RegEx

Regular expressions

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()` and `test()` methods of `RegExp`, and with the `match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()`, and `split()` methods of `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:`const 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` object, as follows:`const 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 groups. Note: If you are already familiar

with the forms of a regular expression, you may also read the cheatsheet for a quick lookup for a specific pattern/construct.

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 the exact sequence `"abc"` occurs (all characters 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"`, this pattern will match the substring `"abbbbc"`.

The following pages provide lists of the different special characters that fit into each category, along with descriptions and examples.

Assertions

Assertions include boundaries, which indicate the beginnings and endings of lines and words, and other patterns indicating in some way that a match is possible (including look-ahead, look-behind, and conditional expressions).

Character classes

Distinguish different types of characters. For example, distinguishing between letters and digits.

Groups and backreferences
Groups group multiple patterns as a whole, and capturing groups provide extra submatch information when using a regular expression pattern to match against a string. Backreferences refer to a previously captured group in the same regular expression.

Quantifiers
Indicate numbers of characters or expressions to match.

Unicode property escapes
Distinguish based on unicode character properties, for example, upp

er- and lower-case letters, math symbols, and punctuation.

If you want to look at all the special characters that can be used in regular expressions in a single table, see the following:

Special characters in regular expressions.

Characters / constructs	Corresponding article
`\`, `.`, `\cX`, `\d`, `\D`, `\f`, `\n`, `\r`, `\s`, `\S`, `\t`, `\v`, `\w`, `\W`, `\0`, `\xhh`, `\uhhhh`, `\uhhhhh`, `[\b]`	Character classes
`^`, `$`, `x(?=y)`, `x(?!y)`, `(?<=y)x`, `(?<!y)x`, `\b`, `\B`	Assertions
`(x)`, `(?:x)`, `(?<Name>x)`, `x|y`, `[xyz]`, `[^xyz]`, `\Number`	Groups and backreferences
`*`, `+`, `?`, `x{n}`, `x{n,}`, `x{n,m}`	Quantifiers
`\p{UnicodeProperty}`, `\P{UnicodeProperty}`	Unicode property escapes

Note: A larger cheatsheet is also available (only aggregating parts of those individual articles).

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.prototype.replace()`:

```
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.
Why isn't this built into JavaScript? There is a proposal to add such a function to RegExp.

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.
See Groups and backreferences for more details.

Using regular expressions in JavaScript

Regular expressions are used with the `RegExp` methods `test()` and `exec()` and with the `String` methods `match()`, `replace()`, `search()`, and `split()`.

Method	Description
`exec()`	Executes a search for a match in a string. It returns an array of information or `null` on a mismatch.
`test()`	Tests for a match in a string. It returns `true` or `false`.
`match()`	Returns an array containing all of the matches, including capturing groups, or `null` if no match is found.
`matchAll()`	Returns an iterator containing all of the matches, including capturing groups.
`search()`	Tests for a match in a string. It returns the index of the match, or `-1` if the search fails.
`replace()`	Executes a search for a match in a string, and replaces the matched substring with a replacement substring.

replaceAll()	Executes a search for all matches in a string, and replaces the matched substrings with a replacement substring.
split()	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()` methods; 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.

```
const myRe = /d(b+)d/g;
const 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:

```
const myArray = /d(b+)d/g.exec('cdbbdbsbz');
// similar to 'cdbbdbsbz'.match(/d(b+)d/g);
however,
// 'cdbbdbsbz'.match(/d(b+)d/g) outputs [ "dbbd" ]
// while /d(b+)d/g.exec('cdbbdbsbz') outputs
[ 'dbbd', 'bb', index: 1, input: 'cdbbdbsbz' ]
```

(See Using the global search flag with `exec()` for further info about the different behaviors.)

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

```
const myRe = new RegExp('d(b+)d', 'g');
const 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.)	`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:

```
const myRe = /d(b+)d/g;
const 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:

```
const 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.

Advanced searching with flags

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

Flag	Description	Corresponding property
`d`	Generate indices for substring matches.	`hasIndices`
`g`	Global search.	`global`
`i`	Case-insensitive search.	`ignoreCase`
`m`	Allows ^ and $ to match newline characters.	`multiline`
`s`	Allows `.` to match newline characters.	`dotAll`
`u`	"Unicode"; treat a pattern as a sequence of Unicode code points.	`unicode`

`y`	Perform a "sticky" search that matches starting at the current position in the target string.	`sticky`

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

```
const re = /pattern/flags;
```

or

```
const 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.

```
const re = /\w+\s/g;
const str = 'fee fi fo fum';
const myArray = str.match(re);
console.log(myArray);

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

You could replace the line:

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

with:

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

and get the same result.

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.

Using the global search flag with exec()

`RegExp.prototype.exec()` method with the `g` flag returns each match and its position iteratively.

```
const str = 'fee fi fo fum';
const re = /\w+\s/g;

console.log(re.exec(str)); // ["fee ", index: 0,
input: "fee fi fo fum"]
console.log(re.exec(str)); // ["fi ", index: 4,
input: "fee fi fo fum"]
console.log(re.exec(str)); // ["fo ", index: 7,
input: "fee fi fo fum"]
console.log(re.exec(str)); // null
```

In contrast, `String.prototype.match()` method returns all matches at once, but without their position.

```
console.log(str.match(re)); // ["fee ", "fi ", "fo
"]
```

Using unicode regular expressions

The "u" flag is used to create "unicode" regular expressions; that is, regular expressions which support matching against unicode text. This is mainly accomplished through the use of Unicode property escapes, which are supported only within "unicode" regular expressions.

For example, the following regular expression might be used to match against an arbitrary unicode "word":

```
/\p{L}*/u
```

There are a number of other differences between unicode and non-unicode regular expressions that one should be aware of:

- Unicode regular expressions do not support so-called "identity escapes"; that is, patterns where an escaping backslash is not needed and effectively ignored. For example, `/\a/` is a valid regular expression matching the letter 'a', but `/\a/u` is not.
- Curly brackets need to be escaped when not used as quantifiers. For example, `/{/` is a valid regular expression matching the curly bracket '{', but `/{/u` is not — instead, the bracket should be escaped and `/\{/u` should be used instead.
- The `-` character is interpreted differently within character classes. In particular, for Unicode regular expressions, `-` is interpreted as a literal `-` (and not as part of a range) only if it appears at the start or end of the character class. For example, `/[\w-:]/` is a valid regular expression matching a word character, a `-`, or `:`, but `/[\w-:]/u` is an invalid regular expression, as `\w` to `:` is not a well-defined range of characters.

Unicode regular expressions have different execution behavior as well. `RegExp.prototype.unicode` contains more explanation about this.

Examples

Note: Several examples are also available in:

- The reference pages for `exec()`, `test()`, `match()`, `matchAll()`, `search()`, `replace()`, `split()`
- The guide articles: character classes, assertions, groups and backreferences, quantifiers, Unicode property escapes

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.

The regular expression looks for:

1. the beginning of the line of data: `^`
2. followed by three numeric characters `\d{3}` OR `|` a left parenthesis `\(`, followed by three digits `\d{3}`, followed by a close parenthesis `\)`, in a non-capturing group `(?:)`
3. followed by one dash, forward slash, or decimal point in a capturing group `()`
4. followed by three digits `\d{3}`
5. followed by the match remembered in the (first) captured group `\1`
6. followed by four digits `\d{4}`
7. followed by the end of the line of data: `$`

HTML

```
<p>
  Enter your phone number (with area code) and
then click "Check".
  <br />
  The expected format is like ###-###-####.
</p>
<form id="form">
  <input id="phone" />
  <button type="submit">Check</button>
</form>
<p id="output"></p>
```

JavaScript

```
const form = document.querySelector('#form');
const input = document.querySelector('#phone');
const output = document.querySelector('#output');

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

function testInfo(phoneInput) {
  const ok = re.exec(phoneInput.value);

  output.textContent = ok
    ? `Thanks, your phone number is ${ok[0]}`
    : `${phoneInput.value} isn't a phone number
with area code!`;
}

form.addEventListener('submit', (event) => {
  event.preventDefault();
  testInfo(input);
});
```

Result

Enter your phone number (with area code) and then click "Check".
The expected format is like ###-###-####.

Check at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Tools

RegExr

An online tool to learn, build, & test Regular Expressions.

Regex tester

An online regex builder/debugger

Regular expressions (regex) are a useful programming tool. They are key to efficient text processing. Knowing how to solve problems using regex is helpful to you as a developer and improves your productivity.

Regular expression syntax cheatsheet

This page provides an overall cheat sheet of all the capabilities of `RegExp` syntax by aggregating the content of the articles in the `RegExp` guide. If you need more information on a specific topic, please follow the link on the corresponding heading to access the full article or head to the guide.

Character classes

Character classes distinguish kinds of characters such as, for example, distinguishing between letters and digits.

Characters	Meaning
`[xyz]` `[a-c]`	A character class. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets, it is taken as a literal hyphen to be included in the character class as a normal character. For example, `[abcd]` is the same as `[a-d]`. They match the "b" in "brisket", and the "a" or the "c" in "arch", but not the "-" (hyphen) in "non-profit". For example, `[abcd-]` and `[-abcd]` match the "b" in "brisket", the "a" or the "c" in "arch", and the "-" (hyphen) in "non-profit". For example, `[\w-]` is the same as `[A-Za-z0-9_-]`. They both match any of the characters in "no_reply@example-server.com" except for the "@" and the ".".

`[^xyz]` `[^a-c]`	A negated or complemented character class. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets, it is taken as a literal hyphen to be included in the character class as a normal character. For example, `[^abc]` is the same as `[^a-c]`. They initially match "o" in "bacon" and "h" in "chop". Note: The ^ character may also indicate the beginning of input.
`.`	Has one of the following meanings: • Matches any single character *except* line terminators: `\n`, `\r`, `\u2028` or `\u2029`. For example, `/.y/` matches "my" and "ay", but not "yes", in "yes make my day". • Inside a character class, the dot loses its special meaning and matches a literal dot. Note that the `m` multiline flag doesn't change the dot behavior. So to match a pattern across multiple lines, the character class `[^]` can be used — it will match any character including newlines. The `s` "dotAll" flag allows the dot to also match line terminators.
`\d`	Matches any digit (Arabic numeral). Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches "2" in "B2 is the suite number".
`\D`	Matches any character that is not a digit (Arabic numeral). Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches "B" in "B2 is the suite number".
`\w`	Matches any alphanumeric character from the basic Latin alphabet, 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 character that is not a word character from the basic Latin alphabet. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^A-Za-z0-9_]/` matches "%" in "50%".

`\s`	Matches a single white space character, including space, tab, form feed, line feed, and other Unicode spaces. 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`	Matches a single 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\w*/` matches "foo" in "foo bar".
`\t`	Matches a horizontal tab.
`\r`	Matches a carriage return.
`\n`	Matches a linefeed.
`\v`	Matches a vertical tab.
`\f`	Matches a form-feed.
`[\b]`	Matches a backspace. If you're looking for the word-boundary character (`\b`), see Boundaries.
`\0`	Matches a NUL character. Do not follow this with another digit.
`\c`*`X`*	Matches a control character using caret notation, where "X" is a letter from A–Z (corresponding to codepoints `U+0001`–`U+001F`). For example, `/\cM/` matches "\r" in "\r\n".
`\x`*`hh`*	Matches the character with the code *`hh`* (two hexadecimal digits).
`\u`*`hhhh`*	Matches a UTF-16 code-unit with the value *`hhhh`* (four hexadecimal digits).
`\u{`*`hhhh`*`}` or `\u{`*`hhhhh`*`}`	(Only when the `u` flag is set.) Matches the character with the Unicode value `U+`*`hhhh`* or `U+`*`hhhhh`* (hexadecimal digits).

`\`	Indicates that the following character should be treated specially, or "escaped". It behaves one of two ways. • For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, `/b/` matches the character "b". By placing a backslash in front of "b", that is by using `/\b/`, the character becomes special to mean match a word boundary. • For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, "*" is a special character that means 0 or more occurrences of the preceding character should be matched; for example, `/a*/` means match 0 or more "a"s. To match `*` literally, precede it with a backslash; for example, `/a\*/` matches "a*". Note that some characters like `:`, `–`, `@`, etc. neither have a special meaning when escaped nor when unescaped. Escape sequences like `\:`, `\–`, `\@` will be equivalent to their literal, unescaped character equivalents in regular expressions. However, in regular expressions with the unicode flag, these will cause an *invalid identity escape* error. This is done to ensure backward compatibility with existing code that uses new escape sequences like `\p` or `\k`. Note: To match this character literally, escape it with itself. In other words to search for `\` use `/\\/`.
x|*y*	Disjunction: Matches either "x" or "y". Each component, separated by a pipe (`|`), is called an *alternative*. For example, `/green|red/` matches "green" in "green apple" and "red" in "red apple". Note: A disjunction is another way to specify "a set of choices", but it's not a character class. Disjunctions are not atoms — you need to use a group to make it part of a bigger pattern. `[abc]` is functionally equivalent to `(?:a|b|c)`.

Assertions

Assertions include boundaries, which indicate the beginnings and endings of lines and words, and other patterns indicating in some

way that a match is possible (including look-ahead, look-behind, and conditional expressions).

Boundary-type assertions

Characters	Meaning
`^`	Matches the 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 first "A" in "An A". Note: This character has a different meaning when it appears at the start of a character class.
`$`	Matches the end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` does not match the "t" in "eater", but does match it in "eat".
`\b`	Matches a word boundary. This is the position where a word character is not followed or preceded by another word-character, such as between a letter and a space. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero. 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. To match a backspace character (`[\b]`), see Character Classes.
`\B`	Matches a non-word boundary. This is a position where the previous and next character are of the same type: Either both must be words, or both must be non-words, for example between two letters or between two spaces. The beginning and end of a string are considered non-words. Same as the matched word boundary, the matched non-word boundary is also not included in the match. For example, `/\Bon/` matches "on" in "at noon", and `/ye\B/` matches "ye" in "possibly yesterday".

Other assertions

Note: The `?` character may also be used as a quantifier.

Characters	Meaning
`x(?=y)`	Lookahead assertion: Matches "x" only if "x" is followed by "y". 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)`	Negative lookahead assertion: Matches "x" only if "x" is not followed by "y". For example, `/\d+(?!\.)/` matches a number only if it is not followed by a decimal point. `/\d+(?!\.)/.exec('3.141')` matches "141" but not "3".
`(?<=y)x`	Lookbehind assertion: Matches "x" only if "x" is preceded by "y". 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.
`(?<!y)x`	Negative lookbehind assertion: Matches "x" only if "x" is not preceded by "y". 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.

Groups and backreferences

Groups and backreferences indicate groups of expression characters.

Characters	Meaning

`(x)`	Capturing group: Matches *x* and remembers the match. For example, `/(foo)/` matches and remembers "foo" in "foo bar". A regular expression may have multiple capturing groups. In results, matches to capturing groups typically in an array whose members are in the same order as the left parentheses in the capturing group. This is usually just the order of the capturing groups themselves. This becomes important when capturing groups are nested. Matches are accessed using the index of the result's elements (`[1], ..., [n]`) or from the predefined `RegExp` object's properties (`$1, ..., $9`). Capturing groups have a performance penalty. If you don't need the matched substring to be recalled, prefer non-capturing parentheses (see below). `String.prototype.match()` won't return groups if the `/.../g` flag is set. However, you can still use `String.prototype.matchAll()` to get all matches.
`(?<Name>x)`	Named capturing group: Matches "x" and stores it on the groups property of the returned matches under the name specified by `<Name>`. The angle brackets (< and >) are required for group name. For example, to extract the United States area code from a phone number, we could use `/\((?<area>\d\d\d)\)/`. The resulting number would appear under `matches.groups.area`.
`(?:x)`	Non-capturing group: Matches "x" but does not remember the match. The matched substring cannot be recalled from the resulting array's elements (`[1], ..., [n]`) or from the predefined `RegExp` object's properties (`$1, ..., $9`).
`\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".

\k<Name>	A back reference to the last substring matching the Named capture group specified by `<Name>`. For example, `/(?<title>\w+), yes \k<title>/` matches "Sir, yes Sir" in "Do you copy? Sir, yes Sir!". Note: `\k` is used literally here to indicate the beginning of a back reference to a Named capture group.

Quantifiers

Quantifiers indicate numbers of characters or expressions to match.

Note: In the following, *item* refers not only to singular characters, but also includes character classes, Unicode property escapes, groups and backreferences.

Characters	Meaning
*x**	Matches the preceding item "x" 0 or more times. For example, `/bo*/` matches "boooo" in "A ghost booooed" and "b" in "A bird warbled", but nothing in "A goat grunted".
x+	Matches the preceding item "x" 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the "a" in "candy" and all the "a"'s in "caaaaaaandy".
x?	Matches the preceding item "x" 0 or 1 times. For example, `/e?le?/` matches the "el" in "angel" and the "le" in "angle." If used immediately after any of the quantifiers `*`, `+`, `?`, or `{}`, makes the quantifier non-greedy (matching the minimum number of times), as opposed to the default, which is greedy (matching the maximum number of times).
x{*n*}	Where "n" is a positive integer, matches exactly "n" occurrences of the preceding item "x". For example, `/a{2}/` doesn't match the "a" in "candy", but it matches all of the "a"'s in "caandy", and the first two "a"'s in "caaandy".
x{*n*,}	Where "n" is a positive integer, matches at least "n" occurrences of the preceding item "x". For example, `/a{2,}/` doesn't match the "a" in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy".

x{*n*,*m*}	Where "n" is 0 or a positive integer, "m" is a positive integer, and *m* > *n*, matches at least "n" and at most "m" occurrences of the preceding item "x". For example, /a{1,3}/ matches nothing in "cndy", the "a" in "candy", the 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.
*x**? *x*+? *x*?? *x*{n}? *x*{n,}? *x*{n,m}?	By default quantifiers like * and + are "greedy", meaning that they try to match as much of the string as possible. The ? character after the quantifier makes the quantifier "non-greedy": meaning that it will stop as soon as it finds a match. For example, given a string like "some <foo> <bar> new </bar> </foo> thing": • /<.*>/ will match "<foo> <bar> new </bar> </foo>" • /<.*?>/ will match "<foo>"

Unicode property escapes

Unicode property escapes allow for matching characters based on their Unicode properties.

```
// Non-binary values
/\p{UnicodePropertyValue}/
/\p{UnicodePropertyName=UnicodePropertyValue}/

// Binary and non-binary values
/\p{UnicodeBinaryPropertyName}/

// Negation: \P is negated \p
/\P{UnicodePropertyValue}/
/\P{UnicodeBinaryPropertyName}/
```

`UnicodeBinaryPropertyName`
The name of a binary property.
E.g.: `ASCII`, `Alpha`, `Math`, `Diacritic`, `Emoji`, `Hex_Digit`, `Math`, `White_space`, etc. See Unicode Data PropList.txt for more info.

`UnicodePropertyName`
The name of a non-binary property:

- General_Category (`gc`)

- Script (sc)
- Script_Extensions (scx)

See also PropertyValueAliases.txt

`UnicodePropertyValue`
One of the tokens listed in the Values section, below. Many values have aliases or shorthand (e.g. the value `Decimal_Number` for the `General_Category` property may be written `Nd`, `digit`, or `Decimal_Number`). For most values, the `UnicodePropertyName` part and equals sign may be omitted. If a `UnicodePropertyName` is specified, the value must correspond to the property type given.

———————————————————————————————

What Are Regular Expressions?

Regular expressions are patterns that allow you to describe, match, or parse text. With regular expressions, you can do things like find and replace text, verify that input data follows the format required, and and other similar things.

Here's a scenario: you want to verify that the telephone number entered by a user on a form matches a format, say, ###-###-#### (where # represents a number). One way to solve this could be:

Notice how we've refactored the code using regex. Amazing right? That is the power of regular expressions.

What Are Regular Expressions?

Regular expressions are patterns that allow you to describe, match, or parse text. With regular expressions, you can do things like find and replace text, verify that input data follows the format required, and and other similar things.

Here's a scenario: you want to verify that the telephone number entered by a user on a form matches a format, say, ###-###-#### (where # represents a number). One way to solve this could be:

```
function isPattern(userInput) {
  if (typeof userInput !== 'string' || userInput.length !== 12) {
    return false;
  }
  for (let i = 0; i < userInput.length; i++) {
    let c = userInput[i];
    switch (i) {
      case 0:
      case 1:
      case 2:
      case 4:
      case 5:
      case 6:
      case 8:
      case 9:
      case 10:
      case 11:
        if (c < 0 || c > 9) return false;
        break;
      case 3:
      case 7:
        if (c !== '-') return false;
        break;
    }
  }
  return true;
}
```

Alternatively, we can use a regular expression here like this:

```
function isPattern(userInput) {
  return /^\d{3}-\d{3}-\d{4}$/.test(userInput);
}
```

How to Create A Regular Expression

In JavaScript, you can create a regular expression in either of two ways:

- Method #1: using a regular expression literal. This consists of a pattern enclosed in forward slashes. You can write this with or without a flag (we will see what flag means shortly). The syntax is as follows:

```
const regExpLiteral = /pattern/;          // Without flags

const regExpLiteralWithFlags = /pattern/; // With flags
```

The forward slashes `/.../` indicate that we are creating a regular expression pattern, just the same way you use quotes `" "` to create a string.

- Method #2: using the RegExp constructor function. The syntax is as follows:

```
new RegExp(pattern [, flags])
```

Here, the pattern is enclosed in quotes, the same as the flag parameter, which is optional.

So when do you use each of these pattern?

You should use a regex literal when you know the regular expression pattern at the time of writing the code.

On the other hand, use the Regex constructor if the regex pattern is to be created dynamically. Also, the regex constructor lets you write a pattern using a template literal, but this is not possible with the regex literal syntax.

What are Regular Expression Flags?

Flags or modifiers are characters that enable advanced search features including case-insensitive and global searching. You can use them individually or collectively. Some commonly used ones are:

- `g` is used for global search which means the search will not return after the first match.
- `i` is used for case-insensitive search meaning that a match can occur regardless of the casing.
- `m` is used for multiline search.
- `u` is used for Unicode search.

Let's look at some regular expression patterns using both syntaxes.

How to use a regular expression literal:

```
// Syntax: /pattern/flags

const regExpStr = 'Hello world! hello there';

const regExpLiteral = /Hello/gi;

console.log(regExpStr.match(regExpLiteral));

// Output: ['Hello', 'hello']
```

Note that if we did not flag the pattern with `i`, only `Hello` will be returned.

The pattern `/Hello/` is an example of a simple pattern. A simple pattern consists of characters that must appear literally in the target text. For a match to occur, the target text must follow the same sequence as the pattern.

For example, if you re-write the text in the previous example and try to match it:

```
const regExpLiteral = /Hello/gi;

const regExpStr = 'oHell world, ohell there!';

console.log(regExpStr.match(regExpLiteral));

// Output: null
```

We get *null* because the characters in the string do not appear as specified in the pattern. So a literal pattern such as `/hello/`, means *h* followed by *e* followed by *l* followed by *l* followed by *o*, exactly like that.

How to use a regex constructor:

```
// Syntax: RegExp(pattern [, flags])

const regExpConstructor = new RegExp('xyz', 'g'); // With flag -g

const str = 'xyz xyz';

console.log(str.match(regExpConstructor));

// Output: ['xyz', 'xyz']
```

Here, the pattern `xyz` is passed in as a string same as the flag. Also both occurrences of `xyz` got matched because we passed in the -g flag. Without it, only the first match will be returned.

We can also pass in dynamically created patterns as template literals using the constructor function. For example:

```
const pattern = prompt('Enter a pattern');
// Suppose the user enters 'xyz'

const regExpConst = new RegExp(`${pattern}`, 'gi');

const str = 'xyz XYZ';

console.log(str.match(regExpConst)); // Output: ['xyz', 'XYZ']
```

How to Use Regular Expression Special Characters

A **special character** in a regular expression is a character with a reserved meaning. Using special characters, you can do more than just find a direct match.

For example, if you want to match a character in a string that may or may not appear once or multiple times, you can do this with special characters. These characters fit into different subgroups that perform similar functions.

Let's take a look at each subgroup and the characters that go with them.

Anchors and Boundaries:

Anchors are metacharacters that match the start and end of a line of text they are examining. You use them to assert where a boundary should be.

The two characters used are `^` and `$`.

- `^` matches the start of a line and anchors a literal at the beginning of that line. For example:

```
const regexPattern1 = /^cat/;

console.log(regexPattern1.test('cat and mouse')); // Output: true

console.log(regexPattern1.test('The cat and mouse')); // Output: false because the lir

// Without the ^ in the pattern, the output will return true
// because we did not assert a boundary.

const regexPattern2 = /cat/;

console.log(regexPattern2.test('The cat and mouse')); // Output: true
```

- `$` matches the end of a line and anchors a literal at the end of that line. For example:

```
const regexPattern = /cat$/;

console.log(regexPattern.test('The mouse and the cat')); // Output: true

console.log(regexPattern.test('The cat and mouse')); // Output: false
```

Note that anchors characters `^` and `$` **match just the position of the characters in the pattern** and not the actual characters themselves.

Word Boundaries are metacharacters that match the start and end position of a word – a sequence of alphanumeric characters. You can think of them as a word-based version of `^` and `$`. You use the metacharacters `b` and `B` to assert a word boundary.

- `\b` matches the start or end of a word. The word is matched according to the position of the metacharacter. Here's an example:

```
// Syntax 1: /\b.../ where .... represents a word.

// Search for a word that begins with the pattern ward
const regexPattern1 = /\bward/gi;

const text1 = 'backward Wardrobe Ward';

console.log(text1.match(regexPattern1)); // Output: ['Ward', 'Ward']

// Syntax 2: /...\b/

// Search for a word that ends with the pattern ward
const regexPattern2 = /ward\b/gi;

const text2 = 'backward Wardrobe Ward';

console.log(text2.match(regexPattern2)); // Output: ['ward', 'Ward']

// Syntax 3: /\b....\b/

// Search for a stand-alone word that begins and end with the pattern ward
const regexPattern3 = /\bward\b/gi;

const text3 = 'backward Wardrobe Ward';

console.log(text3.match(regexPattern3)); // Output: ['Ward']
```

- `\B` is opposite of `\b` . It matches every position `\b` doesn't.

Shortcodes for Other Metacharacters:

In addition to the metacharacters we have looked at, here are some of the most commonly used ones:

- `\d` – matches any decimal digit and is shorthand for [0-9].
- `\w` – matches any alphanumeric character which could be a letter, a digit, or an underscore. `\w` is shorthand for [A-Za-z0-9_].
- `\s` – matches any white space character.
- `\D` – matches any non-digit and is the same as [^0-9.]
- `\W` – matches any non-word (that is non-alphanumeric) character and is shorthand for [^A-Za-z0-9_].
- `\S` – matches a non-white space character.
- `.` – matches any character.

What is a Character Class?

A character class is used to match any one of several characters in a particular position. To denote a character class, you use square brackets `[]` and then list the characters you want to match inside the brackets.

Let's look at an example:

```
// Find and match a word with two alternative spellings

const regexPattern = /ambi[ea]nce/;

console.log(regexPattern.test('ambiance')); // Output: true

console.log(regexPattern.test('ambiance')); // Output: true

// The regex pattern interprets as:  find a followed by m, then b,
// then i, then either e or a, then n, then c, and then e.
```

What is a Negated Character Class?

If you add a caret symbol inside a character class like this `[^...]`, it will match any character that is not listed inside the square brackets. For example:

```
const regexPattern = /[^bc]at/;

console.log(regexPattern.test('bat')); // Output: false

console.log(regexPattern.test('cat')); // Output: false

console.log(regexPattern.test('mat')); // Output: true
```

What is a Range?

A hyphen `-` indicates range when used inside a character class. Suppose you want to match a set of numbers, say [0123456789], or a set of characters, say[abcdefg]. You can write it as a range like this, [0-9] and [a-g], respectively.

What is Alternation?

Alternation is yet another way you can specify a set of options. Here, you use the pipe character `|` to match any of several subexpressions. Either of the subexpressions is called an **alternative**.

The pipe symbol means 'or', so it matches a series of options. It allows you combine subexpressions as alternatives.

For example, `(x|y|z)a` will match `xa` or `ya`, or `za`. In order to limit the reach of the alternation, you can use parentheses to group the alternatives together.

Without the parentheses, `x|y|za` would mean `x` or `y` or `za`. For example:

```
const regexPattern = /(Bob|George)\sClan/;

console.log(regexPattern.test('Bob Clan')); // Output: true

console.log(regexPattern.test('George Clan')); // Output: true
```

What are Quantifiers and Greediness?

Quantifiers denote how many times a character, a character class, or group should appear in the target text for a match to occur. Here are some peculiar ones:

- `+` will match any character it is appended to if the character appears at least once. For example:

```
const regexPattern = /hel+o/;

console.log(regexPattern.test('helo'));          // Output:true

console.log(regexPattern.test('hellllllllllo')); // Output: true

console.log(regexPattern.test('heo'));           // Output: false
```

- `*` is similar to the + character but with a slight difference. When you append * to a character, it means you want to match any number of that character including none. Here's an example:

```
const regexPattern = /hel*o/;

console.log(regexPattern.test('helo'));    // Output: true

console.log(regexPattern.test('hellllo')); // Output: true

console.log(regexPattern.test('heo'));     // Output: true

// Here the * matches 0 or any number of 'l'
```

- `?` implies "optional". When you append it to a character, it means the character may or may not appear. For example:

```
const regexPattern = /colou?r/;

console.log(regexPattern.test('color'));  // Output: true

console.log(regexPattern.test('colour')); // Output: true

// The ? after the character u makes u optional
```

- `{N}`, when appended to a character or character class, specifies how many of the character we want. For example `/\d{3}/` means match three consecutive digits.
- `{N,M}` is called the interval quantifier and is used to specify a range for the minimum and maximum possible match. For example `/\d{3, 6}/` means match a minimum of 3 and a maximum of 6 consecutive digits.
- `{N, }` denotes an open-ended range. For example `/\d{3, }/` means match any 3 or more consecutive digits.

What is Greediness in Regex?

All quantifiers by default are **greedy**. This means that they will try to match all possible characters.

To remove this default state and make them non-greedy, you append a `?` to the operator like this `+?`, `*?`, `{N}?`, `{N,M}?`.....and so on.

What are Grouping and Backreferencing?

We previously looked at how we can limit the scope of alternation using the parentheses.

What if you want to use a quantifier like `+` or `*` on more than one character at a time – say a character class or group? You can group them together as a whole using the parentheses before appending the quantifier, just like in this example:

```
const regExp = /abc+(xyz+)+/i;

console.log(regExp.test('abcxyzzzzXYZ')); // Output: true
```

Here's what the pattern means: The first + matches the c of abc, the second + matches the z of xyz, and the third + matches the subexpression xyz, which will match if the sequence repeats.

Backreferencing allows you to match a new pattern that is the same as a previously matched pattern in a regular expression. You also use parentheses for backreferencing because it can remember a previously matched subexpression it encloses (that is, the captured group).

However, it is possible to have more than one captured group in a regular expression. So, to backreference any of the captured group, you use a number to identify the parentheses.

Suppose you have 3 captured groups in a regex and you want to backreference any of them. You use `\1`, `\2`, or `\3`, to refer to the first, second, or third parentheses. To number the parentheses, you start counting the open parentheses from the left.

Let's look at some examples:

`(x)` matches x and remembers the match.

```
const regExp = /(abc)bar\1/i;

// abc is backreferenced and is anchored at the same position as \1
console.log(regExp.test('abcbarAbc')); // Output: true

console.log(regExp.test('abcbar')); // Output: false
```

`(?:x)` matches x but does not recall the match. Also, \n (where n is a number) does not remember a previously captured group, and will match as a literal. Using an example:

```
const regExp = /(?:abc)bar\1/i;

console.log(regExp.test('abcbarabc')); // Output: false

console.log(regExp.test('abcbar\1')); // Output: true
```

The Escape Rule

A metacharacter has to be escaped with a backslash if you want it to appear as a literal in your regular expression. By escaping a metacharacter in regex, the metacharacter loses its special meaning.

Regular Expression Methods

The `test()` method

We have used this method a number of times in this article. The `test()` method compares the target text with the regex pattern and returns a boolean value accordingly. If there is a match, it returns true, otherwise it returns false.

```
const regExp = /abc/i;

console.log(regExp.test('abcdef')); // Output: true

console.log(regExp.test('bcadef')); // Output: false
```

The `exec()` method

The `exec()` method compares the target text with the regex pattern. If there's a match, it returns an array with the match – otherwise it returns null. For example:

```
const regExp = /abc/i;

console.log(regExp.exec('abcdef'));
// Output: ['abc', index: 0, input: 'abcdef', groups: undefined]

console.log(regExp.exec('bcadef'));
// Output: null
```

Also, there are string methods that accept regular expressions as a parameter like `match()`, `replace()`, `replaceAll()`, `matchAll()`, `search()`, and `split()`.

Regex Examples

Here are some examples to reinforce some of the concepts we've learned in this article.

First example: How to use a regex pattern to match an email address:

```
const regexPattern = /^[(\w\d\W)+]+@[\w+]+\.[\w+]+$/i;

console.log(regexPattern.test('abcdef123@gmailcom'));
// Output: false, missing dot

console.log(regexPattern.test('abcdef123gmail.'));
// Output: false, missing end literal 'com'

console.log(regexPattern.test('abcdef123@gmail.com'));
// Output: true, the input matches the pattern correctly
```

Let's interpret the pattern. Here's what's happening:

- `/` represents the start of the regular expression pattern.
- `^` checks for the start of a line with the characters in the character class.
- `[(\w\d\W)+ ]+` matches any word, digit and non-word character in the character class at least once. Notice how the parentheses were used to group the characters before adding the quantifier. This is same as this `[\w+\d+\W+]+` .
- `@` matches the literal @ in the email format.
- `[\w+]+` matches any word character in this character class at least once.
- `\.` escapes the dot so it appears as a literal character.
- `[\w+]+$` matches any word character in this class. Also this character class is anchored at the end of the line.
- `/` - ends the pattern

Alright, next example: how to match a URL with format http://example.com or https://www.example.com:

```
const pattern = /^[https?]+:\/\/((w{3}.)?[\w+]+)\.[\w+]+$/i;

console.log(pattern.test('https://www.example.com'));
// Output: true

console.log(pattern.test('http://example.com'));
// Output: true

console.log(pattern.test('https://example'));
// Output: false
```

Let's also interpret this pattern. Here's what's happening:

- `/...../` represents the start and end of the regex pattern
- `^` asserts for the start of the line
- `[https?]+` matches the characters listed at least once, however `?` makes 's' optional.
- `:` matches a literal semi-colon.
- `\/\/` escapes the two forward slashes.
- `(w{3}.)?` matches the character w 3 times and the dot that follows immediately. However, this group is optional.
- `[\w+]+` matches character in this class at least once.
- `\.` escapes the dot
- `[\w+]+$` matches any word character in this class. Also this character class is anchored at the end of the line.

Conclusion

In this article, we looked at the fundamentals of regular expressions. We also explained some regular expression patterns, and practiced with a few examples.

There's more to regular expressions beyond this article. To help you learn more about regular expressions, here are some resources you can read through:

- Regular Expression
- Learn Regex crash course
- Regular Expression Tutorial
- Regular Expression Cheatsheet