# JavaScript Regex

Syntax, flags and matching.



## 01 Character classes

| | |
|---|---|
| `.` | Any character but a line break |
| `\d` `\D` | Digit 0-9 / anything else |
| `\w` `\W` | a-z, A-Z, 0-9, _ / anything else |
| `\s` `\S` | Whitespace / anything else |
| `[abc]` `[^abc]` | One of a, b, c / none of them |
| `[a-z]` `[0-9]` | A range of characters |
| `\.` `\$` `\/` | A literal . $ or / |

The dot skips \n and \r: add the s flag or use [\s\S] to match anything. \d and \w are ASCII-only, even with the u flag.

## 02 Anchors & boundaries

```
/^cat/.test('cat nap');      // true
/cat$/.test('tomcat');       // true
/\bcat\b/.test('cat nap');   // true
/\bcat\b/.test('tomcat');    // false
/^nap/m.test('cat\nnap');    // true
```

^ and $ are the ends of the string, or of each line with the m flag. \b is a zero-width word boundary, \B its opposite.

## 03 Quantifiers: greedy & lazy

| | |
|---|---|
| `*` `+` `?` | 0 or more / 1 or more / 0 or 1 |
| `{3}` `{2,}` `{2,4}` | Exactly 3 / 2 or more / 2 to 4 |
| `*?` `+?` `??` `{2,}?` | Lazy: as few as possible |

```
'12345'.match(/\d{2,4}/)[0];   // '1234'
'<b>x</b>'.match(/<.*>/)[0];   // '<b>x</b>'
'<b>x</b>'.match(/<.*?>/)[0];  // '<b>'
```

Greedy by default: a quantifier takes all it can.

## 04 Groups & backreferences

```
const m = 'John Smith'.match(/(\w+) (\w+)/);
m[0];                        // 'John Smith'
m[1];                        // 'John'
m[2];                        // 'Smith'
/(?:ab)+c/.test('ababc');    // true
/(\w)\1/.test('hello');      // true
/(\w+) \1/.test('ha ha');    // true
```

(x) captures, numbered by opening parenthesis; (?:x) only groups. \1 matches the text that group 1 matched.

## 05 Named groups

```
const re = /(?<year>\d{4})-(?<month>\d\d)/;
const m = '2024-03'.match(re);
m.groups.year;                  // '2024'
m.groups.month;                 // '03'
/(?<c>\w)\k<c>/.test('hello');  // true
const u = /(?<n>\d+)px|(?<n>\d+)em/;
u.exec('12em').groups.n;        // '12'
```

\k<name> is the backreference. ES2025 lets two alternatives share a name, as in the last pattern.

## 06 Lookahead & lookbehind

```
/\d+(?=px)/.exec('100px')[0];   // '100'
/\d+(?!px)/.exec('100em')[0];   // '100'
/\d+(?!px)/.exec('100px')[0];   // '10'
/(?<=\$)\d+/.exec('$100')[0];   // '100'
/(?<!\$)\d+/.exec('$100')[0];   // '00'
/\w+(?=@)/.exec('me@x.io')[0];  // 'me'
```

(?=x) and (?!x) look ahead; (?<=x) and (?<!x) look behind. Both are zero-width: checked but not returned. A negative one can back off to a shorter match, hence '10' and '00'.

## 07 Flags

| | |
|---|---|
| `g` | All matches, not only the first |
| `i` | Ignore case |
| `m` | ^ and $ match at each line |
| `s` | . also matches line breaks |
| `u` | Unicode: \u{...} and \p{...} |
| `v` | u plus set operations (ES2024) |
| `y` | Sticky: match only at lastIndex |
| `d` | Add match.indices (ES2022) |

u and v cannot be combined. ES2025 modifiers set i, m or s for part of a pattern: (?i:x) on, (?-i:x) off.

## 08 test() & exec()

```
/\d+/.test('abc123');    // true
const m = /(\d+)-(\d+)/.exec('tel 555-12');
m[0];                    // '555-12'
m[1];                    // '555'
m.index;                 // 4
```

exec returns the match, its groups and index, or null. With g or y, test and exec start at lastIndex (18).

## 09 match() & matchAll()

```
'a1b22'.match(/\d+/)[0];    // '1'
'a1b22'.match(/\d+/g);      // ['1', '22']
'abc'.match(/\d+/g);        // null
const kv = 'x=1 y=2'.matchAll(/(\w)=(\d)/g);
[...kv].map(m => m[2]);     // ['1', '2']
```

With g, match returns every match as a plain string, without groups. matchAll (ES2020) keeps them and throws without g.

# JavaScript Regex

Replace, split, Unicode and gotchas.



## 10 replace() & replaceAll()

```
'a-b-c'.replace(/-/, '+');      // 'a+b-c'
'a-b-c'.replace(/-/g, '+');     // 'a+b+c'
'a-b-c'.replaceAll(/-/g, '+');  // 'a+b+c'
'a-b-c'.replaceAll(/-/, '+');   // TypeError
'a1b2'.replace(/\d/g, d => d * 2);
// 'a2b4'
```

Without g, replace changes only the first match and replaceAll (ES2021) throws. A function gets (match, ...groups).

## 11 Replacement patterns

```
const re = /(?<a>\w+) (?<b>\w+)/;
const s = 'Jo Kim';
s.replace(re, '$2, $1');      // 'Kim, Jo'
s.replace(re, '$<b>, $<a>');  // 'Kim, Jo'
'abc'.replace(/b/, '[$&]');   // 'a[b]c'
'abc'.replace(/b/, '[$`]');   // 'a[a]c'
'abc'.replace(/b/, "[$']");   // 'a[c]c'
'abc'.replace(/b/, '$$');     // 'a$c'
```

$1 and $<name> insert a group, $& the match, $` the text before it, $' the text after it, $$ a literal $.

## 12 split()

```
'a,b;c'.split(/[,;]/);    // ['a', 'b', 'c']
'aB'.split(/(?=[A-Z])/);  // ['a', 'B']
'a,,b'.split(/,/);        // ['a', '', 'b']
'a1b2c'.split(/(\d)/);
// ['a', '1', 'b', '2', 'c']
'a,,b'.split(/,/).filter(Boolean);
// ['a', 'b']
```

A capture group keeps the separators; a lookahead removes nothing. filter(Boolean) drops the empty strings.

## 13 Dynamic patterns

```
new RegExp('\\d+', 'g');      // /\d+/g
const alt = ['cat', 'dog'].join('|');
new RegExp(`\\b(${alt})\\b`, 'i');
// /\b(cat|dog)\b/i
const dot = RegExp.escape('a.b');
new RegExp(dot).test('axb');  // false
// before RegExp.escape:
s.replace(/[.*+?^${}()|[\]\\]/g, '\\$&');
```

In a string, double each backslash: '\d' is just 'd'. Escape user input first. RegExp.escape is ES2025 (Node 24+).

## 14 Unicode: u, v & \p{...}

```
/\p{L}+/u.exec('Zoë 42')[0];        // 'Zoë'
/\w+/.exec('Zoë 42')[0];            // 'Zo'
/\p{Script=Greek}/u.test('λ');      // true
/^.$/u.test('\u{1F600}');           // true
/[\p{L}--[a-z]]/v.test('a');        // false
'Zoë'.normalize('NFD')
  .replace(/\p{Diacritic}/gu, '');  // 'Zoe'
```

\p{...} needs u or v: \p{L} letters, \p{N} numbers. \w, \d and \b stay ASCII-only. With u, the dot matches a whole code point. v (ES2024) adds -- and && inside classes.

## 15 Catastrophic backtracking

```
const s = 'x'.repeat(28);
/(x+x+)+y/.test(s);     // false, slowly
/x+y/.test(s);          // false, at once
/(?=(x+))\1y/.test(s);  // false, at once
```

Nested quantifiers try every way to split the input before failing: 28 characters take seconds in Node 24, and each extra one doubles the time. (?=(x+))\1 emulates an atomic group.

## 16 Validation patterns

```
// email sanity check: one @, then a dot
/^[^\s@]+@[^\s@]+\.[^\s@]+$/;
// password: 8+ with lower, upper, digit
/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{8,}$/;
// 24-hour time HH:MM:SS
/^([01]\d|2[0-3]):[0-5]\d:[0-5]\d$/;
// hex color: fff, #fff or #ffffff
/^#?([a-f\d]{3}|[a-f\d]{6})$/i;
```

These check shape, not truth: the email pattern accepts a@b..c. Validate again on the server. For URLs, use new URL() instead of a regex.

## 17 Text recipes

```
// collapse whitespace
s.replace(/\s+/g, ' ').trim();
// camelCase to kebab-case
s.replace(/[A-Z]/g, '-$&').toLowerCase();
// snake_case to camelCase
s.replace(/_./g, m => m[1].toUpperCase());
// remove a repeated word
s.replace(/\b(\w+)\s+\1\b/gi, '$1');
```

Stripping tags with /<[^>]*>/g is not a sanitizer.

## 18 lastIndex with g & y

```
const re = /a/g;
re.test('a');     // true, lastIndex is 1
re.test('a');     // false, lastIndex is 0
re.test('a');     // true again
/b/y.test('ab');  // false: no b at index 0
```

With g or y, test and exec resume at lastIndex, so a reused regex alternates. Use a new literal or set lastIndex = 0.