `Hello`.
3. Then the test `\b` matches again, as we're between `o` and a space.

The pattern `\bJava\b` would also match. But not `\bHell\b` (because there's no word boundary after `l`) and not `Java!\b` (because the exclamation sign is not a wordly character `\w`, so there's no word boundary after it).

```
alert( "Hello, Java!".match(/\bHello\b/) ); // Hello
alert( "Hello, Java!".match(/\bJava\b/) );  // Java
alert( "Hello, Java!".match(/\bHell\b/) );  // null (no match)
alert( "Hello, Java!".match(/\bJava!\b/) ); // null (no match)
```

We can use `\b` not only with words, but with digits as well.

For example, the pattern `\b\d\d\b` looks for standalone 2-digit numbers. In other words, it looks for 2-digit numbers that are surrounded by characters different from `\w`, such as spaces or punctuation (or text start/end).

```
alert( "1 23 456 78".match(/\b\d\d\b/g) ); // 23,78
alert( "12,34,56".match(/\b\d\d\b/g) ); // 12,34,56
```

Word boundary `\b` doesn't work for non-latin alphabets

The word boundary test `\b` checks that there should be `\w` on the one side from the position and "not `\w`" – on the other side.

But `\w` means a latin letter `a-z` (or a digit or an underscore), so the test doesn't work for other characters, e.g. cyrillic letters or hieroglyphs.

Escaping, special characters

As we've seen, a backslash `\` is used to denote character classes, e.g. `\d`. So it's a special character in regexps (just like in regular strings).

There are other special characters as well, that have special meaning in a regexp. They are used to do more powerful searches. Here's a full list of them: `[ \ ^ $ . | ? * + ( )`.

Don't try to remember the list – soon we'll deal with each of them separately and you'll know them by heart automatically.

Escaping

Let's say we want to find literally a dot. Not "any character", but just a dot.

To use a special character as a regular one, prepend it with a backslash: `\.`.

That's also called "escaping a character".

For example:

```
alert( "Chapter 5.1".match(/\d\.\d/) ); // 5.1 (match!)
alert( "Chapter 511".match(/\d\.\d/) ); // null (looking for a real dot \.)
```

Parentheses are also special characters, so if we want them, we should use `\(`. The example below looks for a string `"g()"`:

```
alert( "function g()".match(/g\(\)/) ); // "g()"
```

If we're looking for a backslash `\`, it's a special character in both regular strings and regexps, so we should double it.

```
alert( "1\\2".match(/\\/) ); // '\'
```

A slash

A slash symbol `'/'` is not a special character, but in JavaScript it is used to open and close the regexp: `/...pattern.../`, so we should escape it too.

Here's what a search for a slash `'/'` looks like:

```
alert( "/".match(/\//) ); // '/'
```

On the other hand, if we're not using `/.../`, but create a regexp using `new RegExp`, then we don't need to escape it:

```
alert( "/".match(new RegExp("/")) ); // finds /
```

new RegExp

If we are creating a regular expression with `new RegExp`, then we don't have to escape `/`, but need to do some other escaping.

For instance, consider this:

```
let regexp = new RegExp("\d\.\d");

alert( "Chapter 5.1".match(regexp) ); // null
```

The similar search in one of previous examples worked with `/\d\.\d/`, but `new RegExp("\d\.\d")` doesn't work, why?

The reason is that backslashes are "consumed" by a string. As we may recall, regular strings have their own special characters, such as `\n`, and a backslash is used for escaping.

Here's how "\d.\d" is preceived:

```
alert("\d\.\d"); // d.d
```

String quotes "consume" backslashes and interpret them on their own, for instance:

- `\n` – becomes a newline character,
- `\u1234` – becomes the Unicode character with such code,
- …And when there's no special meaning: like `\d` or `\z`, then the backslash is simply removed.

So `new RegExp` gets a string without backslashes. That's why the search doesn't work!

To fix it, we need to double backslashes, because string quotes turn `\\` into `\`:

```
let regStr = "\\d\\.\\d";
alert(regStr); // \d\.\d (correct now)

let regexp = new RegExp(regStr);

alert( "Chapter 5.1".match(regexp) ); // 5.1
```

Summary

- To search for special characters `[ \ ^ $ . | ? * + ( )` literally, we need to prepend them with a backslash `\` ("escape them").
- We also need to escape `/` if we're inside `/.../` (but not inside `new RegExp`).
- When passing a string `new RegExp`, we need to double backslashes `\\`, cause string quotes consume one of them.

Sets and ranges [...]

Several characters or character classes inside square brackets `[…]` mean to "search for any character among given".

Sets

For instance, `[eao]` means any of the 3 characters: `'a'`, `'e'`, or `'o'`.

That's called a *set*. Sets can be used in a regexp along with regular characters:

```
// find [t or m], and then "op"
alert( "Mop top".match(/[tm]op/gi) ); // "Mop", "top"
```

Please note that although there are multiple characters in the set, they correspond to exactly one character in the match.

So the example below gives no matches:

```
// find "V", then [o or i], then "la"
alert( "Voila".match(/V[oi]la/) ); // null, no matches
```

The pattern searches for:

- `V`,
- then *one* of the letters `[oi]`,
- then `la`.

So there would be a match for `Vola` or `Vila`.

Ranges

Square brackets may also contain *character ranges*.

For instance, `[a-z]` is a character in range from `a` to `z`, and `[0-5]` is a digit from `0` to `5`.

In the example below we're searching for `"x"` followed by two digits or letters from `A` to `F`:

```
alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF
```

Here `[0-9A-F]` has two ranges: it searches for a character that is either a digit from `0` to `9` or a letter from `A` to `F`.

If we'd like to look for lowercase letters as well, we can add the range `a-f`: `[0-9A-Fa-f]`. Or add the flag `i`.

We can also use character classes inside `[…]`.

For instance, if we'd like to look for a wordly character `\w` or a hyphen `-`, then the set is `[\w-]`.

Combining multiple classes is also possible, e.g. `[\s\d]` means "a space character or a digit".

Character classes are shorthands for certain character sets

For instance:

- **\d** – is the same as `[0-9]`,
- **\w** – is the same as `[a-zA-Z0-9_]`,
- **\s** – is the same as `[\t\n\v\f\r ]`, plus few other rare unicode space characters.

Example: multi-language \w

As the character class `\w` is a shorthand for `[a-zA-Z0-9_]`, it can't find Chinese hieroglyphs, Cyrillic letters, etc.

We can write a more universal pattern, that looks for wordly characters in any language. That's easy with unicode properties: `[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]`.

Let's decipher it. Similar to `\w`, we're making a set of our own that includes characters with following unicode properties:

- `Alphabetic` (`Alpha`) – for letters,
- `Mark` (`M`) – for accents,
- `Decimal_Number` (`Nd`) – for digits,
- `Connector_Punctuation` (`Pc`) – for the underscore `'_'` and similar characters,
- `Join_Control` (`Join_C`) – two special codes `200c` and `200d`, used in ligatures, e.g. in Arabic.

An example of use:

```
let regexp = /[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]/gu;

let str = `Hi 你好 12`;

// finds all letters and digits:
alert( str.match(regexp) ); // H,i,你,好,1,2
```

Of course, we can edit this pattern: add unicode properties or remove them. Unicode properties are covered in more details in the article Unicode: flag "u" and class \p{...}.

Unicode properties aren't supported in Edge and Firefox

Unicode properties `p{…}` are not yet implemented in Edge and Firefox. If we really need them, we can use library XRegExp .

Or just use ranges of characters in a language that interests us, e.g. `[а-я]` for Cyrillic letters.

Excluding ranges

Besides normal ranges, there are "excluding" ranges that look like `[^…]`.

They are denoted by a caret character `^` at the start and match any character *except the given ones*.

For instance:

- `[^aeyo]` – any character except `'a'`, `'e'`, `'y'` or `'o'`.
- `[^0-9]` – any character except a digit, the same as `\D`.
- `[^\s]` – any non-space character, same as `\S`.

The example below looks for any characters except letters, digits and spaces:

```
alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // @ and .
```

Escaping in […]

Usually when we want to find exactly a special character, we need to escape it like `\.`. And if we need a backslash, then we use `\\`, and so on.

In square brackets we can use the vast majority of special characters without escaping:

- Symbols `. + ( )` never need escaping.
- A hyphen `-` is not escaped in the beginning or the end (where it does not define a range).
- A caret `^` is only escaped in the beginning (where it means exclusion).
- The closing square bracket `]` is always escaped (if we need to look for that symbol).

In other words, all special characters are allowed without escaping, except when they mean something for square brackets.

A dot `.` inside square brackets means just a dot. The pattern `[.,]` would look for one of characters: either a dot or a comma.

In the example below the regexp `[-().^+]` looks for one of the characters `-().^+`:

```
// No need to escape
let regexp = /[-().^+]/g;

alert( "1 + 2 - 3".match(regexp) ); // Matches +, -
```

…But if you decide to escape them "just in case", then there would be no harm:

```
// Escaped everything
let regexp = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(regexp) ); // also works: +, -
```

Ranges and flag "u"

If there are surrogate pairs in the set, flag `u` is required for them to work correctly.

For instance, let's look for `[𝒳𝒴]` in the string `𝒳`:

```
alert( '𝒳'.match(/[𝒳𝒴]/) ); // shows a strange character, like [?]
// (the search was performed incorrectly, half-character returned)
```

The result is incorrect, because by default regular expressions "don't know" about surrogate pairs.

The regular expression engine thinks that `[𝒳𝒴]` – are not two, but four characters:

1. left half of `𝒳` `(1)`,
2. right half of `𝒳` `(2)`,
3. left half of `𝒴` `(3)`,
4. right half of `𝒴` `(4)`.

We can see their codes like this:

```
for(let i=0; i<'𝒳𝒴'.length; i++) {
  alert('𝒳𝒴'.charCodeAt(i)); // 55349, 56499, 55349, 56500
};
```

So, the example above finds and shows the left half of `𝒳`.

If we add flag `u`, then the behavior will be correct:

```
alert( '𝒳'.match(/[𝒳𝒴]/u) ); // 𝒳
```

The similar situation occurs when looking for a range, such as `[𝒳-𝒴]`.

If we forget to add flag `u`, there will be an error:

```
'𝒳'.match(/[𝒳-𝒴]/); // Error: Invalid regular expression
```

The reason is that without flag `u` surrogate pairs are perceived as two characters, so `[𝒳-𝒴]` is interpreted as `[<55349><56499>-<55349><56500>]` (every surrogate pair is replaced with its codes). Now it's easy to see that the range `56499-55349` is invalid: its starting code `56499` is greater than the end `55349`. That's the formal reason for the error.

With the flag `u` the pattern works correctly:

```
// look for characters from 𝒳 to 𝒵
alert( '𝒴'.match(/[𝒳-𝒵]/u) ); // 𝒴
```

Quantifiers +, *, ? and {n}

Let's say we have a string like `+7(903)-123-45-67` and want to find all numbers in it. But unlike before, we are interested not in single digits, but full numbers: `7, 903, 123, 45, 67`.

A number is a sequence of 1 or more digits `\d`. To mark how many we need, we can append a *quantifier*.

Quantity {n}

The simplest quantifier is a number in curly braces: `{n}`.

A quantifier is appended to a character (or a character class, or a `[...]` set etc) and specifies how many we need.

It has a few advanced forms, let's see examples:

The exact count: `{5}`

`\d{5}` denotes exactly 5 digits, the same as `\d\d\d\d\d`.

The example below looks for a 5-digit number:

```
alert( "I'm 12345 years old".match(/\d{5}/) ); //  "12345"
```

We can add `\b` to exclude longer numbers: `\b\d{5}\b`.

The range: `{3,5}`, match 3-5 times

To find numbers from 3 to 5 digits we can put the limits into curly braces: `\d{3,5}`

```
alert( "I'm not 12, but 1234 years old".match(/\d{3,5}/) ); // "1234"
```

We can omit the upper limit.

Then a regexp `\d{3,}` looks for sequences of digits of length `3` or more:

```
alert( "I'm not 12, but 345678 years old".match(/\d{3,}/) ); // "345678"
```

Let's return to the string `+7(903)-123-45-67`.

A number is a sequence of one or more digits in a row. So the regexp is `\d{1,}`:

```
let str = "+7(903)-123-45-67";

let numbers = str.match(/\d{1,}/g);

alert(numbers); // 7,903,123,45,67
```

Shorthands

There are shorthands for most used quantifiers:

`+`

Means "one or more", the same as `{1,}`.

For instance, `\d+` looks for numbers:

```
let str = "+7(903)-123-45-67";

alert( str.match(/\d+/g) ); // 7,903,123,45,67
```

`?`

Means “zero or one”, the same as `{0,1}`. In other words, it makes the symbol optional.

For instance, the pattern `ou?r` looks for `o` followed by zero or one `u`, and then `r`.

So, `colou?r` finds both `color` and `colour`:

```
let str = "Should I write color or colour?";

alert( str.match(/colou?r/g) ); // color, colour
```

`*`

Means “zero or more”, the same as `{0,}`. That is, the character may repeat any times or be absent.

For example, `\d0*` looks for a digit followed by any number of zeroes (may be many or none):

```
alert( "100 10 1".match(/\d0*/g) ); // 100, 10, 1
```

Compare it with `+` (one or more):

```
alert( "100 10 1".match(/\d0+/g) ); // 100, 10
// 1 not matched, as 0+ requires at least one zero
```

More examples

Quantifiers are used very often. They serve as the main “building block” of complex regular expressions, so let’s see more examples.

Regexp for decimal fractions (a number with a floating point): `\d+\.\d+`

In action:

```
alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345
```

Regexp for an “opening HTML-tag without attributes”, such as `<span>` or `<p>`.

1. The simplest one: `/<[a-z]+>/i`

```
alert( "<body> ... </body>".match(/<[a-z]+>/gi) ); // <body>
```

The regexp looks for character `'<'` followed by one or more Latin letters, and then `'>'`.

2. Improved: `/<[a-z][a-z0-9]*>/i`

   According to the standard, HTML tag name may have a digit at any position except the first one, like `<h1>`.

```
alert( "<h1>Hi!</h1>".match(/<[a-z][a-z0-9]*>/gi) ); // <h1>
```

Regexp "opening or closing HTML-tag without attributes": `/<\/?[a-z][a-z0-9]*>/i`

We added an optional slash `/?` near the beginning of the pattern. Had to escape it with a backslash, otherwise JavaScript would think it is the pattern end.

```
alert( "<h1>Hi!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi) ); // <h1>, </h1>
```

To make a regexp more precise, we often need make it more complex

We can see one common rule in these examples: the more precise is the regular expression – the longer and more complex it is.

For instance, for HTML tags we could use a simpler regexp: `<\w+>`. But as HTML has stricter restrictions for a tag name, `<[a-z][a-z0-9]*>` is more reliable.

Can we use `<\w+>` or we need `<[a-z][a-z0-9]*>`?

In real life both variants are acceptable. Depends on how tolerant we can be to "extra" matches and whether it's difficult or not to remove them from the result by other means.

Greedy and lazy quantifiers

Quantifiers are very simple from the first sight, but in fact they can be tricky.

We should understand how the search works very well if we plan to look for something more complex than `/\d+/`.

Let's take the following task as an example.

We have a text and need to replace all quotes `"..."` with guillemet marks: `«...»`. They are preferred for typography in many countries.

For instance: `"Hello, world"` should become `«Hello, world»`. There exist other quotes, such as `„Witam, świat!”` (Polish) or `「你好，世界」` (Chinese), but for our task let’s choose `«...»`.

The first thing to do is to locate quoted strings, and then we can replace them.

A regular expression like `/".+"/g` (a quote, then something, then the other quote) may seem like a good fit, but it isn’t!

Let’s try it:

```
let regexp = /".+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "witch" and her "broom"
```

…We can see that it works not as intended!

Instead of finding two matches `"witch"` and `"broom"`, it finds one: `"witch" and her "broom"`.

That can be described as “greediness is the cause of all evil”.

Greedy search

To find a match, the regular expression engine uses the following algorithm:

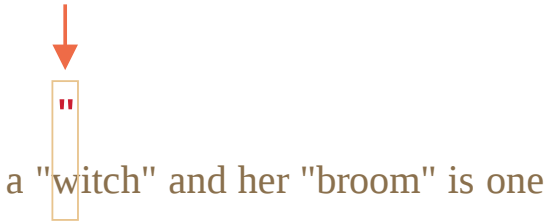
- For every position in the string
  - Try to match the pattern at that position.
  - If there’s no match, go to the next position.

These common words do not make it obvious why the regexp fails, so let’s elaborate how the search works for the pattern `".+"`.

1. The first pattern character is a quote `"`.

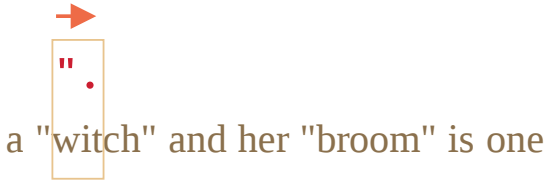
   The regular expression engine tries to find it at the zero position of the source string `a "witch" and her "broom" is one`, but there’s `a` there, so there’s immediately no match.

   Then it advances: goes to the next positions in the source string and tries to find the first character of the pattern there, fails again, and finally finds the quote at the 3rd position:



2. The quote is detected, and then the engine tries to find a match for the rest of the pattern. It tries to see if the rest of the subject string conforms to `.+"`.

   In our case the next pattern character is `.` (a dot). It denotes “any character except a newline”, so the next string letter `'w'` fits:



3. Then the dot repeats because of the quantifier `.+`. The regular expression engine adds to the match one character after another.

   …Until when? All characters match the dot, so it only stops when it reaches the end of the string:

4. Now the engine finished repeating `.+` and tries to find the next character of the pattern. It’s the quote `"`. But there’s a problem: the string has finished, there are no more characters!

   The regular expression engine understands that it took too many `.+` and starts to *backtrack*.

   In other words, it shortens the match for the quantifier by one character:

Now it assumes that `.+` ends one character before the string end and tries to match the rest of the pattern from that position.

If there were a quote there, then the search would end, but the last character is `'e'`, so there's no match.

5. …So the engine decreases the number of repetitions of `.+` by one more character:

The quote `'"'` does not match `'n'`.

6. The engine keep backtracking: it decreases the count of repetition for `'.'` until the rest of the pattern (in our case `'"'`) matches:

7. The match is complete.
8. So the first match is `"witch" and her "broom"`. If the regular expression has flag `g`, then the search will continue from where the first match ends. There are no more quotes in the rest of the string `is one`, so no more results.

That's probably not what we expected, but that's how it works.

In the greedy mode (by default) a quantifier is repeated as many times as possible.

The regexp engine adds to the match as many characters as it can for `.+`, and then shortens that one by one, if the rest of the pattern doesn't match.

For our task we want another thing. That's where a lazy mode can help.

Lazy mode

The lazy mode of quantifiers is an opposite to the greedy mode. It means: "repeat minimal number of times".

We can enable it by putting a question mark `'?'` after the quantifier, so that it becomes `*?` or `+?` or even `??` for `'?'`.

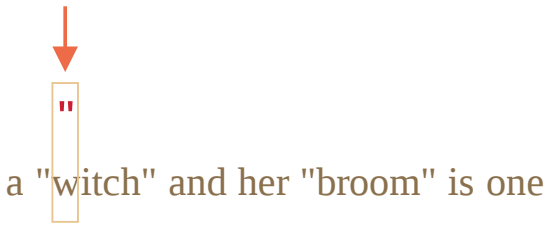
To make things clear: usually a question mark `?` is a quantifier by itself (zero or one), but if added *after another quantifier (or even itself)* it gets another meaning – it switches the matching mode from greedy to lazy.

The regexp `/".+?"/g` works as intended: it finds `"witch"` and `"broom"`:

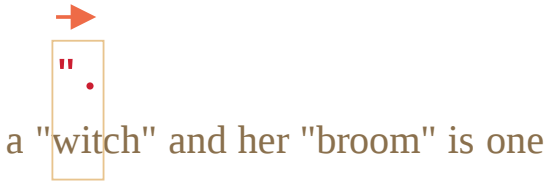
```
let regexp = /".+?"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // witch, broom
```

To clearly understand the change, let's trace the search step by step.

1. The first step is the same: it finds the pattern start `'"'` at the 3rd position:



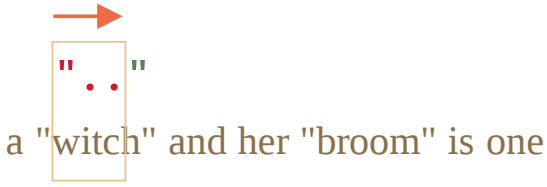
2. The next step is also similar: the engine finds a match for the dot `'.'`:



3. And now the search goes differently. Because we have a lazy mode for `+?`, the engine doesn't try to match a dot one more time, but stops and tries to match the rest of the pattern `'"'` right now:

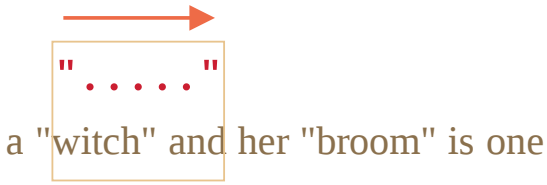
   If there were a quote there, then the search would end, but there's `'i'`, so there's no match.

4. Then the regular expression engine increases the number of repetitions for the dot and tries one more time:



Failure again. Then the number of repetitions is increased again and again…

5. …Till the match for the rest of the pattern is found:



6. The next search starts from the end of the current match and yield one more result:

In this example we saw how the lazy mode works for `+?`. Quantifiers `*?` and `??` work the similar way – the regexp engine increases the number of repetitions only if the rest of the pattern can't match on the given position.

Laziness is only enabled for the quantifier with `?`.

Other quantifiers remain greedy.

For instance:

```
alert( "123 456".match(/\d+ \d+?/) ); // 123 4
```

1. The pattern `\d+` tries to match as many digits as it can (greedy mode), so it finds `123` and stops, because the next character is a space `' '`.
2. Then there's a space in the pattern, it matches.

3. Then there's `\d+?`. The quantifier is in lazy mode, so it finds one digit `4` and tries to check if the rest of the pattern matches from there.

   …But there's nothing in the pattern after `\d+?`.

   The lazy mode doesn't repeat anything without a need. The pattern finished, so we're done. We have a match `123 4`.

> **Optimizations**
>
> Modern regular expression engines can optimize internal algorithms to work faster. So they may work a bit differently from the described algorithm.
>
> But to understand how regular expressions work and to build regular expressions, we don't need to know about that. They are only used internally to optimize things.
>
> Complex regular expressions are hard to optimize, so the search may work exactly as described as well.

Alternative approach

With regexps, there's often more than one way to do the same thing.

In our case we can find quoted strings without lazy mode using the regexp `"[^"]+"`:

```
let regexp = /"[^"]+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // witch, broom
```

The regexp `"[^"]+"` gives correct results, because it looks for a quote `'"'` followed by one or more non-quotes `[^"]`, and then the closing quote.

When the regexp engine looks for `[^"]+` it stops the repetitions when it meets the closing quote, and we're done.

Please note, that this logic does not replace lazy quantifiers!

It is just different. There are times when we need one or another.

Let's see an example where lazy quantifiers fail and this variant works right.

For instance, we want to find links of the form `<a href="..." class="doc">`, with any `href`.

Which regular expression to use?

The first idea might be: `/<a href=".*" class="doc">/g`.

Let's check it:

```
let str = '...<a href="link" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;

// Works!
alert( str.match(regexp) ); // <a href="link" class="doc">
```

It worked. But let's see what happens if there are many links in the text?

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;

// Whoops! Two links in one match!
alert( str.match(regexp) ); // <a href="link1" class="doc">... <a href="link2" clas
```

Now the result is wrong for the same reason as our "witches" example. The quantifier `.*` took too many characters.

The match looks like this:

```
<a href="....................................." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Let's modify the pattern by making the quantifier `.*?` lazy:

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;

// Works!
alert( str.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class=
```

Now it seems to work, there are two matches:

```
<a href="....." class="doc">    <a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

…But let's test it on one more text input:

```
let str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;

// Wrong match!
alert( str.match(regexp) ); // <a href="link1" class="wrong">... <p style="" class=
```

Now it fails. The match includes not just a link, but also a lot of text after it, including `<p...>`.

Why?

That's what's going on:

1. First the regexp finds a link start `<a href="`.
2. Then it looks for `.*?`: takes one character (lazily!), check if there's a match for `" class="doc">` (none).
3. Then takes another character into `.*?`, and so on... until it finally reaches `" class="doc">`.

But the problem is: that's already beyond the link `<a...>`, in another tag `<p>`. Not what we want.

Here's the picture of the match aligned with the text:

```
<a href="..................................." class="doc">
<a href="link1" class="wrong">... <p style="" class="doc">
```

So, we need the pattern to look for `<a href="...something..." class="doc">`, but both greedy and lazy variants have problems.

The correct variant can be: `href="[^"]*"`. It will take all characters inside the `href` attribute till the nearest quote, just what we need.

A working example:

```
let str1 = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let str2 = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href="[^"]*" class="doc">/g;

// Works!
alert( str1.match(regexp) ); // null, no matches, that's correct
alert( str2.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class=
```

Summary

Quantifiers have two modes of work:

Greedy

By default the regular expression engine tries to repeat the quantifier as many times as possible. For instance, `\d+` consumes all possible digits. When it becomes impossible to consume more (no more digits or string end), then it continues to match the rest of the pattern. If there's no match then it decreases the number of repetitions (backtracks) and tries again.

Lazy

Enabled by the question mark `?` after the quantifier. The regexp engine tries to match the rest of the pattern before each repetition of the quantifier.

As we've seen, the lazy mode is not a "panacea" from the greedy search. An alternative is a "fine-tuned" greedy search, with exclusions, as in the pattern `"[^"]+"`.

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@site.com.
```

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

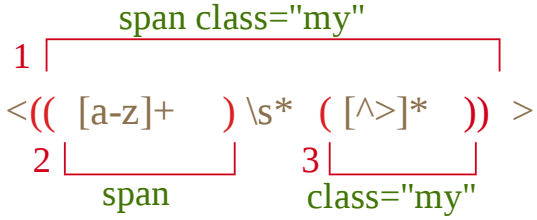
Parentheses can be nested. In this case the numbering also goes from left to right.

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:

```
let str = '<span class="my">';

let regexp = /<(([a-z]+)\s*([^>]*))>/;
```

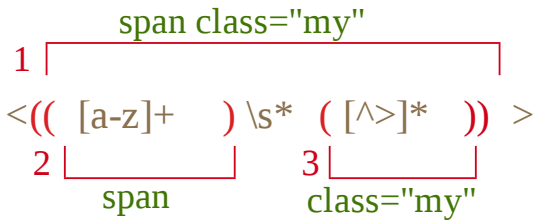
```
let result = str.match(regexp);
alert(result[0]); // <span class="my">
alert(result[1]); // span class="my"
alert(result[2]); // span
alert(result[3]); // class="my"
```

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.

Backreferences in pattern: \N and \k<name>

We can use the contents of capturing groups `(...)` not only in the result or in the replacement string, but also in the pattern itself.

Backreference by number: \N

A group can be referenced in the pattern using `\N`, where `N` is the group number.

To make clear why that's helpful, let's consider a task.

We need to find quoted strings: either single-quoted `'...'` or a double-quoted `"..."` – both variants should match.

How to find them?

We can put both kinds of quotes in the square brackets: `['"](.*?)['"]`, but it would find strings with mixed quotes, like `"...'` and `'..."`. That would lead to incorrect matches when one quote appears inside other ones, like in the string `"She's the one!"`:

```
let str = `He said: "She's the one!".`;

let regexp = /['"](.*?)['"]/g;

// The result is not what we'd like to have
alert( str.match(regexp) ); // "She'
```

As we can see, the pattern found an opening quote `"`, then the text is consumed till the other quote `'`, that closes the match.

To make sure that the pattern looks for the closing quote exactly the same as the opening one, we can wrap it into a capturing group and backreference it: `(['"])(.*?)\1`.

Here's the correct code:

```
let str = `He said: "She's the one!".`;

let regexp = /(['"])(.*?)\1/g;

alert( str.match(regexp) ); // "She's the one!"
```

Now it works! The regular expression engine finds the first quote `(['"])` and memorizes its content. That's the first capturing group.

Further in the pattern `\1` means "find the same text as in the first group", exactly the same quote in our case.

Similar to that, `\2` would mean the contents of the second group, `\3` – the 3rd group, and so on.

Please note:

If we use `?:` in the group, then we can't reference it. Groups that are excluded from capturing `(?:...)` are not memorized by the engine.

Don't mess up: in the pattern `\1`, in the replacement: `$1`

In the replacement string we use a dollar sign: `$1`, while in the pattern – a backslash `\1`.

Backreference by name: `\k<name>`

If a regexp has many parentheses, it's convenient to give them names.

To reference a named group we can use `\k<имя>`.

In the example below the group with quotes is named `?<quote>`, so the backreference is `\k<quote>`:

```
let str = `He said: "She's the one!".`;

let regexp = /(?<quote>['"])(.*?)\k<quote>/g;

alert( str.match(regexp) ); // "She's the one!"
```

Alternation (OR) |

Alternation is the term in regular expression that is actually a simple "OR".

In a regular expression it is denoted with a vertical line character `|`.

For instance, we need to find programming languages: HTML, PHP, Java or JavaScript.

The corresponding regexp: `html|php|java(script)?`.

A usage example:

```
let regexp = /html|php|css|java(script)?/gi;

let str = "First HTML appeared, then CSS, then JavaScript";

alert( str.match(regexp) ); // 'HTML', 'CSS', 'JavaScript'
```

We already saw a similar thing – square brackets. They allow to choose between multiple characters, for instance `gr[ae]y` matches `gray` or `grey`.

Square brackets allow only characters or character sets. Alternation allows any expressions. A regexp `A|B|C` means one of expressions `A`, `B` or `C`.

For instance:

- `gr(a|e)y` means exactly the same as `gr[ae]y`.
- `gra|ey` means `gra` or `ey`.

To apply alternation to a chosen part of the pattern, we can enclose it in parentheses:

- `I love HTML|CSS` matches `I love HTML` or `CSS`.
- `I love (HTML|CSS)` matches `I love HTML` or `I love CSS`.

Example: regexp for time

In previous articles there was a task to build a regexp for searching time in the form `hh:mm`, for instance `12:00`. But a simple `\d\d:\d\d` is too vague. It accepts `25:99` as the time (as 99 seconds match the pattern, but that time is invalid).

How can we make a better pattern?

We can use more careful matching. First, the hours:

- If the first digit is `0` or `1`, then the next digit can be any: `[01]\d`.
- Otherwise, if the first digit is `2`, then the next must be `[0-3]`.
- (no other first digit is allowed)

We can write both variants in a regexp using alternation: `[01]\d|2[0-3]`.

Next, minutes must be from `00` to `59`. In the regular expression language that can be written as `[0-5]\d`: the first digit `0-5`, and then any digit.

If we glue minutes and seconds together, we get the pattern: `[01]\d|2[0-3]:[0-5]\d`.

We're almost done, but there's a problem. The alternation `|` now happens to be between `[01]\d` and `2[0-3]:[0-5]\d`.

That is: minutes are added to the second alternation variant, here's a clear picture:

```
[01]\d  |  2[0-3]:[0-5]\d
```

That pattern looks for `[01]\d` or `2[0-3]:[0-5]\d`.

But that's wrong, the alternation should only be used in the "hours" part of the regular expression, to allow `[01]\d` OR `2[0-3]`. Let's correct that by enclosing "hours" into parentheses: `([01]\d|2[0-3]):[0-5]\d`.

The final solution:

```
let regexp = /([01]\d|2[0-3]):[0-5]\d/g;

alert("00:00 10:10 23:59 25:99 1:2".match(regexp)); // 00:00,10:10,23:59
```

Lookahead and lookbehind

Sometimes we need to find only those matches for a pattern that are followed or preceeded by another pattern.

There's a special syntax for that, called "lookahead" and "lookbehind", together referred to as "lookaround".

For the start, let's find the price from the string like `1 turkey costs 30€`. That is: a number, followed by `€` sign.

Lookahead

The syntax is: `X(?=Y)`, it means "look for `X`, but match only if followed by `Y`". There may be any pattern instead of `X` and `Y`.

For an integer number followed by `€`, the regexp will be `\d+(?=€)`:

```
let str = "1 turkey costs 30€";

alert( str.match(/\d+(?=€)/) ); // 30, the number 1 is ignored, as it's not followe
```

Please note: the lookahead is merely a test, the contents of the parentheses `(?=...)` is not included in the result `30`.

When we look for `X(?=Y)`, the regular expression engine finds `X` and then checks if there's `Y` immediately after it. If it's not so, then the potential match is skipped, and the search continues.

More complex tests are possible, e.g. `X(?=Y)(?=Z)` means:

1. Find `X`.
2. Check if `Y` is immediately after `X` (skip if isn't).
3. Check if `Z` is immediately after `Y` (skip if isn't).
4. If both tests passed, then it's the match.

In other words, such pattern means that we're looking for `X` followed by `Y` and `Z` at the same time.

That's only possible if patterns `Y` and `Z` aren't mutually exclusive.

For example, `\d+(?=\s)(?=.*30)` looks for `\d+` only if it's followed by a space, and there's `30` somewhere after it:

```
let str = "1 turkey costs 30€";

alert( str.match(/\d+(?=\s)(?=.*30)/) ); // 1
```

In our string that exactly matches the number `1`.

Negative lookahead

Let's say that we want a quantity instead, not a price from the same string. That's a number `\d+`, NOT followed by `€`.

For that, a negative lookahead can be applied.

The syntax is: `X(?!Y)`, it means "search `X`, but only if not followed by `Y`".

```
let str = "2 turkeys cost 60€";

alert( str.match(/\d+(?!€)/) ); // 2 (the price is skipped)
```

Lookbehind

Lookahead allows to add a condition for "what follows".

Lookbehind is similar, but it looks behind. That is, it allows to match a pattern only if there's something before it.

The syntax is:

- Positive lookbehind: `(?<=Y)X`, matches `X`, but only if there's `Y` before it.
- Negative lookbehind: `(?<!Y)X`, matches `X`, but only if there's no `Y` before it.

For example, let's change the price to US dollars. The dollar sign is usually before the number, so to look for `$30` we'll use `(?<=\$)\d+` – an amount preceded by `$`:

```
let str = "1 turkey costs $30";

// the dollar sign is escaped \$
alert( str.match(/(?<=\$)\d+/) ); // 30 (skipped the sole number)
```

And, if we need the quantity – a number, not preceded by `$`, then we can use a negative lookbehind `(?<!\$)\d+`:

```
let str = "2 turkeys cost $60";

alert( str.match(/(?<!\$)\d+/) ); // 2 (skipped the price)
```

Capturing groups

Generally, the contents inside lookaround parentheses does not become a part of the result.

E.g. in the pattern `\d+(?=€)`, the `€` sign doesn't get captured as a part of the match. That's natural: we look for a number `\d+`, while `(?=€)` is just a test that it should be followed by `€`.

But in some situations we might want to capture the lookaround expression as well, or a part of it. That's possible. Just wrap that part into additional parentheses.

In the example below the currency sign `(€|kr)` is captured, along with the amount:

```
let str = "1 turkey costs 30€";
let regexp = /\d+(?=(€|kr))/; // extra parentheses around €|kr

alert( str.match(regexp) ); // 30, €
```

And here's the same for lookbehind:

```
let str = "1 turkey costs $30";
let regexp = /(?<=(\$|£))\d+/;

alert( str.match(regexp) ); // 30, $
```

Summary

Lookahead and lookbehind (commonly referred to as "lookaround") are useful when we'd like to match something depending on the context before/after it.

For simple regexps we can do the similar thing manually. That is: match everything, in any context, and then filter by context in the loop.

Remember, `str.match` (without flag `g`) and `str.matchAll` (always) return matches as arrays with `index` property, so we know where exactly in the text it is, and can check the context.

But generally lookaround is more convenient.

Lookaround types:

Pattern	type	matches
`X(?=Y)`	Positive lookahead	`X` if followed by `Y`
`X(?!Y)`	Negative lookahead	`X` if not followed by `Y`

Pattern	type	matches
`(?<=Y)X`	Positive lookbehind	`X` if after `Y`
`(?<!Y)X`	Negative lookbehind	`X` if not after `Y`

Catastrophic backtracking

Some regular expressions are looking simple, but can execute veeeeeery long time, and even “hang” the JavaScript engine.

Sooner or later most developers occasionally face such behavior, because it’s quite easy to create such a regexp.

The typical symptom – a regular expression works fine sometimes, but for certain strings it “hangs”, consuming 100% of CPU.

In such case a web-browser suggests to kill the script and reload the page. Not a good thing for sure.

For server-side JavaScript it may become a vulnerability if regular expressions process user data.

Example

Let’s say we have a string, and we’d like to check if it consists of words `\w+` with an optional space `\s?` after each.

We’ll use a regexp `^(\w+\s?)*$`, it specifies 0 or more such words.

In action:

```
let regexp = /^(\w+\s?)*$/;

alert( regexp.test("A good string") ); // true
alert( regexp.test("Bad characters: $@#") ); // false
```

It seems to work. The result is correct. Although, on certain strings it takes a lot of time. So long that JavaScript engine “hangs” with 100% CPU consumption.

If you run the example below, you probably won’t see anything, as JavaScript will just “hang”. A web-browser will stop reacting on events, the UI will stop working. After some time it will suggest to reloaad the page. So be careful with this:

```
let regexp = /^(\w+\s?)*$/;
let str = "An input string that takes a long time or even makes this regexp to hang
```

```
// will take a very long time
alert( regexp.test(str) );
```

Some regular expression engines can handle such search, but most of them can't.

Simplified example

What's the matter? Why the regular expression "hangs"?

To understand that, let's simplify the example: remove spaces `\s?`. Then it becomes `^(\w+)*$`.

And, to make things more obvious, let's replace `\w` with `\d`. The resulting regular expression still hangs, for instance:

```
let regexp = /^(\d+)*$/;

let str = "012345678901234567890123456789!";

// will take a very long time
alert( regexp.test(str) );
```

So what's wrong with the regexp?

First, one may notice that the regexp `(\d+)*` is a little bit strange. The quantifier `*` looks extraneous. If we want a number, we can use `\d+`.

Indeed, the regexp is artificial. But the reason why it is slow is the same as those we saw above. So let's understand it, and then the previous example will become obvious.

What happens during the search of `^(\d+)*$` in the line `123456789!` (shortened a bit for clarity), why does it take so long?

1. First, the regexp engine tries to find a number `\d+`. The plus `+` is greedy by default, so it consumes all digits:

   ```
   \d+.......
   (123456789)z
   ```

   Then it tries to apply the star quantifier, but there are no more digits, so it the star doesn't give anything.

   The next in the pattern is the string end `$`, but in the text we have `!`, so there's no match:

```
          X
\d+.......$
(123456789)!
```

2. As there's no match, the greedy quantifier `+` decreases the count of repetitions, backtracks one character back.

   Now `\d+` takes all digits except the last one:

```
\d+.......
(12345678)9!
```

3. Then the engine tries to continue the search from the new position (`9`).

   The star `(\d+)*` can be applied – it gives the number `9`:

```
\d+.......\d+
(12345678)(9)!
```

   The engine tries to match `$` again, but fails, because meets `!`:

```
            X
\d+.......\d+
(12345678)(9)z
```

4. There's no match, so the engine will continue backtracking, decreasing the number of repetitions. Backtracking generally works like this: the last greedy quantifier decreases the number of repetitions until it can. Then the previous greedy quantifier decreases, and so on.

   All possible combinations are attempted. Here are their examples.

   The first number `\d+` has 7 digits, and then a number of 2 digits:

```
            X
\d+......\d+
(1234567)(89)!
```

   The first number has 7 digits, and then two numbers of 1 digit each:

```
              X
\d+.....\d+\d+
(1234567)(8)(9)!
```

The first number has 6 digits, and then a number of 3 digits:

```
            X
\d+.......\d+
(123456)(789)!
```

The first number has 6 digits, and then 2 numbers:

```
              X
\d+.....\d+ \d+
(123456)(78)(9)!
```

…And so on.

There are many ways to split a set of digits `123456789` into numbers. To be precise, there are 2^n-1, where `n` is the length of the set.

For `n=20` there are about 1 million combinations, for `n=30` – a thousand times more. Trying each of them is exactly the reason why the search takes so long.

What to do?

Should we turn on the lazy mode?

Unfortunately, that won't help: if we replace `\d+` with `\d+?`, the regexp will still hang. The order of combinations will change, but not their total count.

Some regular expression engines have tricky tests and finite automations that allow to avoid going through all combinations or make it much faster, but not all engines, and not in all cases.

Back to words and strings

The similar thing happens in our first example, when we look words by pattern `^(\w+\s?)*$` in the string `An input that hangs!`.

The reason is that a word can be represented as one `\w+` or many:

```
(input)
(inpu)(t)
```

```
(inp)(u)(t)
(in)(p)(ut)
...
```

For a human, it's obvious that there may be no match, because the string ends with an exclamation sign `!`, but the regular expression expects a wordly character `\w` or a space `\s` at the end. But the engine doesn't know that.

It tries all combinations of how the regexp `(\w+\s?)*` can "consume" the string, including variants with spaces `(\w+\s)*` and without them `(\w+)*` (because spaces `\s?` are optional). As there are many such combinations, the search takes a lot of time.

How to fix?

There are two main approaches to fixing the problem.

The first is to lower the number of possible combinations.

Let's rewrite the regular expression as `^(\w+\s)*\w*` – we'll look for any number of words followed by a space `(\w+\s)*`, and then (optionally) a word `\w*`.

This regexp is equivalent to the previous one (matches the same) and works well:

```
let regexp = /^(\w+\s)*\w*$/;
let str = "An input string that takes a long time or even makes this regex to hang!

alert( regexp.test(str) ); // false
```

Why did the problem disappear?

Now the star `*` goes after `\w+\s` instead of `\w+\s?`. It became impossible to represent one word of the string with multiple successive `\w+`. The time needed to try such combinations is now saved.

For example, the previous pattern `(\w+\s?)*` could match the word `string` as two `\w+`:

```
\w+\w+
string
```

The previous pattern, due to the optional `\s` allowed variants `\w+`, `\w+\s`, `\w+\w+` and so on.

With the rewritten pattern `(\w+\s)*`, that's impossible: there may be `\w+\s` or `\w+\s\w+\s`, but not `\w+\w+`. So the overall combinations count is greatly decreased.

Preventing backtracking

It's not always convenient to rewrite a regexp. And it's not always obvious how to do it.

The alternative approach is to forbid backtracking for the quantifier.

The regular expressions engine tries many combinations that are obviously wrong for a human.

E.g. in the regexp `(\d+)*$` it's obvious for a human, that `+` shouldn't backtrack. If we replace one `\d+` with two separate `\d+\d+`, nothing changes:

```
\d+........
(123456789)!

\d+...\d+....
(1234)(56789)!
```

And in the original example `^(\w+\s?)*$` we may want to forbid backtracking in `\w+`. That is: `\w+` should match a whole word, with the maximal possible length. There's no need to lower the repetitions count in `\w+`, try to split it into two words `\w+\w+` and so on.

Modern regular expression engines support possessive quantifiers for that. They are like greedy ones, but don't backtrack (so they are actually simpler than regular quantifiers).

There are also so-called "atomic capturing groups" – a way to disable backtracking inside parentheses.

Unfortunately, in JavaScript they are not supported. But there's another way.

Lookahead to the rescue!

We can prevent backtracking using lookahead.

The pattern to take as much repetitions of `\w` as possible without backtracking is: `(?=(\w+))\1`.

Let's decipher it:

- Lookahead `?=` looks forward for the longest word `\w+` starting at the current position.

- The contents of parentheses with `?=...` isn't memorized by the engine, so wrap `\w+` into parentheses. Then the engine will memorize their contents
- …And allow us to reference it in the pattern as `\1`.

That is: we look ahead – and if there's a word `\w+`, then match it as `\1`.

Why? That's because the lookahead finds a word `\w+` as a whole and we capture it into the pattern with `\1`. So we essentially implemented a possessive plus `+` quantifier. It captures only the whole word `\w+`, not a part of it.

For instance, in the word `JavaScript` it may not only match `Java`, but leave out `Script` to match the rest of the pattern.

Here's the comparison of two patterns:

```
alert( "JavaScript".match(/\w+Script/)); // JavaScript
alert( "JavaScript".match(/(?=(\w+))\1Script/)); // null
```

1. In the first variant `\w+` first captures the whole word `JavaScript` but then `+` backtracks character by character, to try to match the rest of the pattern, until it finally succeeds (when `\w+` matches `Java`).
2. In the second variant `(?=(\w+))` looks ahead and finds the word `JavaScript`, that is included into the pattern as a whole by `\1`, so there remains no way to find `Script` after it.

We can put a more complex regular expression into `(?=(\w+))\1` instead of `\w`, when we need to forbid backtracking for `+` after it.

Please note:

There's more about the relation between possessive quantifiers and lookahead in articles Regex: Emulate Atomic Grouping (and Possessive Quantifiers) with LookAhead and Mimicking Atomic Groups.

Let's rewrite the first example using lookahead to prevent backtracking:

```
let regexp = /^((?=(\w+))\2\s?)*$/;

alert( regexp.test("A good string") ); // true

let str = "An input string that takes a long time or even makes this regex to hang!"

alert( regexp.test(str) ); // false, works and fast!
```

Here `\2` is used instead of `\1`, because there are additional outer parentheses. To avoid messing up with the numbers, we can give the parentheses a name, e.g. `(?<word>\w+)`.

```
// parentheses are named ?<word>, referenced as \k<word>
let regexp = /^((?=(?<word>\w+))\k<word>\s?)*$/;

let str = "An input string that takes a long time or even makes this regex to hang!

alert( regexp.test(str) ); // false

alert( regexp.test("A correct string") ); // true
```

The problem described in this article is called “catastrophic backtracking”.

We covered two ways how to solve it:

- Rewrite the regexp to lower the possible combinations count.
- Prevent backtracking.

Sticky flag "y", searching at position

The flag `y` allows to perform the search at the given position in the source string.

To grasp the use case of `y` flag, and see how great it is, let’s explore a practical use case.

One of common tasks for regexps is “lexical analysis”: we get a text, e.g. in a programming language, and analyze it for structural elements.

For instance, HTML has tags and attributes, JavaScript code has functions, variables, and so on.

Writing lexical analyzers is a special area, with its own tools and algorithms, so we don’t go deep in there, but there’s a common task: to read something at the given position.

E.g. we have a code string `let varName = "value"`, and we need to read the variable name from it, that starts at position `4`.

We’ll look for variable name using regexp `\w+`. Actually, JavaScript variable names need a bit more complex regexp for accurate matching, but here it doesn’t matter.

A call to `str.match(/\w+/)` will find only the first word in the line. Or all words with the flag `g`. But we need only one word at position `4`.

To search from the given position, we can use method `regexp.exec(str)`.

If the `regexp` doesn't have flags `g` or `y`, then this method looks for the first match in the string `str`, exactly like `str.match(regexp)`. Such simple no-flags case doesn't interest us here.

If there's flag `g`, then it performs the search in the string `str`, starting from position stored in its `regexp.lastIndex` property. And, if it finds a match, then sets `regexp.lastIndex` to the index immediately after the match.

When a regexp is created, its `lastIndex` is `0`.

So, successive calls to `regexp.exec(str)` return matches one after another.

An example (with flag `g`):

```
let str = 'let varName';

let regexp = /\w+/g;
alert(regexp.lastIndex); // 0 (initially lastIndex=0)

let word1 = regexp.exec(str);
alert(word1[0]); // let (1st word)
alert(regexp.lastIndex); // 3 (position after the match)

let word2 = regexp.exec(str);
alert(word2[0]); // varName (2nd word)
alert(regexp.lastIndex); // 11 (position after the match)

let word3 = regexp.exec(str);
alert(word3); // null (no more matches)
alert(regexp.lastIndex); // 0 (resets at search end)
```

Every match is returned as an array with groups and additional properties.

We can get all matches in the loop:

```
let str = 'let varName';
let regexp = /\w+/g;

let result;

while (result = regexp.exec(str)) {
  alert( `Found ${result[0]} at position ${result.index}` );
  // Found let at position 0, then
  // Found varName at position 4
}
```

Such use of `regexp.exec` is an alternative to method `str.matchAll`.

Unlike other methods, we can set our own `lastIndex`, to start the search from the given position.

For instance, let's find a word, starting from position `4`:

```
let str = 'let varName = "value"';

let regexp = /\w+/g; // without flag "g", property lastIndex is ignored

regexp.lastIndex = 4;

let word = regexp.exec(str);
alert(word); // varName
```

We performed a search of `\w+`, starting from position `regexp.lastIndex = 4`.

Please note: the search starts at position `lastIndex` and then goes further. If there's no word at position `lastIndex`, but it's somewhere after it, then it will be found:

```
let str = 'let varName = "value"';

let regexp = /\w+/g;

regexp.lastIndex = 3;

let word = regexp.exec(str);
alert(word[0]); // varName
alert(word.index); // 4
```

…So, with flag `g` property `lastIndex` sets the starting position for the search.

Flag `y` makes `regexp.exec` to look exactly at position `lastIndex`, not before, not after it.

Here's the same search with flag `y`:

```
let str = 'let varName = "value"';

let regexp = /\w+/y;

regexp.lastIndex = 3;
alert( regexp.exec(str) ); // null (there's a space at position 3, not a word)

regexp.lastIndex = 4;
alert( regexp.exec(str) ); // varName (word at position 4)
```

As we can see, regexp `/\w+/y` doesn't match at position `3` (unlike the flag `g`), but matches at position `4`.

Imagine, we have a long text, and there are no matches in it, at all. Then searching with flag `g` will go till the end of the text, and this will take significantly more time than the search with flag `y`.

In such tasks like lexical analysis, there are usually many searches at an exact position. Using flag `y` is the key for a good performance.

Methods of RegExp and String

In this article we'll cover various methods that work with regexps in-depth.

str.match(regexp)

The method `str.match(regexp)` finds matches for `regexp` in the string `str`.

It has 3 modes:

1. If the `regexp` doesn't have flag `g`, then it returns the first match as an array with capturing groups and properties `index` (position of the match), `input` (input string, equals `str`):

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/);

alert( result[0] );     // JavaScript (full match)
alert( result[1] );     // Script (first capturing group)
alert( result.length ); // 2

// Additional information:
alert( result.index );  // 0 (match position)
alert( result.input );  // I love JavaScript (source string)
```

2. If the `regexp` has flag `g`, then it returns an array of all matches as strings, without capturing groups and other details.

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/g);

alert( result[0] ); // JavaScript
alert( result.length ); // 1
```

3. If there are no matches, no matter if there's flag `g` or not, `null` is returned.

   That's an important nuance. If there are no matches, we don't get an empty array, but `null`. It's easy to make a mistake forgetting about it, e.g.:

```
let str = "I love JavaScript";

let result = str.match(/HTML/);

alert(result); // null
alert(result.length); // Error: Cannot read property 'length' of null
```

   If we want the result to be an array, we can write like this:

```
let result = str.match(regexp) || [];
```

str.matchAll(regexp)

> **A recent addition**
>
> This is a recent addition to the language. Old browsers may need polyfills.

The method `str.matchAll(regexp)` is a "newer, improved" variant of `str.match`.

It's used mainly to search for all matches with all groups.

There are 3 differences from `match`:

1. It returns an iterable object with matches instead of an array. We can make a regular array from it using `Array.from`.
2. Every match is returned as an array with capturing groups (the same format as `str.match` without flag `g`).
3. If there are no results, it returns not `null`, but an empty iterable object.

Usage example:

```
let str = '<h1>Hello, world!</h1>';
let regexp = /<(.*?)>/g;

let matchAll = str.matchAll(regexp);

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

matchAll = Array.from(matchAll); // array now
```

```
let firstMatch = matchAll[0];
alert( firstMatch[0] );  // <h1>
alert( firstMatch[1] );  // h1
alert( firstMatch.index );  // 0
alert( firstMatch.input );  // <h1>Hello, world!</h1>
```

If we use `for..of` to loop over `matchAll` matches, then we don't need `Array.from`, разумеется, не нужен.

str.split(regexp|substr, limit)

Splits the string using the regexp (or a substring) as a delimiter.

We can use `split` with strings, like this:

```
alert('12-34-56'.split('-')) // array of [12, 34, 56]
```

But we can split by a regular expression, the same way:

```
alert('12, 34, 56'.split(/,\s*/)) // array of [12, 34, 56]
```

str.search(regexp)

The method `str.search(regexp)` returns the position of the first match or `-1` if none found:

```
let str = "A drop of ink may make a million think";

alert( str.search( /ink/i ) ); // 10 (first match position)
```

The important limitation: `search` only finds the first match.

If we need positions of further matches, we should use other means, such as finding them all with `str.matchAll(regexp)`.

str.replace(str|regexp, str|func)

This is a generic method for searching and replacing, one of most useful ones. The swiss army knife for searching and replacing.

We can use it without regexps, to search and replace a substring:

```
// replace a dash by a colon
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

There's a pitfall though.

When the first argument of `replace` is a string, it only replaces the first match.

You can see that in the example above: only the first `"-"` is replaced by `":"`.

To find all hyphens, we need to use not the string `"-"`, but a regexp `/-/g`, with the obligatory `g` flag:

```
// replace all dashes by a colon
alert( '12-34-56'.replace( /-/g, ":" ) )  // 12:34:56
```

The second argument is a replacement string. We can use special character in it:

| Symbols | Action in the replacement string |

Symbols	Action in the replacement string
`$&`	inserts the whole match
`$``	inserts a part of the string before the match
`$'`	inserts a part of the string after the match
`$n`	if `n` is a 1-2 digit number, inserts the contents of n-th capturing group, for details see Capturing groups
`$<name>`	inserts the contents of the parentheses with the given `name`, for details see Capturing groups
`$$`	inserts character `$`

For instance:

```
let str = "John Smith";

// swap first and last name
alert(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

For situations that require "smart" replacements, the second argument can be a function.

It will be called for each match, and the returned value will be inserted as a replacement.

The function is called with arguments `func(match, p1, p2, ..., pn, offset, input, groups)`:

1. `match` – the match,
2. `p1, p2, ..., pn` – contents of capturing groups (if there are any),
3. `offset` – position of the match,
4. `input` – the source string,
5. `groups` – an object with named groups.

If there are no parentheses in the regexp, then there are only 3 arguments: `func(str, offset, input)`.

For example, let's uppercase all matches:

```
let str = "html and css";

let result = str.replace(/html|css/gi, str => str.toUpperCase());

alert(result); // HTML and CSS
```

Replace each match by its position in the string:

```
alert("Ho-Ho-ho".replace(/ho/gi, (match, offset) => offset)); // 0-3-6
```

In the example below there are two parentheses, so the replacement function is called with 5 arguments: the first is the full match, then 2 parentheses, and after it (not used in the example) the match position and the source string:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (match, name, surname) => `${surname}, ${na

alert(result); // Smith, John
```

If there are many groups, it's convenient to use rest parameters to access them:

Если в регулярном выражении много скобочных групп, то бывает удобно использовать остаточные аргументы для обращения к ним:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (...match) => `${match[2]}, ${match[1]}`);
```

```
alert(result); // Smith, John
```

Or, if we're using named groups, then `groups` object with them is always the last, so we can obtain it like this:

```
let str = "John Smith";

let result = str.replace(/(?<name>\w+) (?<surname>\w+)/, (...match) => {
  let groups = match.pop();

  return `${groups.surname}, ${groups.name}`;
});

alert(result); // Smith, John
```

Using a function gives us the ultimate replacement power, because it gets all the information about the match, has access to outer variables and can do everything.

regexp.exec(str)

The method `regexp.exec(str)` method returns a match for `regexp` in the string `str`. Unlike previous methods, it's called on a regexp, not on a string.

It behaves differently depending on whether the regexp has flag `g`.

If there's no `g`, then `regexp.exec(str)` returns the first match exactly as `str.match(regexp)`. This behavior doesn't bring anything new.

But if there's flag `g`, then:

- A call to `regexp.exec(str)` returns the first match and saves the position immediately after it in the property `regexp.lastIndex`.
- The next such call starts the search from position `regexp.lastIndex`, returns the next match and saves the position after it in `regexp.lastIndex`.
- …And so on.
- If there are no matches, `regexp.exec` returns `null` and resets `regexp.lastIndex` to `0`.

So, repeated calls return all matches one after another, using property `regexp.lastIndex` to keep track of the current search position.

In the past, before the method `str.matchAll` was added to JavaScript, calls of `regexp.exec` were used in the loop to get all matches with groups:

```
let str = 'More about JavaScript at https://javascript.info';
let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
  alert( `Found ${result[0]} at position ${result.index}` );
  // Found JavaScript at position 11, then
  // Found javascript at position 33
}
```

This works now as well, although for newer browsers `str.matchAll` is usually more convenient.

We can use `regexp.exec` to search from a given position by manually setting `lastIndex`.

For instance:

```
let str = 'Hello, world!';

let regexp = /\w+/g; // without flag "g", lastIndex property is ignored
regexp.lastIndex = 5; // search from 5th position (from the comma)

alert( regexp.exec(str) ); // world
```

If the regexp has flag `y`, then the search will be performed exactly at the position `regexp.lastIndex`, not any further.

Let's replace flag `g` with `y` in the example above. There will be no matches, as there's no word at position `5`:

```
let str = 'Hello, world!';

let regexp = /\w+/y;
regexp.lastIndex = 5; // search exactly at position 5

alert( regexp.exec(str) ); // null
```

That's convenient for situations when we need to "read" something from the string by a regexp at the exact position, not somewhere further.

regexp.test(str)

The method `regexp.test(str)` looks for a match and returns `true/false` whether it exists.

For instance:

```
let str = "I love JavaScript";

// these two tests do the same
alert( /love/i.test(str) ); // true
alert( str.search(/love/i) != -1 ); // true
```

An example with the negative answer:

```
let str = "Bla-bla-bla";

alert( /love/i.test(str) ); // false
alert( str.search(/love/i) != -1 ); // false
```

If the regexp has flag `g`, then `regexp.test` looks from `regexp.lastIndex` property and updates this property, just like `regexp.exec`.

So we can use it to search from a given position:

```
let regexp = /love/gi;

let str = "I love JavaScript";

// start the search from position 10:
regexp.lastIndex = 10;
alert( regexp.test(str) ); // false (no match)
```

Same global regexp tested repeatedly on different sources may fail

If we apply the same global regexp to different inputs, it may lead to wrong result, because `regexp.test` call advances `regexp.lastIndex` property, so the search in another string may start from non-zero position.

For instance, here we call `regexp.test` twice on the same text, and the second time fails:

```
let regexp = /javascript/g;  // (regexp just created: regexp.lastIndex=0)

alert( regexp.test("javascript") ); // true (regexp.lastIndex=10 now)
alert( regexp.test("javascript") ); // false
```

That's exactly because `regexp.lastIndex` is non-zero in the second test.

To work around that, we can set `regexp.lastIndex = 0` before each search. Or instead of calling methods on regexp, use string methods `str.match/search/...`, they don't use `lastIndex`.