**Difference between match and exec**

/regex/.exec() returns only the first match found, while "string".match() returns all of them if you use the g flag in the regex.

If the regular expression does not include the g flag, returns the same result as RegExp.exec(). The returned Array has an extra input property, which contains the original string that was parsed. In addition, it has an index property, which represents the zero-based index of the match in the string.

If the regular expression includes the g flag, the method returns an Array containing all matched substrings rather than match objects. Captured groups are not returned. If there were no matches, the method returns null.

If you want to obtain capture groups and the global flag is set, you need to use [RegExp.exec()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/exec) instead.

var myRe = /(\d)(\d)/g;

var str = '12 34';

var myArray;

while (myArray = myRe.exec(str)) {

console.log(myArray);

}

**Regex in javascript**

Difference between \* and + in words

console.log("ishan vimukthi vihanga kandage don".match(/\w\*/g));

console.log("ishan vimukthi vihanga kandage don".match(/\w+/g));

let sentence = "is somewhere waldo is hiding in this text";

console.log(/walDo/i.test(sentence));

console.log(sentence.match(/is/));

console.log(sentence.match(/is/g));

console.log(sentence.match(/\bis\b/g));

console.log(sentence.match(/\bi.\b/g));

**match eke g kiyala global danna naththam enne na okkoma**

var re = /\w+\s/g;

var str = 'fee fi fo fum';

var myArray = str.match(re);

console.log(myArray);

// ["fee ", "fi ", "fo "]

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the [exec](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/exec) and [test](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/test) methods of [RegExp](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp" \o "The RegExp constructor creates a regular expression object for matching text with a pattern.), and with the [match](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match), [matchAll](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/matchAll" \o "The matchAll() method returns an iterator of all results matching a string against a regular expression, including capturing groups.), [replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace), [search](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/search), and [split](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split) methods of [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String). This chapter describes JavaScript regular expressions.

**Creating a regular expression**

You construct a regular expression in one of two ways:

Using a regular expression literal, which consists of a pattern enclosed between slashes, as follows:

var re = /ab+c/;

Regular expression literals provide compilation of the regular expression when the script is loaded. If the regular expression remains constant, using this can improve performance.

Or calling the constructor function of the [RegExp](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp" \o "The RegExp constructor creates a regular expression object for matching text with a pattern.) object, as follows:

var re = new RegExp('ab+c');

Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

**Writing a regular expression pattern**

A regular expression pattern is composed of simple characters, such as /abc/, or a combination of simple and special characters, such as /ab\*c/ or /Chapter (\d+)\.\d\*/. The last example includes parentheses which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in [Using parenthesized substring matches](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#Using_parenthesized_substring_matches).

Using simple patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern /abc/ matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string 'Grab crab' because while it contains the substring 'ab c', it does not contain the exact substring 'abc'.

Using special characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, you can include special characters in the pattern. For example, to match a single 'a' followed by zero or more 'b's followed by 'c', you'd use the pattern /ab\*c/: the \* after 'b' means "0 or more occurrences of the preceding item." In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

The following pages and table provide a complete list and description of the special characters that can be used in regular expressions.

[**Assertions**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Assertions)

Indicates in some way that a match is possible. Assertions include look-ahead, look-behind, and conditional expressions.

[**Boundaries**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Boundaries)

Indicate the beginnings and endings of lines and words.

[**Character Classes**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Character_Classes)

Distinguishes kinds of characters such as, for example, distinguishing between letters and digits.

[**Groups and Ranges**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Groups_and_Ranges)

Indicates groups and ranges of expression characters.

[**Quantifiers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Quantifiers)

Indicates numbers of characters or expressions to match.

[**Unicode Property Escapes**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Unicode_Property_Escapes)

Distinguishes based on unicode character properties, for example, upper and lower case letters, math symbols, and punctuation.

| **Special characters in regular expressions.** | |
| --- | --- |
| **Character** | **Meaning** |
| [\](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backslash) | Matches according to the following rules:  A backslash that precedes a non-special character indicates that the next character is special and is not to be interpreted literally. For example, a 'b' without a preceding '\' generally matches lowercase 'b's wherever they occur — the character will be interpreted literally. But a sequence of '\b' doesn't match any character; it denotes a [word boundary](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#special-word-boundary).  A backslash that precedes a special character indicates that the next character is not special and should be interpreted literally. See "Escaping" below for details.  If you're using the RegExp constructor with a string, don't forget that backslash is an escape character in string literals, and so to put a backslash in the pattern, you need to escape it in the string literal. /[a-z]\s/i and new RegExp("[a-z]\\s", "i") create the same regular expression: an expression that searches for any letter in the range A-Z followed by a whitespace character (\s, see below). To include a *literal* backslash in an expression created via new RegExp with a string literal, you need to escape it at both the string literal level and the regular expression level: /[a-z]:\\/i and new RegExp("[a-z]:\\\\","i") create the same expression, which would match a string like "C:\". |
| [^](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-caret) | Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character.  For example, /^A/ does not match the 'A' in "an A", but does match the 'A' in "An E".  The '^' has a different meaning when it appears as the first character in a character set pattern. See [complemented character sets](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#special-negated-character-set) for details and an example. |
| [$](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-dollar) | Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character.  For example, /t$/ does not match the 't' in "eater", but does match it in "eat". |
| [\*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-asterisk) | Matches the preceding expression 0 or more times. Equivalent to {0,}.  For example, /bo\*/ matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled" but nothing in "A goat grunted". |
| [+](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-plus) | Matches the preceding expression 1 or more times. Equivalent to {1,}.  For example, /a+/ matches the 'a' in "candy" and all the a's in "caaaaaaandy", but nothing in "cndy". |
| [?](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-questionmark) | Matches the preceding expression 0 or 1 time. Equivalent to {0,1}.  For example, /e?le?/ matches the 'el' in "angel" and the 'le' in "angle" and also the 'l' in "oslo".  If used immediately after any of the quantifiers \*, +, ?, or {}, makes the quantifier non-greedy (matching the fewest possible characters), as opposed to the default, which is greedy (matching as many characters as possible). For example, applying /\d+/ to "123abc" matches "123". But applying /\d+?/ to that same string matches only the "1".  Also used in lookahead assertions, as described in the x(?=y) and x(?!y) entries of this table. |
| [.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-dot) | (The decimal point) matches any single character except the newline character, by default.  For example, /.n/ matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.  If the s ("dotAll") flag is set to true, it also matches newline characters. |
| [(x)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-capturing-parentheses) | Matches 'x' and remembers the match, as the following example shows. The parentheses are called *capturing parentheses*.  The '(foo)' and '(bar)' in the pattern /(foo) (bar) \1 \2/ match and remember the first two words in the string "foo bar foo bar". The \1 and \2 denote the first and second parenthesized substring matches - foo and bar, matching the string's last two words. Note that \1, \2, ..., \n are used in the matching part of the regex, for more information, see [\n](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#special-backreference) below. In the replacement part of a regex the syntax $1, $2, ..., $n must be used, e.g.: 'bar foo'.replace(/(...) (...)/, '$2 $1'). $& means the whole matched string. |
| [(?:x)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-capturing-parentheses) | Matches 'x' but does not remember the match. The parentheses are called *non-capturing parentheses*, and let you define subexpressions for regular expression operators to work with. Consider the sample expression /(?:foo){1,2}/. If the expression was /foo{1,2}/, the {1,2} characters would apply only to the last 'o' in 'foo'. With the non-capturing parentheses, the {1,2} applies to the entire word 'foo'. For more information, see [Using parentheses](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#Using_parentheses) below. |
| [x(?=y)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-lookahead) | Matches 'x' only if 'x' is followed by 'y'. This is called a lookahead.  For example, /Jack(?=Sprat)/ matches 'Jack' only if it is followed by 'Sprat'. /Jack(?=Sprat|Frost)/ matches 'Jack' only if it is followed by 'Sprat' or 'Frost'. However, neither 'Sprat' nor 'Frost' is part of the match results. |
| [x(?!y)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-negated-look-ahead) | Matches 'x' only if 'x' is not followed by 'y'. This is called a negated lookahead.  For example, /\d+(?!\.)/ matches a number only if it is not followed by a decimal point. The regular expression /\d+(?!\.)/.exec("3.141") matches '141' but not '3.141'. |
| [(?<=y)x](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-lookbehind) | Matches *x* only if *x* is preceded by *y*.This is called a lookbehind.  For example, /(?<=Jack)Sprat/ matches "Sprat" only if it is preceded by "Jack". /(?<=Jack|Tom)Sprat/ matches "Sprat" only if it is preceded by "Jack" or "Tom". However, neither "Jack" nor "Tom" is part of the match results.  (Added in ES2018, not yet supported in Firefox) |
| [(?<!y)x](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-negative-lookbehind) | Matches *x* only if *x* is not preceded by *y*.This is called a negated lookbehind.  For example, /(?<!-)\d+/ matches a number only if it is not preceded by a minus sign. /(?<!-)\d+/.exec('3') matches "3". /(?<!-)\d+/.exec('-3') match is not found because the number is preceded by the minus sign.  (Added in ES2018, not yet supported in Firefox) |
| [x|y](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-or) | Matches 'x', or 'y' (if there is no match for 'x').  For example, /green|red/ matches 'green' in "green apple" and 'red' in "red apple." The order of 'x' and 'y' matters. For example a\*|b matches the empty string in "b", but b|a\* matches "b" in the same string. |
| [{n}](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-quantifier) | Matches exactly n occurrences of the preceding expression. N must be a positive integer.  For example, /a{2}/ doesn't match the 'a' in "candy," but it does match all of the a's in "caandy," and the first two a's in "caaandy." |
| [{n,}](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#special-quantifier) | Matches at least n occurrences of the preceding expression. N must be a positive integer.  For example, /a{2,}/ will match "aa", "aaaa" and "aaaaa" but not "a" |
| [{n,m}](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-quantifier-range) | Where n and m are positive integers and n <= m. Matches at least n and at most m occurrences of the preceding expression. When m is omitted, it's treated as ∞.  For example, /a{1,3}/ matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it. |
| [[xyz]](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-character-set) | Character set. This pattern type matches any one of the characters in the brackets, including [escape sequences](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Using_special_characters_in_strings). Special characters like the dot(.) and asterisk (\*) are not special inside a character set, so they don't need to be escaped. You can specify a range of characters by using a hyphen, as the following examples illustrate.  The pattern [a-d], which performs the same match as [abcd], matches the 'b' in "brisket" and the 'c' in "city". The patterns /[a-z.]+/ and /[\w.]+/ match the entire string "test.i.ng". |
| [[^xyz]](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-negated-character-set) | A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen. Everything that works in the normal character set also works here.  For example, [^abc] is the same as [^a-c]. They initially match 'r' in "brisket" and 'h' in "chop." |
| [[\b]](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backspace) | Matches a backspace (U+0008). You need to use square brackets if you want to match a literal backspace character. (Not to be confused with \b.) |
| [\b](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-word-boundary) | Matches a *word boundary*. A word boundary matches the position between a word character followed by a non-word character, or between a non-word character followed by a word character, or the beginning of the string, or the end of the string. A word boundary is not a "character" to be matched; like an anchor, a word boundary is not included in the match. In other words, the length of a matched word boundary is zero. (Not to be confused with [\b].)  Examples using the input string "moon": /\bm/ matches, because the \b is at the beginning of the string; the \b in /oo\b/ does not match, because the \b is both preceded and followed by word characters; the \b in /oon\b/ matches, because it appears at the end of the string; the \b in /\w\b\w/ will never match anything, because it is both preceded and followed by a word character..  **Note:** JavaScript's regular expression engine defines a [specific set of characters](http://www.ecma-international.org/ecma-262/5.1/#sec-15.10.2.6) to be "word" characters. Any character not in that set is considered a non-word character. This set of characters is fairly limited: it consists solely of the Roman alphabet in both upper- and lower-case, decimal digits, and the underscore character. Accented characters, such as "é" or "ü" are, unfortunately, treated as non-word characters for the purposes of word boundaries, as are ideographic characters in general. |
| [\B](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-word-boundary) | Matches a non-*word boundary*. This matches the following cases:   * Before the first character of the string * After the last character of the string * Between two word characters * Between two non-word characters * The empty string   For example, /\B../ matches 'oo' in "noonday", and /y\B./ matches 'ye' in "possibly yesterday." |
| [\c](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-control)*[X](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-control)* | Where *X* is a character ranging from A to Z. Matches a control character in a string.  For example, /\cM/ matches control-M (U+000D) in a string. |
| [\d](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-digit) | Matches a digit character. Equivalent to [0-9].  For example, /\d/ or /[0-9]/ matches '2' in "B2 is the suite number." |
| [\D](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-digit) | Matches a non-digit character. Equivalent to [^0-9].  For example, /\D/ or /[^0-9]/ matches 'B' in "B2 is the suite number." |
| [\f](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-form-feed) | Matches a form feed (U+000C). |
| [\n](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-line-feed) | Matches a line feed (U+000A). |
| [\r](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-carriage-return) | Matches a carriage return (U+000D). |
| [\s](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-white-space) | Matches a white space character, including space, tab, form feed, line feed. Equivalent to [ \f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff].  For example, /\s\w\*/ matches ' bar' in "foo bar." |
| [\S](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-white-space) | Matches a character other than white space. Equivalent to [^ \f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff].  For example, /\S\*/ matches 'foo' in "foo bar." |
| [\t](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-tab) | Matches a tab (U+0009). |
| [\v](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-vertical-tab) | Matches a vertical tab (U+000B). |
| [\w](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-word) | Matches any alphanumeric character including the underscore. Equivalent to [A-Za-z0-9\_].  For example, /\w/ matches 'a' in "apple," '5' in "$5.28," and '3' in "3D." |
| [\W](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-word) | Matches any non-word character. Equivalent to [^A-Za-z0-9\_].  For example, /\W/ or /[^A-Za-z0-9\_]/ matches '%' in "50%." |
| [\](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backreference)*[n](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backreference)* | Where *n* is a positive integer, a back reference to the last substring matching the *n* parenthetical in the regular expression (counting left parentheses).  For example, /apple(,)\sorange\1/ matches 'apple, orange,' in "apple, orange, cherry, peach." |
| [\0](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-null) | Matches a NULL (U+0000) character. Do not follow this with another digit, because \0<digits> is an octal [escape sequence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Using_special_characters_in_strings). Instead use \x00. |
| [\xhh](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-hex-escape) | Matches the character with the code hh (two hexadecimal digits) |
| [\uhhhh](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-unicode-escape) | Matches the character with the code hhhh (four hexadecimal digits). |
| [\u{hhhh}](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-unicode-escape-es6) | (only when u flag is set) Matches the character with the Unicode value hhhh (hexadecimal digits). |

Escaping

If you need to use any of the special characters literally (actually searching for a '\*', for instance), you must escape it by putting a backslash in front of it. For instance, to search for 'a' followed by '\*' followed by 'b', you'd use /a\\*b/—the backslash "escapes" the '\*', making it literal instead of special.

Similarly, if you're writing a regular expression literal and need to match a slash ('/'), you need to escape that (otherwise, it terminates the pattern). For instance, to search for the string "/example/" followed by one or more alphabetic characters, you'd use /\/example\/[a-z]+/i—the backslashes before each slash make them literal.

To match a literal backslash, you need to escape the backslash. For instance, to match the string "C:\" where 'C' can be any letter, you'd use /[A-Z]:\\/—the first backslash escapes the one after it, so the expression searches for a single literal backslash.

If using the RegExp constructor with a string literal, remember that the backslash is an escape in string literals, so to use it in the regular expression, you need to escape it at the string literal level. /a\\*b/ and new RegExp("a\\\*b") create the same expression, which searches for 'a' followed by a literal '\*' followed by 'b'.

If escape strings are not already part of your pattern you can add them using [String.replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace" \o "The replace() method returns a new string with some or all matches of a pattern replaced by a replacement.):

function escapeRegExp(string) {

return string.replace(/[.\*+?^${}()|[\]\\]/g, '\\$&'); // $& means the whole matched string

}

The g after the regular expression is an option or flag that performs a global search, looking in the whole string and returning all matches. It is explained in detail below in [Advanced Searching With Flags](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#Advanced_searching_with_flags).

Using parentheses

Parentheses around any part of the regular expression pattern causes that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use, as described in [Using Parenthesized Substring Matches](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#Using_parenthesized_substring_matches).

For example, the pattern /Chapter (\d+)\.\d\*/ illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by one or more numeric characters (\d means any numeric character and + means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with \ means the pattern must look for the literal character '.'), followed by any numeric character 0 or more times (\d means numeric character, \* means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapter 3 and 4", because that string does not have a period after the '3'.

To match a substring without causing the matched part to be remembered, within the parentheses preface the pattern with ?:. For example, (?:\d+) matches one or more numeric characters but does not remember the matched characters.

**Working with regular expressions**

Regular expressions are used with the RegExp methods test and exec and with the String methods match, replace, search, and split. These methods are explained in detail in the [JavaScript reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference).

| **Methods that use regular expressions** | |
| --- | --- |
| **Method** | **Description** |
| [exec](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/exec) | A RegExp method that executes a search for a match in a string. It returns an array of information or null on a mismatch. |
| [test](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/test) | A RegExp method that tests for a match in a string. It returns true or false. |
| [match](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match) | A String method that returns an array containing all of the matches, including capturing groups, or null if no match is found. |
| [matchAll](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/matchAll) | A String method that returns an iterator containing all of the matches, including capturing groups. |
| [search](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/search) | A String method that tests for a match in a string. It returns the index of the match, or -1 if the search fails. |
| [replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace) | A String method that executes a search for a match in a string, and replaces the matched substring with a replacement substring. |
| [split](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split) | A String method that uses a regular expression or a fixed string to break a string into an array of substrings. |

When you want to know whether a pattern is found in a string, use the test or search method; for more information (but slower execution) use the exec or match methods. If you use exec or match and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, RegExp. If the match fails, the exec method returns null (which coerces to false).

In the following example, the script uses the exec method to find a match in a string.

var myRe = /d(b+)d/g;

var myArray = myRe.exec('cdbbdbsbz');

If you do not need to access the properties of the regular expression, an alternative way of creating myArray is with this script:

var myArray = /d(b+)d/g.exec('cdbbdbsbz');

// similar to "cdbbdbsbz".match(/d(b+)d/g); however,

// "cdbbdbsbz".match(/d(b+)d/g) outputs Array [ "dbbd" ], while

// /d(b+)d/g.exec('cdbbdbsbz') outputs Array [ 'dbbd', 'bb', index: 1, input: 'cdbbdbsbz' ].

(See [g different behaviors](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#g-different-behaviors) for further info about the different behaviors.)

If you want to construct the regular expression from a string, yet another alternative is this script:

var myRe = new RegExp('d(b+)d', 'g');

var myArray = myRe.exec('cdbbdbsbz');

With these scripts, the match succeeds and returns the array and updates the properties shown in the following table.

| **Results of regular expression execution.** | | | |
| --- | --- | --- | --- |
| **Object** | **Property or index** | **Description** | **In this example** |
| myArray |  | The matched string and all remembered substrings. | ['dbbd', 'bb', index: 1, input: 'cdbbdbsbz'] |
| index | The 0-based index of the match in the input string. | 1 |
| input | The original string. | "cdbbdbsbz" |
| [0] | The last matched characters. | "dbbd" |
| myRe | lastIndex | The index at which to start the next match. (This property is set only if the regular expression uses the g option, described in [Advanced Searching With Flags](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions#Advanced_searching_with_flags).) | 5 |
| source | The text of the pattern. Updated at the time that the regular expression is created, not executed. | "d(b+)d" |

As shown in the second form of this example, you can use a regular expression created with an object initializer without assigning it to a variable. If you do, however, every occurrence is a new regular expression. For this reason, if you use this form without assigning it to a variable, you cannot subsequently access the properties of that regular expression. For example, assume you have this script:

var myRe = /d(b+)d/g;

var myArray = myRe.exec('cdbbdbsbz');

console.log('The value of lastIndex is ' + myRe.lastIndex);

// "The value of lastIndex is 5"

However, if you have this script:

var myArray = /d(b+)d/g.exec('cdbbdbsbz');

console.log('The value of lastIndex is ' + /d(b+)d/g.lastIndex);

// "The value of lastIndex is 0"

The occurrences of /d(b+)d/g in the two statements are different regular expression objects and hence have different values for their lastIndex property. If you need to access the properties of a regular expression created with an object initializer, you should first assign it to a variable.

Using parenthesized substring matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, /a(b)c/ matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the Array elements [1], ..., [n].

The number of possible parenthesized substrings is unlimited. The returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

The following script uses the [replace()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace) method to switch the words in the string. For the replacement text, the script uses the $1 and $2 in the replacement to denote the first and second parenthesized substring matches.

var re = /(\w+)\s(\w+)/;

var str = 'John Smith';

var newstr = str.replace(re, '$2, $1');

console.log(newstr);

// "Smith, John"

Advanced searching with flags

Regular expressions have six optional flags that allow for functionality like global and case insensitive searching. These flags can be used separately or together in any order, and are included as part of the regular expression.

| **Regular expression flags** | |
| --- | --- |
| **Flag** | **Description** |
| g | Global search. |
| i | Case-insensitive search. |
| m | Multi-line search. |
| s | Allows . to match newline characters. (Added in ES2018, not yet supported in Firefox) |
| u | "unicode"; treat a pattern as a sequence of unicode code points |
| y | Perform a "sticky" search that matches starting at the current position in the target string. See [sticky](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/sticky) |

To include a flag with the regular expression, use this syntax:

var re = /pattern/flags;

or

var re = new RegExp('pattern', 'flags');

Note that the flags are an integral part of a regular expression. They cannot be added or removed later.

For example, re = /\w+\s/g creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

var re = /\w+\s/g;

var str = 'fee fi fo fum';

var myArray = str.match(re);

console.log(myArray);

// ["fee ", "fi ", "fo "]

You could replace the line:

var re = /\w+\s/g;

with:

var re = new RegExp('\\w+\\s', 'g');

and get the same result.

The behavior associated with the '**g**' flag is different when the .exec() method is used.  (The roles of "class" and "argument" get reversed: In the case of .match(), the string class (or data type) owns the method and the regular expression is just an argument, while in the case of .exec(), it is the regular expression that owns the method, with the string being the argument.  Contrast *str.match(re)* versus *re.exec(str)*.)  The '**g**' flag is used with the **.exec()** method to get iterative progression.

var xArray; while(xArray = re.exec(str)) console.log(xArray);

// produces:

// ["fee ", index: 0, input: "fee fi fo fum"]

// ["fi ", index: 4, input: "fee fi fo fum"]

// ["fo ", index: 7, input: "fee fi fo fum"]

The m flag is used to specify that a multiline input string should be treated as multiple lines. If the m flag is used, ^ and $ match at the start or end of any line within the input string instead of the start or end of the entire string.

**Examples**

The following examples show some uses of regular expressions.

Changing the order in an input string

The following example illustrates the formation of regular expressions and the use of string.split() and string.replace(). It cleans a roughly formatted input string containing names (first name last) separated by blanks, tabs and exactly one semicolon. Finally, it reverses the name order (last name first) and sorts the list.

// The name string contains multiple spaces and tabs,

// and may have multiple spaces between first and last names.

var names = 'Orange Carrot ;Fred Barney; Helen Rigby ; Bill Abel ; Chris Hand ';

var output = ['---------- Original String\n', names + '\n'];

// Prepare two regular expression patterns and array storage.

// Split the string into array elements.

// pattern: possible white space then semicolon then possible white space

var pattern = /\s\*;\s\*/;

// Break the string into pieces separated by the pattern above and

// store the pieces in an array called nameList

var nameList = names.split(pattern);

// new pattern: one or more characters then spaces then characters.

// Use parentheses to "memorize" portions of the pattern.

// The memorized portions are referred to later.

pattern = /(\w+)\s+(\w+)/;

// Below is the new array for holding names being processed.

var bySurnameList = [];

// Display the name array and populate the new array

// with comma-separated names, last first.

//

// The replace method removes anything matching the pattern

// and replaces it with the memorized string—the second memorized portion

// followed by a comma, a space and the first memorized portion.

//

// The variables $1 and $2 refer to the portions

// memorized while matching the pattern.

output.push('---------- After Split by Regular Expression');

var i, len;

for (i = 0, len = nameList.length; i < len; i++) {

output.push(nameList[i]);

bySurnameList[i] = nameList[i].replace(pattern, '$2, $1');

}

// Display the new array.

output.push('---------- Names Reversed');

for (i = 0, len = bySurnameList.length; i < len; i++) {

output.push(bySurnameList[i]);

}

// Sort by last name, then display the sorted array.

bySurnameList.sort();

output.push('---------- Sorted');

for (i = 0, len = bySurnameList.length; i < len; i++) {

output.push(bySurnameList[i]);

}

output.push('---------- End');

console.log(output.join('\n'));

// produces:

//

// ---------- Original String

//

// Orange Carrot ;Fred Barney; Helen Rigby ; Bill Abel ; Chris Hand

//

// ---------- After Split by Regular Expression

// Orange Carrot

// Fred Barney

// Helen Rigby

// Bill Abel

// Chris Hand

// ---------- Names Reversed

// Carrot, Orange

// Barney, Fred

// Rigby, Helen

// Abel, Bill

// Hand, Chris

// ---------- Sorted

// Abel, Bill

// Barney, Fred

// Carrot, Orange

// Hand, Chris

// Rigby, Helen

// ---------- End

Using special characters to verify input

In the following example, the user is expected to enter a phone number. When the user presses the "Check" button, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script shows a message thanking the user and confirming the number. If the number is invalid, the script informs the user that the phone number is not valid.

Within non-capturing parentheses (?: , the regular expression looks for three numeric characters \d{3} OR | a left parenthesis \( followed by three digits \d{3}, followed by a close parenthesis \), (end non-capturing parenthesis )), followed by one dash, forward slash, or decimal point and when found, remember the character ([-\/\.]), followed by three digits \d{3}, followed by the remembered match of a dash, forward slash, or decimal point \1, followed by four digits \d{4}.

The Change event activated when the user presses Enter sets the value of RegExp.input.

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<meta http-equiv="Content-Script-Type" content="text/javascript">

<script type="text/javascript">

var re = /(?:\d{3}|\(\d{3}\))([-\/\.])\d{3}\1\d{4}/;

function testInfo(phoneInput) {

var OK = re.exec(phoneInput.value);

if (!OK)

window.alert(phoneInput.value + ' isn\'t a phone number with area code!');

else

window.alert('Thanks, your phone number is ' + OK[0]);

}

</script>

</head>

<body>

<p>Enter your phone number (with area code) and then click "Check".

<br>The expected format is like ###-###-####.</p>

<form action="#">

<input id="phone"><button onclick="testInfo(document.getElementById('phone'));">Check</button>

</form>

</body>

</html>