Regular Expressions and String Manipulation Playground

Sooner or later every programmer will use Regular Expressions in their code. To me they are hard to understand and reason… this post will try to clear up some of my confusion and provide examples of what these regular expressions look like.

Writing regular expressions

The first step is writing regular expressions. We have two ways of doing it: literals and constructors.

Regular expression literals

The simplest way to write regular expressions is to use them as a literal value in a constant or variable, the pattern is enclosed in slashes (`/`).

```
const myRE = /a+bc/;
```

Literal regular expressions are evaluated when the script is loaded. Use RegExp literals when you know what the regular expression will be.

Regular expression constructors

We can also build regular expressions using the RegExp constructor where we put the regular expression that we want to test in parenthesis.

```
const myRE = new RegExp('a+bc');
```

The constructors are evaluated when the script runs. Use constructors when you’re not certain what the expression will be or when you’re working with user input.

Modifying the regular expression: special characters and creating patterns

There are times when patterns like those we discussed are enough for our needs but there are times when we need to build more complicated expressions. We may want to match portions of a URL or make sure that the protocol used for the URL is `https://`.

There are special characters that we can use to enhace the expressions to perform specific tasks.

The table below, taken from MDN's [Writing a regular expression pattern]() shows the characters you can use in regular expressions and what they do.

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 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 denotes a word boundary. ▪ 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\*/` denotes the '`*`' as not special, enabling 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.

Character	Meaning
	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.
`$`	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` 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

Character	Meaning
	matches only the "`1`". 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, 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)`	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. 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)`	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'.

Character	**Meaning**
`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'.
`(?<=`*y*`)`*x*	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.
`(?<!`*y*`)`*x*	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.

Character	Meaning
`x|y`	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}`	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,}`	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}`	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]`	Character set. This pattern type matches any one of the characters in the brackets, including escape sequences. Special characters like the

Character	Meaning
	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 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;

Character	Meaning
	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 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`	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`*X*	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]`.

Character	Meaning
	For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."
`\D`	Matches a 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 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`	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`	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_]`.

Character	Meaning
	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."
`\0`	Matches a NULL (U+0000) character. Do not follow this with another digit, because `\0<digits>` is an octal (base 8) sequence. Instead use `\x00`.
`\xhh`	Matches the character with the code hh (two hexadecimal digits)

Goingh back to finding a regular expression that tests if our URL is to a secure site we can do the following:

```
const secureURL = /https:\/\//;
```

We’ll use the regular expression when we look at how to test using regular expressions in the next section.

Additional flags for regular expressions

One last set of flags to worry about. These are additional parameters for the expression that will allow additional flexibility beyond what special characters allow.

The table below show what these additional flags are and what they do.

Regular expression flags

Flag	Description
`g`	Global search.
`i`	Case-insensitive search.
`m`	Multi-line search.
`s`	Allows `.` to match newline characters.
`u`	"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.

The flags are appended to the end of the regular expression and are a part of it (no adding or removing flags after the regular expression) and they will change the way we build the regular expressions we want to use.

The literal expression now looks like this:

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

And the constructor changes to the one below

```
const re = new RegExp('pattern', 'flags');
```

Matching text against regular expressions

We have multiple ways to test strings against our regular expressions. Which one to use depends on what results you want to accomplish and how much information about the matches and the results you want to keep and use.

The methods are listed below:

Methods that use regular expressions

Method	Description
`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`	A `RegExp` method that tests for a match in a string. It returns true or false.
`match`	A `String` method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
`matchAll`	A `String` method that returns an iterator containing all of the matches, including capturing groups.
`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`	A `String` method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
`split`	A `String` method that uses a regular expression or a fixed string to break a string into an array of substrings.

What happens when the string has Unicode Characters?

Unicode special characters for regular expressions.

Character	Meaning
`\uhhhh`	Matches the character with the code hhhh (four hexadecimal digits).
`\u{hhhh}`	(only when u flag is set) Matches the character with the Unicode value hhhh (hexadecimal digits).

Making life easier: groups

Matching multiple items in the string

String.prototype.matchAll

A common use case of global (g) or sticky (y) regular expressions is applying it to a string and iterating through all of the matches. The new String.prototype.matchAll API makes this easier than ever before, especially for regular expressions with capture groups:

```
const string = 'Favorite GitHub repos: tc39/ecma262 v8/v8.dev';
const regex = /\b(?<owner>[a-z0-9]+)\/(?<repo>[a-z0-9\.]+)\b/g;

for (const /\b(?<owner>[a-z0-9]+)\/(?<repo>[a-z0-9\.]+)\b/g(`${match[0]} 
  console.log(`→ owner: ${match.groups.owner}`);
  console.log(`→ repo: ${match.groups.repo}`);
}

`→ owner: ${match.groups.owner}`// Output:
//
// tc39/ecma262 at 23 with 'Favorite GitHub repos: tc39/ecma262 v8/v8.dev
// → owner: tc39
// → repo: ecma262
// v8/v8.dev at 36 with 'Favorite GitHub repos: tc39/ecma262 v8/v8.dev'
// → owner: v8
// → repo: v8.dev
```

Links, resources and Ideas for further experimentation

- https://alligator.io/js/regular-expressions-for-regular-people/
- https://www.smashingmagazine.com/2019/02/regexp-features-regular-expressions/