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`.

Creating a regular expressions

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/;
```

- 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.

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 the numbers of characters or expressions to match.

Unicode property escapes

Distinguish based on unicode character properties, for example, upper- 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:

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

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
}
```

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.

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");
```

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"; 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. 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.