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. When the regular expression will remain constant, use this for better performance.

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.

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, the pattern includes special characters. For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is followed by zero or more 'b's (`*` means 0 or more occurrences of the preceding item) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

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

Special characters in regular expressions.

Character	Meaning
\	Matches according to the following rules: A backslash that precedes a non-special character indicates that the next character is

Special characters in regular expressions.	
Character	**Meaning**
	special and is not to be interpreted literally. For example, a '`b`' without a preceding '\' generally matches lowercase 'b's wherever they occur. But a '`\b`' by itself doesn't match any character; it forms the special word boundary character. A backslash that precedes a special character indicates that the next character is not special and should be interpreted literally. For example, the pattern `/a*/` relies on the special character '`*`' to match 0 or more a's. By contrast, the pattern `/a\*/` removes the specialness of the '`*`' to enable matches with strings like 'a*'. Do not forget to escape \ itself while using the RegExp("pattern") notation because \ is also an escape character in strings.
^	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 for details and an example.
$	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".
*	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".
+	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".
?	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".

Special characters in regular expressions.

Character	Meaning
	Also used in lookahead assertions, as described in the `x(?=y)` and `x(?!y)` entries of this table.
`.`	(The decimal point) matches any single character except the newline character. For example, `/.n/` matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
`(x)`	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` in the pattern match the string's last two words. Note that `\1, \2, \n` are used in the matching part of the regex. In the replacement part of a regex the syntax `$1, $2, $n` must be used, e.g.: `'bar foo'.replace( /(...) (...)/, '$2 $1' ).`
`(?:x)`	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'.
`x(?=y)`	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)`	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'.
`x|y`	Matches either 'x' or 'y'. For example, `/green|red/` matches 'green' in "green apple" and 'red' in "red apple."
`{n}`	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,m}`	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 ∞.

Special characters in regular expressions.	
Character	**Meaning**
	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]`	Character set. This pattern type matches any one of the characters in the brackets, including escape sequences. 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]`	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]`	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`	Matches a word boundary. A word boundary matches the position where a word character is not followed or preceeded by another word-character. Note that a matched 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: `/\bm/` matches the 'm' in "moon" ; `/oo\b/` does not match the 'oo' in "moon", because 'oo' is followed by 'n' which is a word character; `/oon\b/` matches the 'oon' in "moon", because 'oon' is the end of the string, thus not followed by a word character; `/\w\b\w/` will never match anything, because a word character can never be followed by both a non-word and a word character. **Note:** JavaScript's regular expression engine defines a specific set of characters to be "word" characters. Any character not in that set is considered a word break. 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 word breaks.
`\B`	Matches a non-word boundary. This matches a position where the previous and next character are of the same type: Either both must be words, or both must be non-words. The beginning and end of a string are considered non-words.

Special characters in regular expressions.

Character	Meaning
	For example, `/\B../` matches 'oo' in "noonday", and `/y\B./` matches 'ye' in "possibly yesterday."
`\cX`	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`	Matches a digit character. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."
`\D`	Matches any non-digit character. Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches 'B' in "B2 is the suite number."
`\f`	Matches a form feed (U+000C).
`\n`	Matches a line feed (U+000A).
`\r`	Matches a carriage return (U+000D).
`\s`	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to `[ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`. For example, `/\s\w*/` matches ' bar' in "foo bar."
`\S`	Matches a single character other than white space. Equivalent to `[^ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`. For example, `/\S\w*/` matches 'foo' in "foo bar."
`\t`	Matches a tab (U+0009).
`\v`	Matches a vertical tab (U+000B).
`\w`	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`	Matches any non-word character. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^A-Za-z0-9_]/` matches '%' in "50%."
`\n`	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."

Special characters in regular expressions.

Character	Meaning
`\0`	Matches a NULL (U+0000) character. Do not follow this with another digit, because `\0<digits>` is an octal escape sequence.
`\xhh`	Matches the character with the code hh (two hexadecimal digits)
`\uhhhh`	Matches the character with the code hhhh (four hexadecimal digits).

Escaping user input to be treated as a literal string within a regular expression can be accomplished by simple replacement:

```
function escapeRegExp(string){
  return string.replace(/[.*+?^${}()|[\]\\]/g, "\\$&");
}
```

Using parentheses

Parentheses around any part of the regular expression pattern cause 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.

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.

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()` 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);
```

This prints "Smith, John".

Advanced searching with flags

Regular expressions have four optional flags that allow for 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.
y	Perform a "sticky" search that matches starting at the current position in the target string. See `sticky`

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

```
var re = /pattern/flags;
```

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

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.